Министерство образования и науки Российской Федерации
НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ЭФФЕКТИВНОЕ ПРОГРАММИРОВАНИЕ СОВРЕМЕННЫХ МИКРОПРОЦЕССОРОВ

Утверждено
Редакционно-издательским советом университета
в качестве учебного пособия

НОВОСИБИРСК
2014

УДК 004.31-181.48(075.8)
Э 949

Коллектив авторов
В.П. Маркова, С.Е. Киреев, М.Б. Остапкевич, В.А. Перепелкин

Рецензенты:
д-р техн. наук, профессор, Я.И. Фет
канд. техн. наук, доцент В.Д. Корнеев

Работа подготовлена
на кафедре параллельных вычислительных технологий

Э 949 **Эффективное программирование современных микропроцессоров**: учеб. пособие / В.П. Маркова, С.Е. Киреев, М.Б. Остапкевич, В.А. Перепелкин. – Новосибирск: Изд-во НГТУ, 2014 – 148 с.

ISBN 978-5-7782-2391-2

Пособие посвящено изучению особенностей архитектуры современных микропроцессоров, которые влияют на скорость выполнения прикладных программ. Для закрепления теоретического материала по курсу «Архитектура ЭВМ и ВС» предлагаются практические работы. Их цель – сформировать практические навыки разработки эффективных программ с учетом организации иерархической памяти, наличия векторных расширений и других особенностей архитектуры.

Учебное пособие предназначено для студентов I курса факультета ФПМИ направлений 010400.62 «Прикладная математика и информатика» и 010500.62 «Математическое обеспечение и администрирование информационных систем».

ВВЕДЕНИЕ

Практика программирования показывает, что типичная программа использует лишь малую часть теоретически достижимой вычислительной мощности современных компьютеров**.** Одна из главных причин этого в том, что программа написана без учёта особенностей организации компьютера. Такое положение вещей свидетельствует о важности понимания принципов работы компьютера на уровне архитектуры и программного обеспечения. Более того, по мере совершенствования компьютеров их программирование усложняется, поэтому важно сформировать у студента регулярный взгляд на архитектуру компьютера. Тогда он сможет адаптировать свои знания и навыки программирования к быстрой смене компьютерных архитектур.

В настоящее время существует большое количество литературы по архитектуре и организации современных микропроцессоров. Чаще всего её содержание сводится к описанию конкретных архитектур и организаций компьютеров. В настоящем учебном пособии основное внимание уделяется:

- рассмотрению особенностей организации современных компьютеров, которые влияют на скорость выполнения программ;
- подходам к разработке программ с учетом организации компьютера.

Для лучшего усвоения теоретического материала и формирования базы практических навыков программирования с учетом организации компьютера в пособии имеется несколько практических работ.

1. ВВЕДЕНИЕ В АРХИТЕКТУРУ КОМПЬЮТЕРА

1.1 ОПРЕДЕЛЕНИЕ АРХИТЕКТУРЫ И ОРГАНИЗАЦИИ КОМПЬЮТЕРА

Под *архитектурой* компьютера принято понимать логическое представление компьютера с точки зрения программиста. Архитектура определяет организацию памяти, набор команд, форматы представления данных,

способы адресации памяти, механизмы ввода/вывода, а также правила функционирования компьютера. Примерами современных архитектур являются x86-64, ARM, POWER.

Каждая архитектура может иметь несколько аппаратных реализаций. Аппаратная реализация компьютера называется *организацией* или *микроархитектурой.* Микроархитектура определяет структуру компьютера, а именно, набор компонентов компьютера, их связи, функциональные возможности каждого компонента (например, количество арифметических логических устройств, число стадий конвейера, размер аппаратного регистрового файла или разрядность шины между оперативной памятью и процессором).

В процессе эволюции архитектуры, предложенной фон Нейманом [1], сформировались несколько классов архитектур: CISC, RISC, VLIW. CISC – архитектура со сложным набором команд, большим число форматов команд, режимов адресации и малым регистровым файлом. Она ориентирована на написание эффективно работающих программ на языке Ассемблера. RISC – архитектура с сокращенным набором команд. Имеется малое число простых форматов этих команд. Доступ к оперативной памяти производится отдельными командами. Команды обработки данных могут работать только с регистрами. Имеется большой регистровый файл. Архитектура ориентирована на использование языков высокого уровня. VLIW – это развитие RISC архитектуры, в котором реализован параллелизм на уровне команд, который выявляется компилятором.

Архитектура более консервативна по сравнению с микроархитектурой, так как на основе архитектуры сформирован большой фонд алгоритмов и программ, и существенные изменения в ней привели бы к необходимости вносить модификации в эти программы. Микроархитектура, напротив, не связана этими ограничениями. Она эволюционировала более свободно и динамично, обеспечивая для заданной архитектуры все большую и большую производительность.

В рамках одной архитектуры по мере развития технологии производства СБИС (сверхбольших интегральных схем) появляются новые более производительные микроархитектуры. Увеличение их производительности достигается за счет возможности размещения на кристалле процессора все большего числа функциональных устройств.

Например, в рамках архитектуры x86 реализовано много микроархитектур. В первых микроархитектурах (8086, 8088) отсутствовали кэш-память, виртуальная память, команды выполнялись последовательно на одном конвейере. Развитие технологий построения СБИС позволило разместить на кристалле большое количество новых функциональных устройств. В микроархитектуре i386 введена кэш-память и страничная виртуальная память. В i486 добавлен конвейер, позволяющий завершать выполнение одной команды на каждом такте даже для команд, которые выполняются в течение нескольких тактов. В P5 появилась суперскалярная организация, включающая два конвейера для целочисленных операций, раздельные кэш-памяти для инструкций и данных. В микроархитектуре P6 были добавлены функциональные устройства спекулятивного выполнения команд, выполнения векторных команд, выполнение команд вне порядка, аппаратный регистровый файл, позволяющий переименовывать программные регистры. В NetBurst число стадий конвейера увеличено до 20. В современной микроархитектуре Haswell количество арифметико-логических устройств увеличено до четырех.

Иногда к организации также относят такие характеристики компьютера, которые не влияют на логику его устройства (тактовая частота, техпроцесс и т.п.). В данном пособии они вынесены за рамки понятия организации.

1.2. АРХИТЕКТУРНЫЕ ПРИНЦИПЫ ПОСТРОЕНИЯ КОМПЬЮТЕРА ФОН НЕЙМАНА

Несмотря на радикальное отличие современных компьютеров от своих предшественников, принципы, на основе которых создавались архитектуры

компьютеров в 40–50-х годах, используются при их построении и сегодня. Эти принципы были введены фон Нейманом в 1945 при работе над проектом EDVAC [26]. Звучат они следующим образом.

- **Принцип программного управления**. Алгоритм решения задачи должен быть представлен в виде программы, состоящей из последовательности команд. Каждая команда должна принадлежать некоторому набору команд, реализуемых компьютером. Команды выполняются последовательно. Именно эта последовательность команд и управляет работой компьютера.
- **Принцип хранимой программы.** Команды представляются в числовой форме и хранятся в оперативной памяти вместе с данными, а не задаются аппаратными средствами. Кроме того, в отличие от раздельного хранения команд и данных, это позволяет экономно распределить память между командами и данными.
- **Синхронное функционирование в ритме, задаваемом тактовым генератором.** Команды выполняются последовательно, каждая за определенный квант времени, называемый *тактом*. Продолжительность такта фиксирована и определяется частотой *тактового генератора*, или *тактовой частотой*.
- **Принцип условного перехода.** В наборе команд компьютера имеются специальные команды условных переходов. В зависимости от своего операнда, команда условного перехода может выбрать инструкцию в программе, которую требуется выполнять далее. Таким образом, возможно, организовывать циклы, итерационные процессы и т. д.
- **Принцип использования двоичной системы счисления для представления информации.** Наиболее технологичной для аппаратной реализации оказалась двоичная система счисления, поэтому в компьютере все числа хранятся и обрабатываются именно в этой системе счисления. К логическим схемам, построенным для этой системы счисления, может быть применен аппарат булевой алгебры.

- **Принцип иерархичности запоминающих устройств**. Причиной его введения стало несоответствие в стоимости и быстродействии различных типов памяти. Этот принцип предписывает располагать программы и данные для долговременного хранения на дешевой медленной памяти большого объема, а программы и данные, используемые в процессе вычислений, на дорогой быстрой памяти малого объема.

1.3. КОМПЬЮТЕР ФОН НЕЙМАНА, ЕГО УЗКИЕ МЕСТА И УСОВЕРШЕНСТВОВАНИЯ

Компьютер фон Неймана включает в себя четыре основных компонента (Рис. 1):

- оперативную память (ОП);
- устройство управления (УУ);
- арифметико-логическое устройство (АЛУ);
- устройства ввода-вывода и внешнюю память.

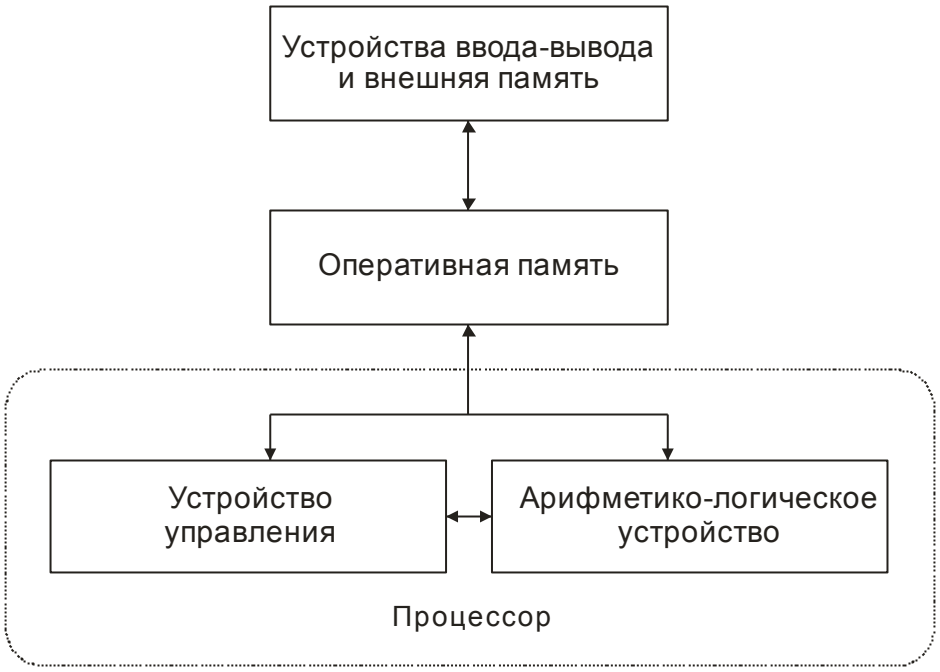

Рис. 1. Упрощенная схема компьютера фон Неймана

Оперативная память (основная память) обеспечивает хранение выполняемых команд и обрабатываемых данных. Она состоит из множества ячеек. У каждой ячейки есть уникальный, однозначно ее определяющий идентификатор, называемый *адресом*. Адреса расположены в строгой последовательности от нуля до предельного значения. Такой способ адресации памяти называется *глобальным.*

В каждой ячейке оперативной памяти в каждый момент времени хранится упорядоченный набор битов. *Ячейка* - это минимальная адресуемая единица информации, т.е. компьютер может прочитать или записать значение в ячейку только целиком. Почти во всех современных компьютерах размер ячейки равен одному байту, т.е. восьми битам.

Большинство современных компьютеров способно производить операции не над одним байтом, а сразу над последовательностями из нескольких байт. Такая группа байт называется *словом*. Его длина в битах определяет один из основных параметров архитектуры компьютера – *разрядность*. Так компьютер со словами, состоящими из четырех байт, имеет 32-разрядную архитектуру, а со словами из восьми байт – 64-разрядную.

Оперативная память - это память с *произвольным доступом*, т. е. в любой момент по адресу может быть доступна произвольная ячейка памяти вне зависимости от того, к каким ячейкам оперативной памяти осуществлялся доступ ранее. При чтении ячейки оперативной памяти по определенному адресу её значение считывается, а оригинал остается в ячейке. При записи ранее находившееся значение в оперативной памяти стирается, и вместо него записывается новое значение.

Кроме оперативной памяти в компьютере есть и другие виды памяти, которые отличаются как по организации, так и по времени доступа. Так, в компьютере есть еще дополнительная (внешняя) память. Она используется для долговременного хранения информации. Ее объем существенно выше, а скорость доступа – намного ниже, чем у оперативной памяти. Доступ к ней осуществляется через канал ввода-вывода.

В процессоре имеется некоторое количество регистров, образующих регистровую память. *Регистр* – это ячейка памяти, расположенная непосредственно в процессоре. Как следствие, она очень быстрая, т.е. имеет очень малое время доступа. Регистры в процессоре физически объединены в так называемый *регистровый файл*. В отличие от ячеек оперативной памяти, регистры имеют размер в несколько байт и идентифицируются не адресами, а именами. В регистрах хранятся операнды и управляющая информация, используемые при вычислениях. Некоторые регистры – многофункциональные, некоторые выполняют какие-то специфические функции. Отдельного рассмотрения заслуживает регистр, называемый *счетчиком команд*, который содержит адрес следующей выполняемой команды в оперативной памяти. Содержимое счетчика изменяется каждый раз, когда команда загружается из оперативной памяти.

Устройство управления и арифметико-логическое устройство образуют *центральный процессор* (далее просто *процессор*). УУ организует пошаговое выполнение программы. Оно последовательно загружает машинный код очередной команды из оперативной памяти, распознаёт его и посылает управляющий сигнал в АЛУ на выполнение той или иной операции. Арифметико-логическое устройство выполняет арифметико-логические операции над операндами, находящимися в памяти или в регистрах. Результат выполнения операции также записывается в оперативную память или в регистры.

Процессор имеет фиксированное множество команд, которое называется *набором команд* или *инструкций*. Каждая выполняемая команда программы задает выполняемую операцию, адреса операндов (исходных данных), над которыми она выполняется, и адрес, по которому должен быть помещен результат операции. Формат простейшей команды приведен на Рис. 2. Битовое представление команды содержит три поля. Каждое поле представляет собой совокупность двоичных разрядов, кодирующих часть команды: это код операции, адрес результата операции, по которому он будет помещен, и адреса операндов (1-го и 2-го).

Рис. 2. Формат команды процессора

В типичном современном процессоре есть такие команды, как копирование данных между памятью и регистрами, арифметические, логические и побитовые операции, команды перехода и команды ввода/вывода данных с внешних устройств.

Функционирование процессора (выполнение программы) состоит в циклическом выполнении двух шагов:

- *выборка* команды (Рис. 3а) и
- *выполнение* команды (Рис. 3б).

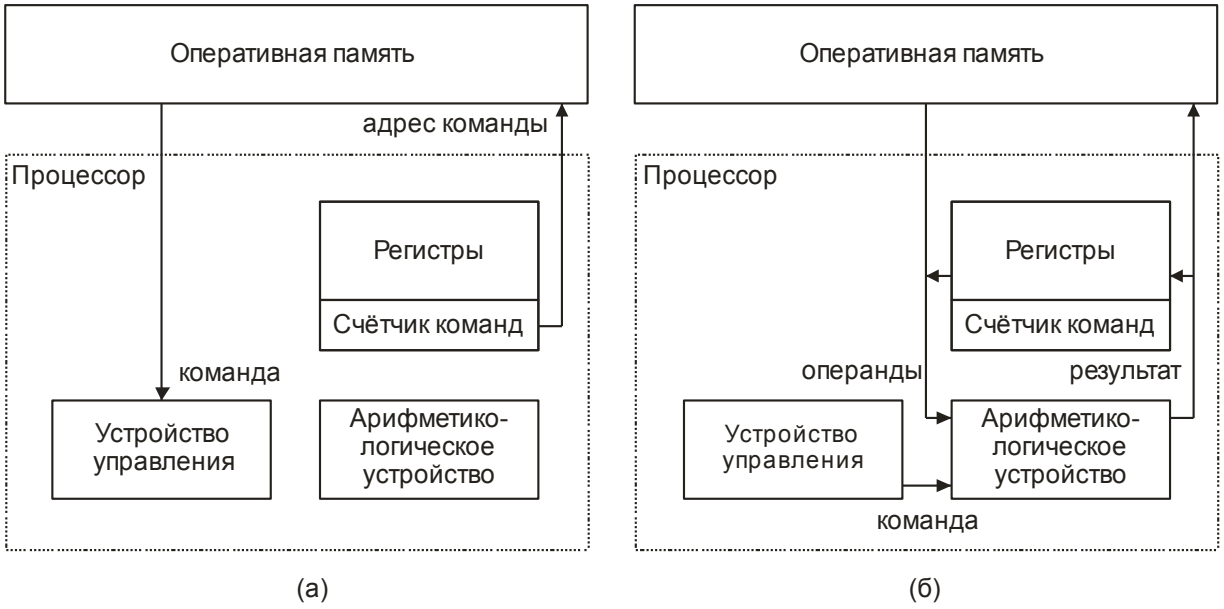

Рис. 3. Два шага функционирования процессора: выборка команды (а) и выполнение команды (б)

На первом шаге УУ выбирает команду из оперативной памяти, адрес которой указывает счетчик команд. После выборки команды значение счетчика команд изменяется. На втором шаге УУ декодирует команду, выбирает операнды и обеспечивает ее выполнение в АЛУ, организуя также запись в оперативную

память результатов. Принцип выполнения команды процессором на основе ее битового представления показан на Рис. 4.

Таким образом, характерная черта компьютера фон Неймана – это наличие глобально адресуемой памяти и счетчика команд, которые позволяют устройству управления многократно повторять один и тот же цикл действий: извлечение очередной команды и ее выполнение в автоматическом режиме. Именно поэтому такие компьютеры носят название *машины потока команд*.

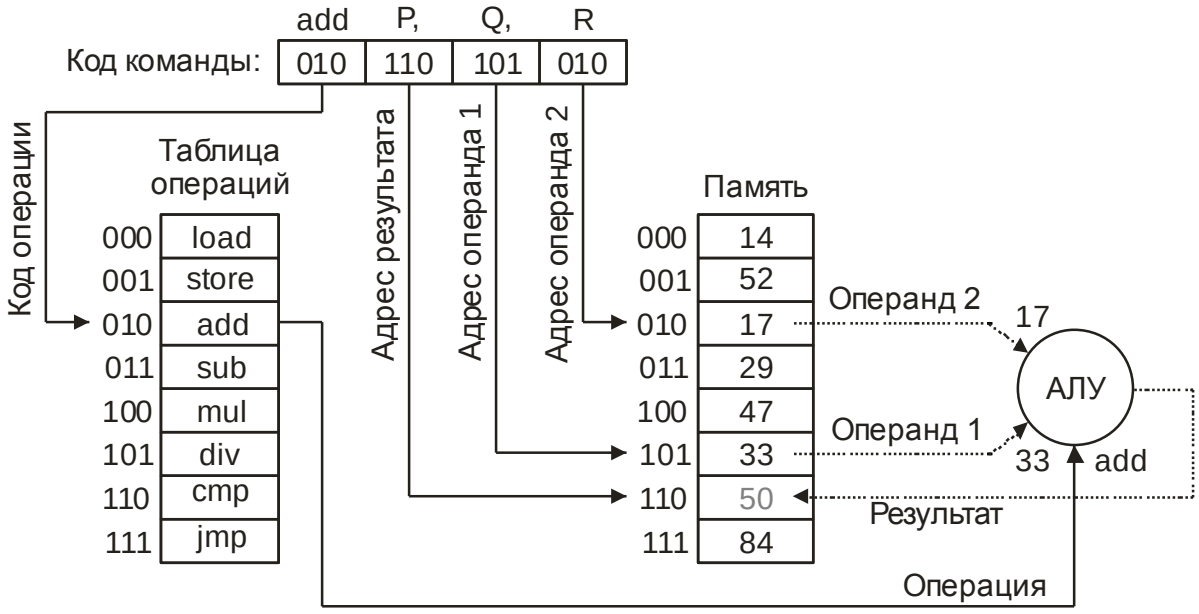

Рис. 4. Принцип выполнения команды на основе ее битового представления

Идея хранить программу и данные в одной памяти представляла собой изящное инженерное решение, соответствующее имевшемуся на тот момент уровню технологий. Однако совместное хранение данных и программы и последовательное выполнение команд привело к так называемому «бутылочному горлу»: доступ к командам и данным осуществляется через один и тот же канал. Пропускная способность этого канала ограничивает скорость работы компьютера. Несмотря на огромный прогресс в области СБИС-технологий и программного обеспечения, производительность процессоров в настоящее время подошла к теоретическому пределу. Исходные положения архитектуры компьютера фон Неймана, с одной стороны, привели к усложнению компьютеров, а с другой

стороны, они препятствовали увеличению скорости выполнения программ. Поэтому постепенно наметилась тенденция отхода от классической реализации архитектуры фон Неймана с целью преодолеть ее ограничения. И если в XX веке вклад в повышение производительности процессоров был в большей степени схемотехнический, за счет повышения тактовой частоты процессора, то с 2000-х годов доля увеличения производительности за счет усовершенствования организации компьютера стала превалировать.

Улучшения микроархитектуры направлены на ускорение доступа к оперативной памяти и выполнения команд. Ускорение доступа к памяти достигается за счет введения аппаратной и программной предвыборки команд и данных, дополнительных уровней иерархии памяти, виртуальной памяти, большего регистрового файла, высокоскоростных шин и т.д. Ускорение выполнения команд основано на упрощении набора команд, конвейеризированном выполнении команд и введении истинного (пространственного) параллелизма выполнения программы (на уровне данных, команд и потоков команд).

1.4. ОСНОВНЫЕ КОМПОНЕНТЫ СОВРЕМЕННОГО КОМПЬЮТЕРА

Современные компьютеры, как настольные персональные, так и серверы, являются многоядерными (Рис. 5). Функционально каждое ядро вместе с оперативной памятью унаследовало принципиальные черты архитектуры фон Неймана и представляет собой пару АЛУ-УУ. С точки зрения организации со времен компьютера фон Неймана ядро претерпело существенные изменения. Например, увеличилось количество функциональных устройств, был введен различного рода параллелизм, появилось внеочередное выполнение команд, были добавлены различные вспомогательные устройства, оптимизирующие работу процессора. В многоядерном компьютере программы и данные хранятся в общей оперативной памяти. Все ядра могут выполнять свои программы одновременно (параллельно).

Ядра процессора, как правило, размещаются на одном кристалле. На том же кристалле располагаются общие для ядер контроллер памяти и шинный интерфейс. Контроллер памяти выступает «посредником» между процессором и памятью. Шинный интерфейс служит для подключения контроллеров периферийных устройств к процессору. Оперативная память расположена отдельно от процессора, на системной плате. Кроме того, на системной плате расположены контроллеры периферийных устройств (например, контроллер дисков или контроллер шины USB).

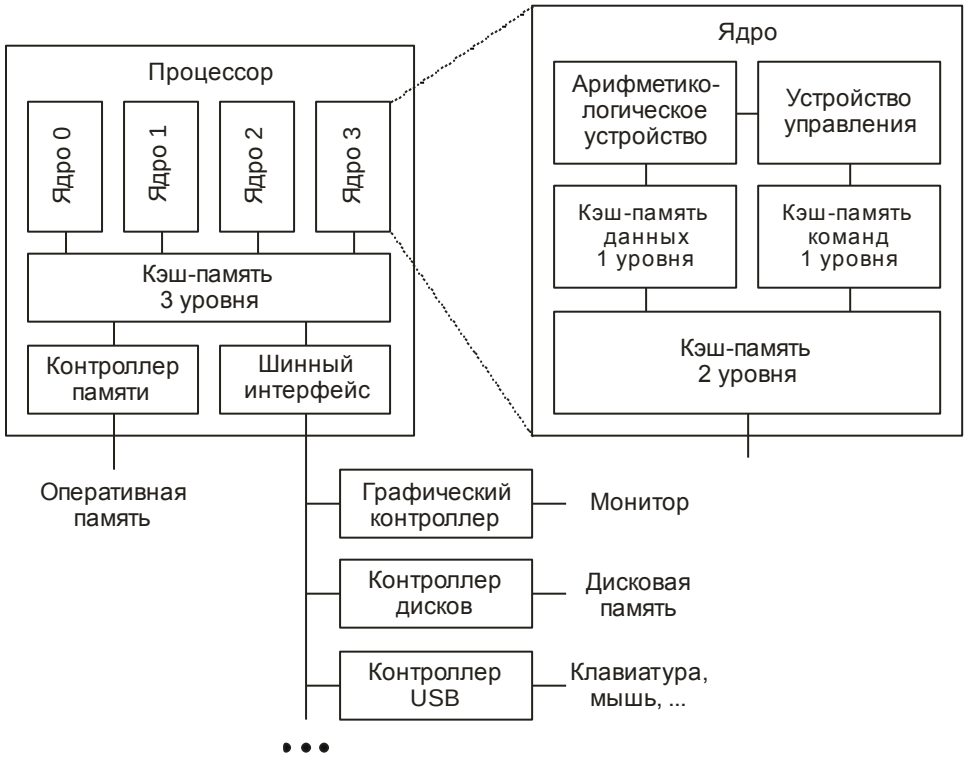

Рис. 5. Схема современного компьютера

Иерархия памяти в современном компьютере, по сравнению с иерархией памяти в архитектуре фон Неймана, стала более сложной. В нее добавлены несколько уровней кэш-памяти. В большинстве современных микропроцессоров каждое ядро имеет собственную кэш-память первого и второго уровней, а кэш-память третьего уровня является общей для нескольких ядер. Кроме того, кэш-память верхних уровней разделилась на кэш-память данных и команд.

1.5. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. На каких принципах базируется архитектура фон Неймана?
2. Какие принципы фон Неймана ограничивают производительность современных компьютеров?
3. Каким образом происходит выполнение команд в компьютерах с архитектурой фон Неймана?
4. В чем различие архитектуры и организации компьютера? Какова взаимосвязь между ними?
5. Почему одну архитектуру многократно реализуют в различных микроархитектурах?
6. Почему при реализации новых микроархитектур стараются сохранить старую архитектуру?

2. ОРГАНИЗАЦИЯ ПОДСИСТЕМЫ ПАМЯТИ

2.1. ОСНОВНОЙ ПРИНЦИП ПОСТРОЕНИЯ ИЕРАРХИЧЕСКОЙ ПАМЯТИ

Оперативная память предназначена для записи и хранения команд и данных и организации доступа к ним процессора. В идеале память должна иметь большой объем и обеспечивать процессор командами и данными таким образом, чтобы сократить до минимума простои процессора. Последние, как известно, возникают вследствие того, что время доступа в оперативную память превосходит время преобразования данных в процессоре. Этот разрыв имеет тенденцию к увеличению при возрастании степени интеграции (количества элементов в единице объема) и быстродействия элементной базы.

Чтобы уменьшить время доступа к памяти и максимально увеличить ее объем, сохраняя при этом высокую надежность за приемлемую цену, был введен принцип *иерархической* организации памяти [20]. Общая идея организации *иерархической* (*многоуровневой*) памяти заключается в использовании на одном компьютере различных типов запоминающих устройств (ЗУ). Каждому типу ЗУ в

зависимости от его характеристик (*времени доступа*, *объема* и стоимости) назначается определенный уровень в иерархии памяти. Порядок расположения уровней в иерархической памяти должен соответствовать следующим требованиям. С увеличением уровня иерархии должно происходить:

- уменьшение стоимости хранения единицы данных на данном уровне;
- увеличение объема памяти данного уровня;
- увеличение времени доступа;
- уменьшение частоты обращений к уровню со стороны процессора.

Первые три требования легко выполняются в рамках существующих технологических решений. Четвертое требование, как правило, тоже выполняется, поскольку является следствием **принципа локальности ссылок** (обращений к памяти). Этот принцип гласит, что большинство программ, выполняемых процессором, обладает свойствами локальности ссылок во времени и пространстве.

Локальность ссылок во времени заключается в том, что процессор после первого обращения к некоторой ячейке оперативной памяти с большой вероятностью обратится к ней снова в течение некоторого короткого промежутка времени. Этот промежуток времени будем называть локальным временным интервалом.

На Рис. 6 показана динамика во времени обращения процессора к некоторой ячейке оперативной памяти, в которой может быть записана команда или переменная. Здесь одна клетка соответствует шагу дискретного времени. Размер локального временного интервала взят равным трем шагам. Первое обращение к ячейке оперативной памяти обозначено черным цветом, последующие обращения к той же ячейке в пределах локального интервала предыдущих обращений – серым цветом.

Первый пример (Рис. 6а) демонстрирует ярко выраженную локальность ссылок во времени, так как в нем высока доля повторных обращений внутри временного интервала. Второй пример (Рис. 6б) показывает нарушение свойства локальности: повторное обращение есть только в четвертом интервале.

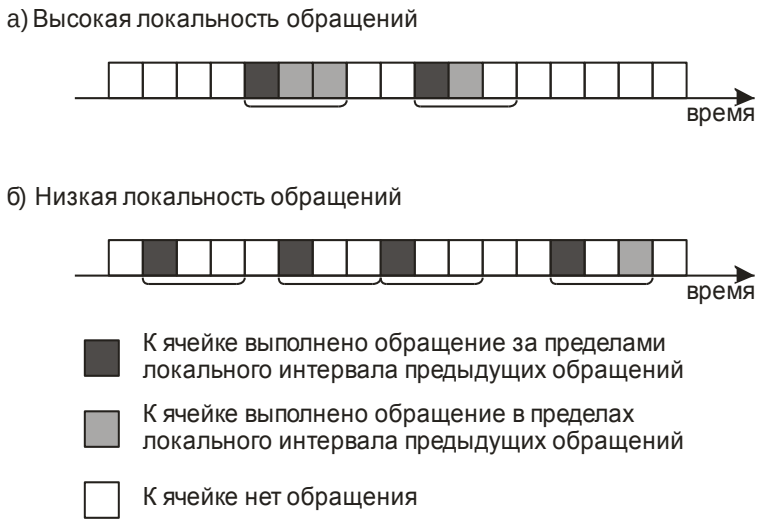

Рис. 6. Динамика обращений к одной ячейке памяти: динамика обращений с высокой локальностью (а) и динамика обращений с низкой локальностью (б)

Локальность ссылок в пространстве заключается в том, что процессор после обращения к некоторой ячейке оперативной памяти с большой вероятностью обращается к ее соседним ячейкам. Множество этих ячеек будем называть ее локальной окрестностью. Действительно, практически все команды обычной программы считываются из последовательных участков памяти. Обрабатываемые данные, как правило, структурированы, и обычно хранятся в последовательно расположенных ячейках оперативной памяти. Поэтому обращение к некоторой области памяти часто означает, что находящиеся в соседних ячейках данные вскоре могут понадобиться для выполнения следующих команд программы.

На Рис. 7 показана пространственная конфигурация обращений к ячейкам оперативной памяти в результате выполнения некоторой программы за некоторый короткий промежуток времени. Здесь одна клетка соответствует ячейке памяти. В качестве локальной окрестности некоторой ячейки взяты три ячейки оперативной памяти: она сама и два ее ближайших соседа. После выполнения программы каждая ячейка памяти находится в одном из четырех состояний, отражающих степень локальности обращений. Каждое состояние имеет свой цвет.

- Белый цвет указывает на ячейку памяти, к которой не было обращений.

- Светло-серый обозначает ячейку памяти, к которой было обращение, а к прочим ячейкам памяти из ее окрестности обращений не было.
- Темно-серый цвет соответствует ячейке памяти, к которой было обращение, и в ее окрестности имеется еще одна ячейка памяти, к которой было обращение.
- Черный цвет обозначает ячейку памяти, к которой было обращение, и в ее окрестности имеются еще две ячейки памяти, к которым было обращение.

Первый пример (Рис. 7а) демонстрирует ярко выраженную локальность ссылок в пространстве, так как в нем высока доля ячеек, в окрестность которых было обращение. Второй пример (Рис. 7б) показывает нарушение свойства локальности: окрестности ячейки, к которым есть обращения, практически не содержат других ячеек, к которым обращается процессор.

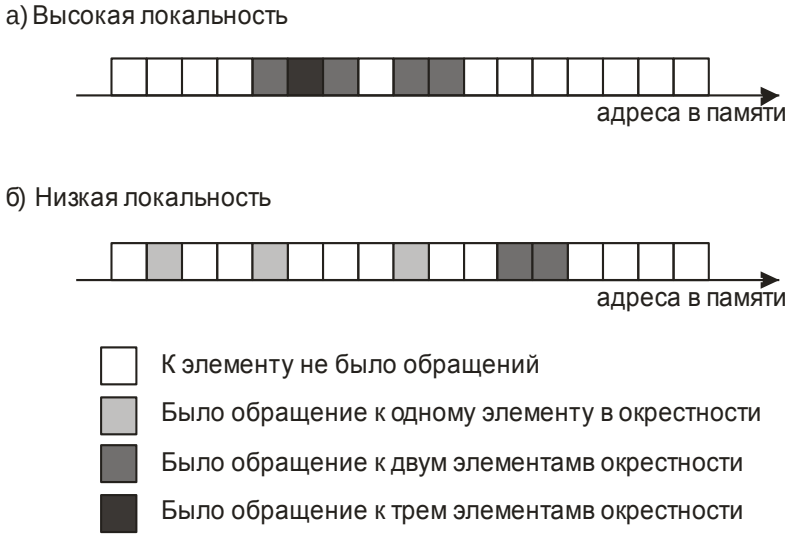

Рис. 7. Пространственная конфигурация обращений к ячейкам памяти: с высокой локальностью (а) и с низкой локальностью (б)

Принцип локальности ссылок – это основной принцип, на котором основано построение иерархической памяти. Благодаря этому принципу иерархически организованная память при выполнении большинства программ эффективно работает. Но не всегда иерархическая память работает эффективно, поскольку не всегда выполняется условие локальности обращений к данным (условие

локальности обращений к командам выполняется практически всегда). В этом случае локализация обрабатываемых данных может гарантированно уменьшить время выполнения программы.

2.2. ТИПИЧНАЯ СХЕМА ИЕРАРХИИ ПАМЯТИ В СОВРЕМЕННОМ КОМПЬЮТЕРЕ

Первая иерархическая память была организована в компьютере фон Неймана. Она имела два уровня иерархии: первый – быстрая оперативная память и второй – медленная внешняя память. Пересылка фрагментов программы между уровнями, в отличие от современных компьютеров, указывалась в программе явно программистом.

Иерархия памяти в современных компьютерах значительно усложнилась: число уровней возросло, передача данных стала преимущественно автоматической, стала выполняться на фоне вычислений. Типовая схема иерархии памяти для современных однопроцессорных компьютеров показана на Рис. 8. Каждый уровень памяти в иерархии характеризуется своим временем доступа, объемом и реализуется определенным типом памяти (быстрая и дорогая статическая память, медленная и дешевая динамическая память, внешняя память). Уровни памяти, наиболее тесно связанные с процессором, характеризуются меньшим временем доступа и меньшим объемом. И наоборот, уровни памяти, более удаленные от процессора, имеют большее время доступа и больший объем. В иерархии фрагменты программ и данных могут «двигаться» между уровнями памяти.

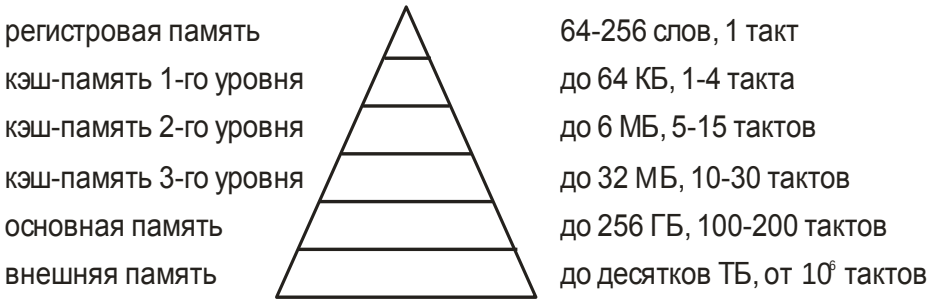

Рис. 8. Иерархическая схема организации памяти

Верхний уровень иерархии занимают *регистры* (регистровый файл). Регистры имеют самое малое время доступа из всех уровней иерархии памяти, поэтому их использование для хранения данных наиболее предпочтительно. В отличие от первых компьютеров, ядро современного микропроцессора содержит большее число аппаратных регистров, чем их предусмотрено в архитектуре. В результате для независимых команд, использующих один и тот же программный регистр, процессор может использовать разные ячейки регистровой памяти, устраняя зависимость по ресурсам. Это имеет большое значение, например, для параллельного выполнения команд.

В современных архитектурах обычно различают несколько групп регистров в зависимости от типа хранимых данных. Например, различают целочисленные, вещественные, векторные регистры. Это различение имеет прямую связь с микроархитектурой, т.к. соответствующие группы регистров должны физически располагаться недалеко от соответствующих функциональных устройств, работающих с ними.

Основными характеристиками регистров являются:

- размер регистра (число битов);
- количество регистров данного типа (в программной и аппаратной архитектуре);
- тип хранимых значений (целочисленные, вещественные, векторные, флаговые, адресные, …);
- набор команд процессора, допускающих использование данного регистра в качестве параметра.

Следующий уровень иерархии занимает *кэш-память*. Она хранит копии команд и данных из оперативной памяти. Для программиста кэш-память прозрачна: она не входит в адресное пространство, ее содержимое не может быть непосредственно прочитано или изменено. Содержимым кэш-памяти управляет специальное устройство – кэш-контроллер. Современные компьютеры имеют два

или три уровня кэш-памяти. Кэш-память может располагаться на кристалле процессора или вне кристалла. В современных процессорах кэш-память всех уровней, как правило, расположена на кристалле процессора, что обеспечивает наиболее быстрый доступ. Кроме того, большинство процессоров используют раздельную кэш-память 1-го уровня для данных и команд, каждая из которых имеет свои реализационные особенности.

За кэш-памятью последнего уровня в иерархии следует оперативное запоминающее устройство. Оперативная память в отличие от кэш-памяти является адресуемой. Она хранит данные и программу, которая выполняется на компьютере в текущий момент. Физически оперативная память обычно располагается отдельно от процессора на системной плате.

Нижний уровень иерархии памяти представлен внешней памятью. Это самая медленная и, обычно, самая большая по объему память. Она используется для организации виртуальной памяти. Кроме того, в отличие от предыдущих уровней иерархии, это энергонезависимая память, т.е. её состояние сохраняется и при выключении питания компьютера. Внешняя память – это, как правило, жесткий или твердотельный диск.

В современном компьютере существует два механизма управления иерархической памятью, которые задают правила пересылок фрагментов кода программы и данных между уровнями памяти. Первый механизм – это механизм кэширования, который связывает все уровни кэш-памяти и оперативную память. Кэширование реализуется целиком аппаратно кэш-контроллерами.

Другой механизм современных компьютеров, связанный с иерархической памятью – это механизм виртуальной памяти. Этот механизм реализует иллюзию большего объема памяти программы, он связывает оперативную память и внешнюю память. Виртуальная память реализуется совместно процессором и операционной системой.

2.3. ОРГАНИЗАЦИЯ КЭШ-ПАМЯТИ

Кэш-память состоит из блоков фиксированного размера, называемых *кэш-строками*. Каждая кэш-строка имеет индивидуальный номер (*индекс* или *слотовый номер*), строка хранит один блок кода или данных из оперативной памяти. Для отображения адресов оперативной памяти в адреса кэш-памяти (Рис. 9), оперативная память, состоящая из 2^n адресуемых слов, разбивается на блоки размером равным размеру кэш-строки. Между блоками оперативной памяти и кэш-строками устанавливается соответствие. Поскольку строк в кэш-памяти меньше, чем блоков в оперативной памяти, то индекс строки не может однозначно принадлежать одному блоку оперативной памяти. Поэтому каждая строка кэш-памяти, кроме блока хранит старшую часть его адрес в оперативной памяти, называемую *тэгом*. Тэг идентифицирует блок оперативной памяти, который записан в строку кэш-памяти в данный момент времени. Таким образом, адрес ячейки памяти состоит из трех компонентов: тэг, номер слота и смещение, которое указывает номер байта в блоке.

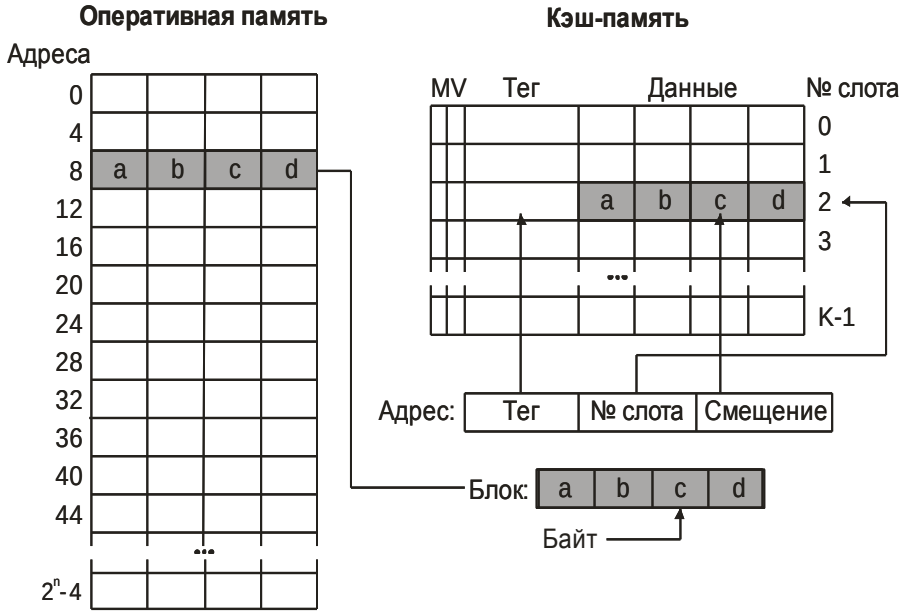

Рис. 9. Отображение блоков основной памяти в кэш-память

Каждая строка кэш-памяти содержит дополнительную информацию о кэшированном блоке. Типичный набор атрибутов содержит бит модификации и бит присутствия.

- Бит модификации указывает, модифицировался ли блок, лежащий в данной кэш-строке, и имеет два состояния: модифицированное и немодифицированное.
- Бит присутствия отражает присутствие блока в кэш-строке и характеризуется двумя состояниями: действительное и недействительное. Недействительное состояние означает, что данная кэш-строка не содержит копии какого-либо блока из оперативной памяти.

Работой кэш-памяти управляет *кэш-контроллер* (в современных микропроцессорах он интегрирован в процессор). Контроллер выполняет следующие функции.

- Загрузка (выгрузка) копии кода и данных из основной памяти в кэш-память выполняется блоками, равными размеру кэш-строки, даже если процессор обращается только к одной ячейке памяти из данного блока.
- Контроль запросов процессора к оперативной памяти и проверка, есть ли действительная копия информации в кэш-памяти. Если копия присутствует (эта ситуация называется *кэш-попадание*), то слово считывается из кэш-памяти и передается в процессор. Если действительная копия блока отсутствует в кэш-памяти (эта ситуация называется *кэш-промах*), тогда запрос адресуется к оперативной памяти и требуемая копия записывается на одну из строк кэш-памяти.
- Обеспечение *когерентности,* т.е. согласованности данных кэш-памятей с данными основной памяти.

Поскольку кэш-память имеет небольшой размер по сравнению с оперативной памятью, в ней должны храниться действительно нужные данные и коды программ. В противном случае трудно обеспечить быстрый доступ к наиболее часто используемым данным. Чтобы добиться этого, кэш-контроллер

использует различные стратегии помещения данных в кэш-память и поиска их в памяти.

2.3.1. Аппаратная и программная предвыборка

Предвыборка данных – это механизм уменьшения простоев процессора, которые связаны с ожиданием команд и данных. Этот механизм заключается в загрузке команд и данных в кэш-память из оперативной памяти до того, как они реально потребуются. В результате при первом обращении к соответствующей ячейке оперативной памяти не возникает кэш-промах, поскольку запрашиваемые данные и команды уже находятся в кэш-памяти. На Рис. 10 приведено сравнение операции чтения последовательно расположенных ячеек оперативной памяти для двух случаях – без предвыборки (Рис. 10а) и с предвыборкой (Рис. 10б). Предположим, что блоки памяти, содержащие эти ячейки, в кэш-памяти отсутствуют.

Рассмотрим случай последовательного доступа к ячейкам памяти без предвыборки (Рис. 10а). После того, как процессор не обнаружил запрашиваемую ячейку в кэш-памяти, инициируется загрузка блока оперативной памяти, содержащего данную ячейку, из памяти в строку кэш-памяти. Этот блок выбирается в течение некоторого времени. После загрузки блока происходит чтение ячейки из кэш-памяти и ее обработка. Далее процессор переходит к чтению следующей ячейки. Здесь возможны два варианта.

1. Ячейка расположена в том же блоке памяти, что и предыдущая ячейка. Тогда происходит ее загрузка из кэш-памяти и ее обработка.
2. Ячейка расположена в следующем блоке памяти. Следовательно, в кэш-памяти нет строки, в которую загружен этот блок, и тогда инициируется загрузка блока памяти, в котором данная ячейка содержится. Во время загрузки процессор простаивает. И только после загрузки отсутствующего блока памяти, ячейка считывается из кэш-памяти и обрабатывается.

Рассмотрим случай последовательного доступа к ячейкам памяти с предвыборкой (Рис. 10б). Загрузка следующего блока начинается не в момент чтения первой ячейки, а раньше. К моменту завершения обработки последнего элемента предыдущего блока очередной блок оперативной памяти уже загружен в строку кэш-памяти. Загрузка и обработка первого элемента очередного блока могут начинаться без задержки.

В зависимости от того, кто инициатор предвыборки – программа или кэш-контроллер, предвыборку разделяют на программную и аппаратную.

При *программной предвыборке* программист или компилятор явно вставляет в программу команды предвыборки данных по тому или иному адресу в оперативной памяти. Например, в некотором итерационном вычислительном процессе разумно делать предвыборку в кэш-память тех данных, которые понадобятся на следующих итерациях.

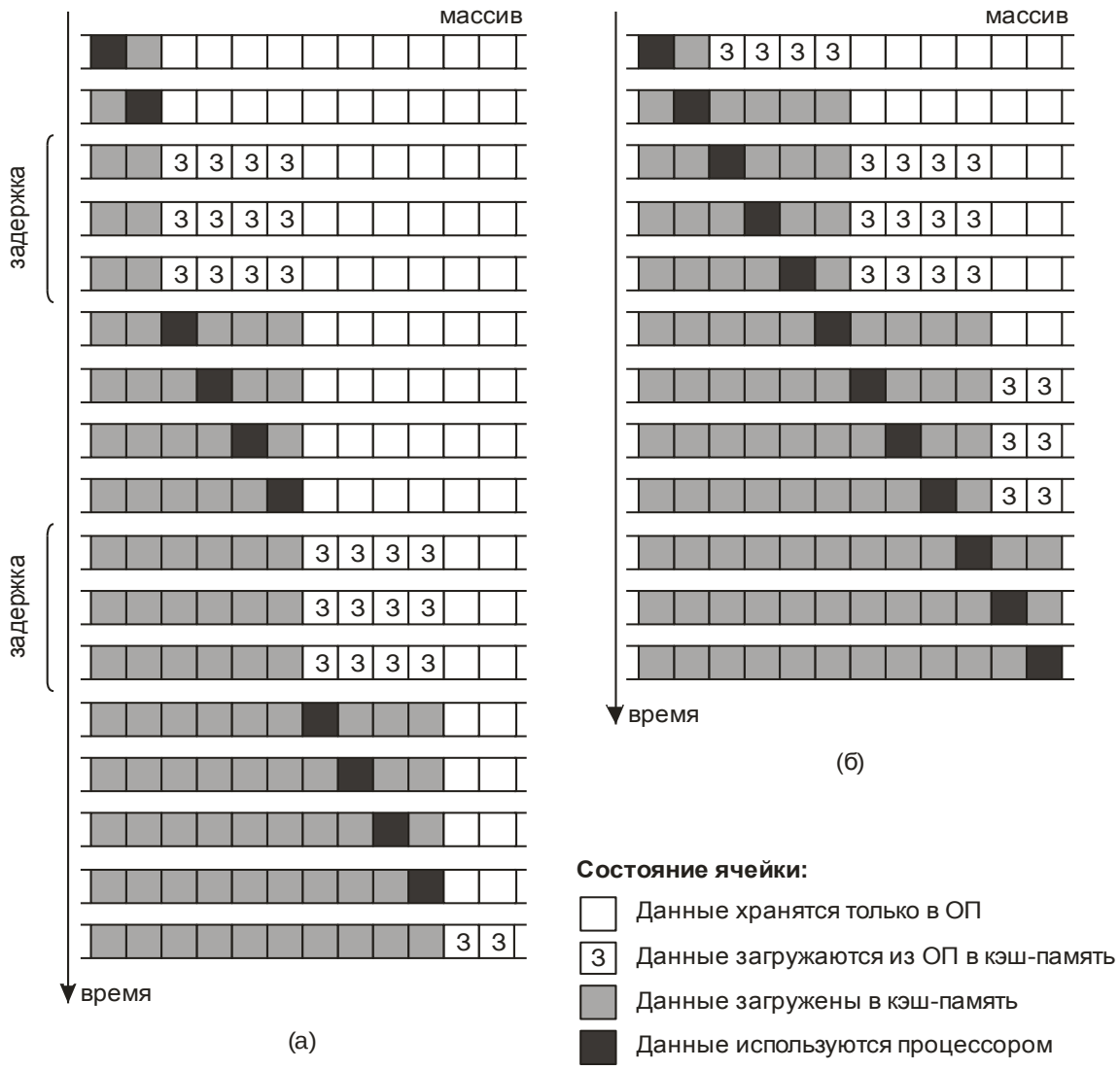

Рис. 10. Динамика последовательного доступа к ячейкам памяти: нет предвыборки – медленная обработка за счёт ожидания загрузки данных (а); есть предвыборка – быстрая обработка за счёт предварительной загрузки данных (б)

Аппаратная предвыборка происходит неявно, без участия человека или компилятора. Кэш-контроллер анализирует, по каким адресам и в каком порядке программа обращается к оперативной памяти и пытается предугадать, какие данные вскоре могут понадобиться программе, и осуществляет их автоматическую предвыборку в кэш-память. Разумеется, по последовательности обращений программы в оперативную память в общем случае невозможно предсказать, какие данные понадобятся впоследствии. Однако, есть часто встречающиеся шаблоны обработки данных в памяти, например,

последовательный обход массива. Если кэш-контроллер обнаруживает, что оперативная память последовательно опрашивается с некоторым фиксированным шагом, то он делает предположение, что в программе имеет место некоторая последовательная обработка массива, и начинает загружать следующие блоки данных заранее, ещё до того, как к ним реально произойдёт обращение.

Аппаратная предвыборка данных в кэш-память не застрахована от ошибок. Если кэш-контроллер распознал в последовательности обращений обход массива ошибочно, то данные, которые он загрузит в кэш-память, могут и не понадобиться. Более того, они могут «вытеснить» какие-либо полезные данные, находившиеся в кэш-памяти, и их придётся загружать снова, когда программа к ним обратится снова. Тем не менее, в большинстве случаев аппаратная предвыборка данных в кэш-память сказывается на скорости работы программ положительно.

Аппаратная предвыборка способна распознавать несколько одновременных последовательных обходов ячеек памяти. Например, если рассмотреть программу покомпонентного сложения двух векторов в третий, то кэш-контроллер, скорее всего, распознает обходы и будет выполнять предвыборку данных из оперативной памяти сразу для всех трех векторов. Количество обходов, которые кэш-контроллер может распознавать одновременно, служит его аппаратной характеристикой и называется количеством потоков предвыборки. Обычно кэш-контроллер имеет несколько потоков предвыборки, способных распознавать обходы памяти в сторону увеличения адресов ячеек, и несколько потоков предвыборки – в сторону уменьшения адресов ячеек.

Не в каждой программе, в принципе, возможна предвыборка данных в кэш-память. Например, при обработке таких структур данных, как списки, в общем случае заранее неизвестно, данные по какому адресу в оперативной памяти понадобятся следующими, пока текущее значение не будет прочитано. В этом случае ни программная, ни аппаратная предвыборки не помогут ускорить выполнение программ.

2.3.2. Алгоритмы отображения адресов оперативной памяти в строки кэш-памяти

Существуют три схемы отображения адресов оперативной памяти в кэш-память: *прямая, ассоциативная и множественно-ассоциативная*. В зависимости от схемы отображения адресов, различают три типа кэш-памяти: кэш-память с прямым отображением, кэш-память с ассоциативным отображением и кэш-память с множественно-ассоциативным отображением.

Рассмотрим каждую из перечисленных схем отображения адресов на следующем примере. Пусть основная память составляет 16 МБ (16 МБ = 2^{24} байт), кэш-память имеет объем 64 КБ, размер кэш-строки составляет 4 байта.

В кэш-памяти **с прямым отображением** адрес ячейки данных в памяти состоит из трех частей: тэга, индекса и смещения внутри блока. Согласно длине блока (4 байта) основная память объединяет 2^{22} блока, а кэш-память содержит 2^{14} кэш-строк. Логика кэш-памяти с прямым отображением интерпретирует 22 разряда адреса блока как 8-битовый тег (старшие разряды адреса), 14-битовый индекс (средние разряды адреса) и 2-битовое смещение (младшие разряды адреса) (Рис. 11).

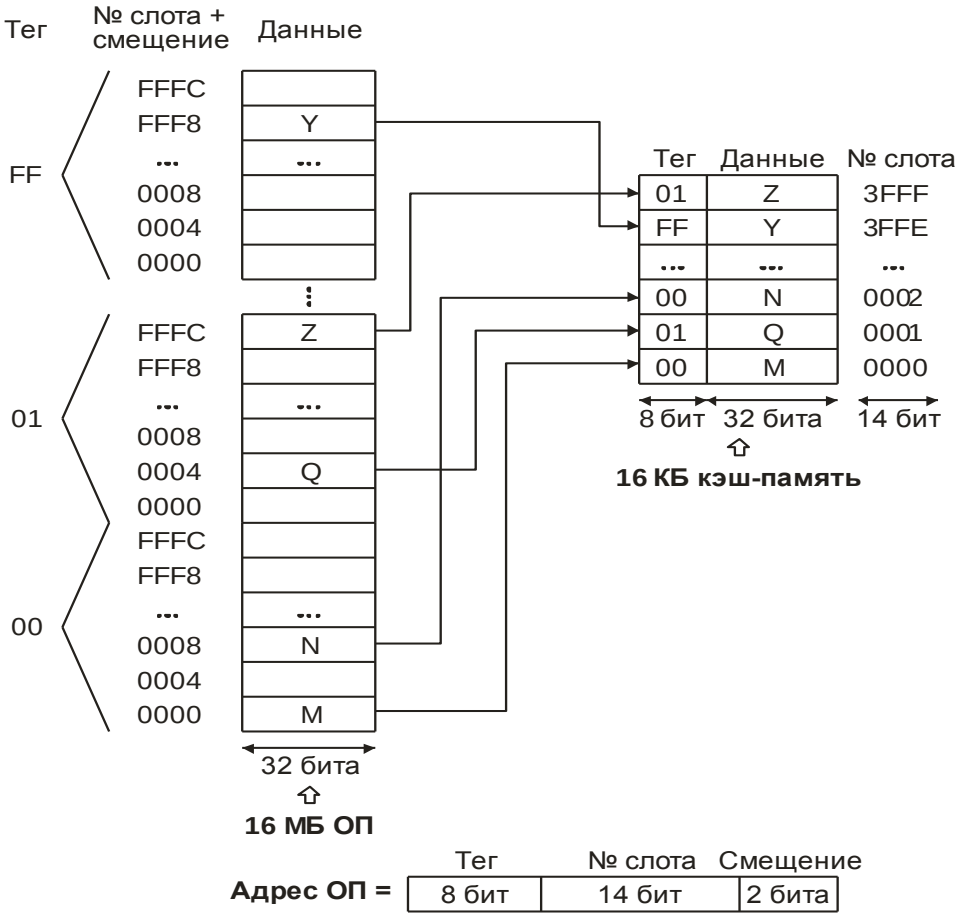

Рис. 11. Пример организации кэш-памяти с прямым отображением

Следующая формула устанавливает отображение адресов основной памяти и строк кэш-памяти:

$$S=A \bmod C, \qquad (1)$$

где S – адрес строки (индекса) кэш-памяти, A – адрес блока основной памяти, C – число строк в кэш-памяти. Так, блоки с номерами **00**0000, **01**0000, …, **FF**0000 основной памяти отображаются в строку кэш-памяти с номером 0, блоки с номерами **00**0004, **01**0004, …, **FF**0004 основной памяти отображаются в строку с номером 1 и так далее.

Кэш-память с прямым отображением имеет высокую скорость выполнения операций, простую и недорогую реализацию, с одной стороны; с другой стороны, видно, что данный алгоритм в одну и ту же строку кэш-памяти отображает несколько блоков оперативной памяти, в нашем случае 2^8 блоков. Это означает, что если в программе последовательно происходят обращения к элементам

памяти, отстоящим на величину, кратную размеру кэш-памяти, то все эти элементы будут претендовать на одну и ту же строку кэш-памяти. Несмотря на то, что в кэш-памяти в данный момент может быть сколько угодно пустых строк. Такая ситуация называется эффектом *буксования* кэш-памяти. Буксование является примером неэффективной работы кэш-памяти, поскольку для размещения каждого следующего элемента данных, участвующего в вычислении, необходимо вытеснить предыдущий элемент данных из кэш-памяти и обратиться в оперативную память. Устранив этот эффект, можно существенно уменьшить время работы программы.

В кэш-памяти с **ассоциативным отображением** любой блок памяти может быть записан в любую кэш-строку. Поэтому здесь эффект буксования кэш-памяти отсутствует. При ассоциативном отображении адрес блока представляется в виде двух компонент: тэга и смещения внутри блока (Рис. 12).

Для примера на Рис. 11 адрес состоит только из тега, занимающего 22 разряда, и смещения, занимающего два младших разряда. В отличие от прямого отображения, в адресе не задается номер слота. Нахождение нужной строки в кэш-памяти сводится к сравнению тега адреса блока затребованных данных с тэгами всех строк параллельно. Одновременное сравнение – это сложная аппаратная задача. Поэтому время доступа к ассоциативной кэш-памяти значительно, как правило, больше, чем время доступа к кэш-памяти с прямым отображением. При отсутствии свободной строки в кэш-памяти любая из ее строк может быть заменена на требуемую строку.

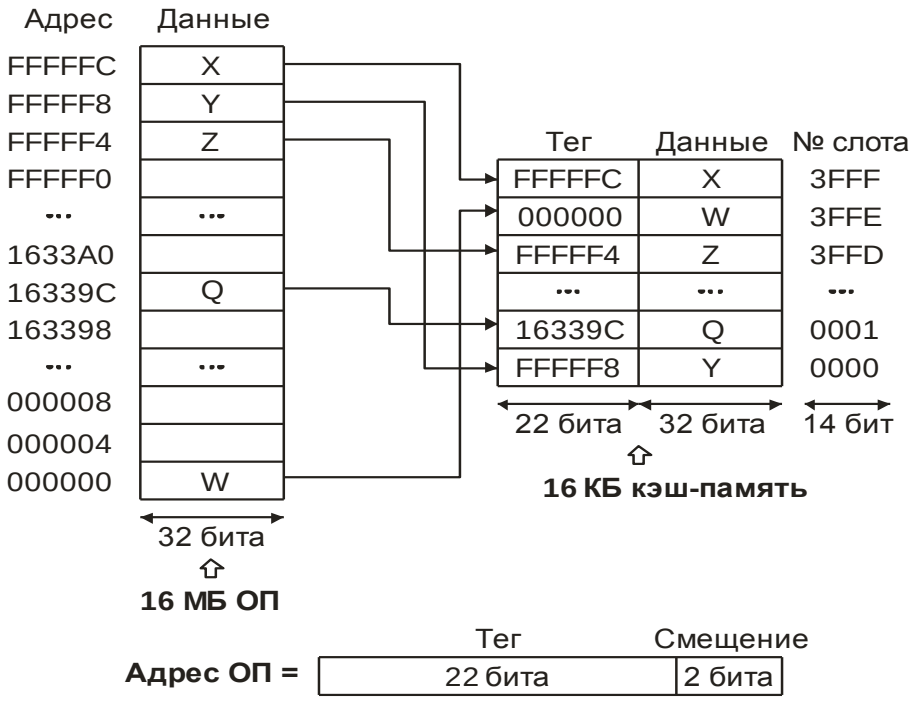

Рис. 12. Пример организации кэш-памяти с ассоциативным отображением

В кэш-памяти с **множественно-ассоциативным отображением** (или наборно-ассоциативным отображением) несколько блоков памяти с одинаковым индексом и разными тэгами могут быть записаны в одно множество (набор). Количество блоков в множестве называется *степенью ассоциативности.* На практике типичными размерами множества бывают 2, 4, 8, 16.

Множественно-ассоциативная кэш-память может быть представлена как несколько параллельно работающих кэш-памятей прямого отображения, которые принято называть *банками*. Число банков равно степени ассоциативности кэш-памяти. Для данной схемы отображения адресов, адрес состоит из трех компонент: тега, номера множества и смещения в блоке. Решение, в какой именно банк размещать блок данных, принимает кэш-контроллер. Выбор нужной кэш-строки в банке однозначно определяется номером множества, получаемом из адреса блока.

На 13 приведен пример множественно-ассоциативной кэш-памяти со степенью ассоциативности, равной двум. Поскольку количество кэш-строк в кэш-памяти равно 2^{14}, а каждое множество содержит по две кэш-строки (по числу

банков), то число множеств будет 2^{13}, соответственно, разрядность номера множества будет 13 бит. Разрядность смещения для адресации 4-х байт блока равна 2-м битам, и на тег остается 9 старших бит адреса.

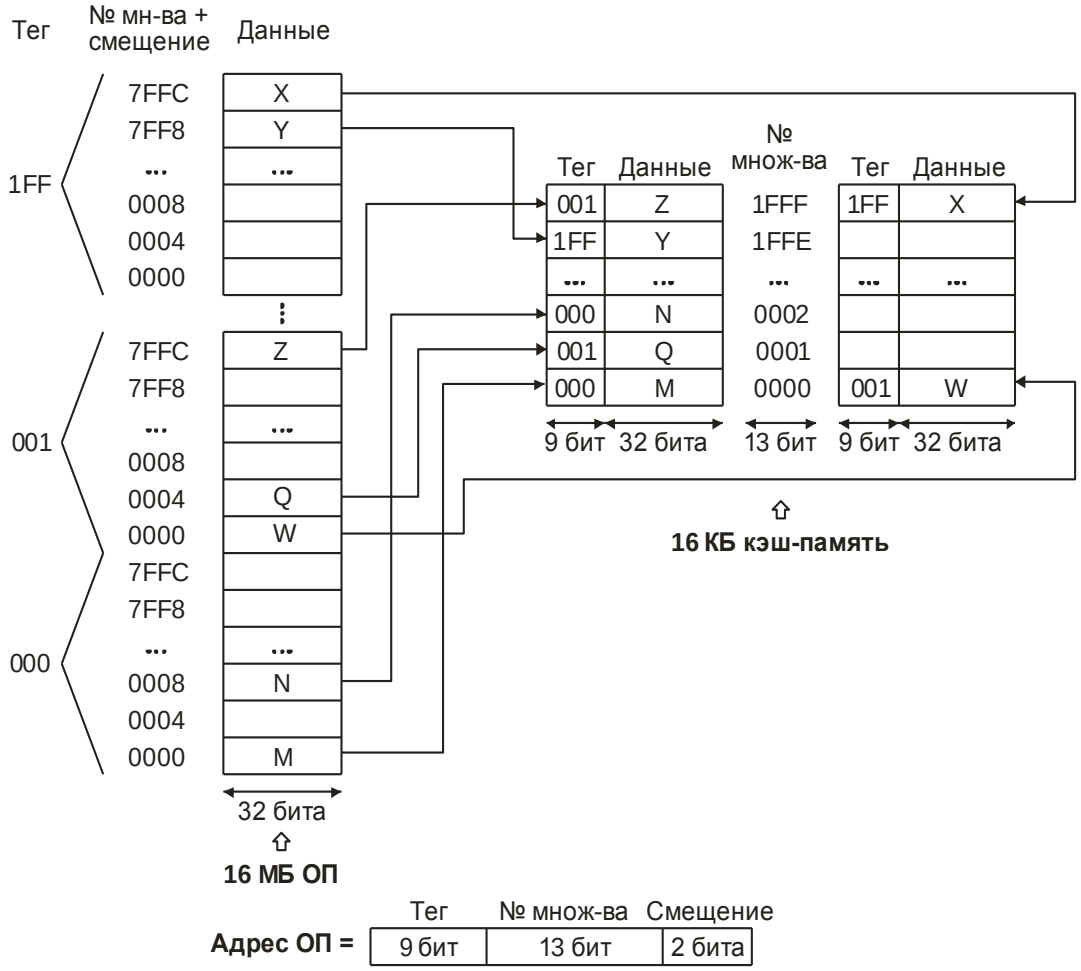

Рис. 13. Пример организации кэш-памяти с множественно-ассоциативным отображением

В множественно-ассоциативной кэш-памяти, как и в кэш-памяти прямого отображения, возможен эффект буксования. Но здесь он происходит гораздо реже, так как для него требуется уже не одно, а несколько обращений, претендующих на одно и то же множество кэш-строк. Причем число таких конфликтующих обращений должно быть больше степени ассоциативности кэш-памяти.

В табл. 1 приведены характеристики кэш-памятей некоторых современных микропроцессоров – объем, степень ассоциативности (ст.а.) и размер кэш-строки. Все представленные процессоры являются многоядерными. Видно, что большинство современных микропроцессоров имеют два-три уровня кэш-памяти, причем первые один или два уровня дублируются на каждом ядре и разделены на кэш-память команд и кэш-память данных, а кэш-память последних уровней является общей для всех ядер. Размер кэш-памяти увеличивается с увеличением уровня.

Таблица 1

Процессор	Кэш-память 1 уровня		Кэш-память 2 уровня		Кэш-память 3 уровня
	Данных	Команд	Данных	Команд	
Intel Atom N570, 2 ядра	24 КБ, ст.а.6, 64 Б	32 КБ, ст.а.8, 64 Б	512 КБ, ст.а.8, 64 Б		
Intel Xeon E7-8870, 10 ядер	16 КБ, ст.а.8, 64 Б	32 КБ, ст.а.4, 64 Б	256 КБ, ст.а.8, 64 Б		30 МБ, ст.а.24, 64 Б, общий для всех ядер
Intel Xeon E5-2687W, 8 ядер	32 КБ, ст.а.8, 64 Б	32 КБ, ст.а.8, 64 Б	256 КБ, ст.а.8, 64 Б		20 МБ, ст.а.20, 64 Б, общий для всех ядер
AMD Opteron 6386 SE, 16 ядер	16 КБ, ст.а.4, 64 Б	64 КБ, ст.а.2, 64 Б, общий для 2-х ядер	2 МБ, ст.а.16, 64 Б, общий для 2-х ядер		8 МБ, ст.а.64, 64 Б, общий для 8-ми ядер
Intel Itanium 9560, 8 ядер	16 КБ, ст.а.4, 64 Б	16 КБ, ст.а.4, 64 Б	256 КБ, ст.а.8, 128 Б	512 КБ, ст.а.8, 128 Б	32 МБ, ст.а.32, 128В, общий для всех ядер
IBM Power7+, 8 ядер	32 КБ, ст.а.8, 128 Б	32 КБ, ст.а.4, 128 Б	256 КБ, ст.а.8, 128 Б		80 МБ, ст.а.8, 128 Б, общий для всех ядер
Fujitsu SPARC64 VIIIfx, 8 ядер	32 КБ, ст.а.2, 128 Б	32 КБ, ст.а.2, 128 Б	6 МБ, ст.а.12, 128 Б, общий для всех ядер		
Intel Xeon Phi 5110P, 60 ядер	32 КБ, ст.а.8, 64 Б	32 КБ, ст.а.8, 64 Б	256 КБ, ст.а.8, 64 Б		

2.4. ПРИМЕРЫ РАБОТЫ С КЭШ-ПАМЯТЬЮ

Для лучшего понимания механизма кэширования данных рассмотрим порядок загрузки блоков оперативной памяти в строки кэш-памяти тремя способами:

1) обход элементов массива по строкам,

2) обход элементов массива по столбцам с буксованием кэш-памяти,

3) обход элементов массива по столбцам без буксования кэш-памяти

Особенности механизма кэширования покажем на кэш-памяти со следующими параметрами: размер 32 КБ, степень ассоциативности 8, размер кэш-строки 64 Б, число множеств 64 (Рис. 14). Такую организацию имеет кэш-память данных 1-го уровня в процессорах с микроархитектурой Intel Sandy Bridge.

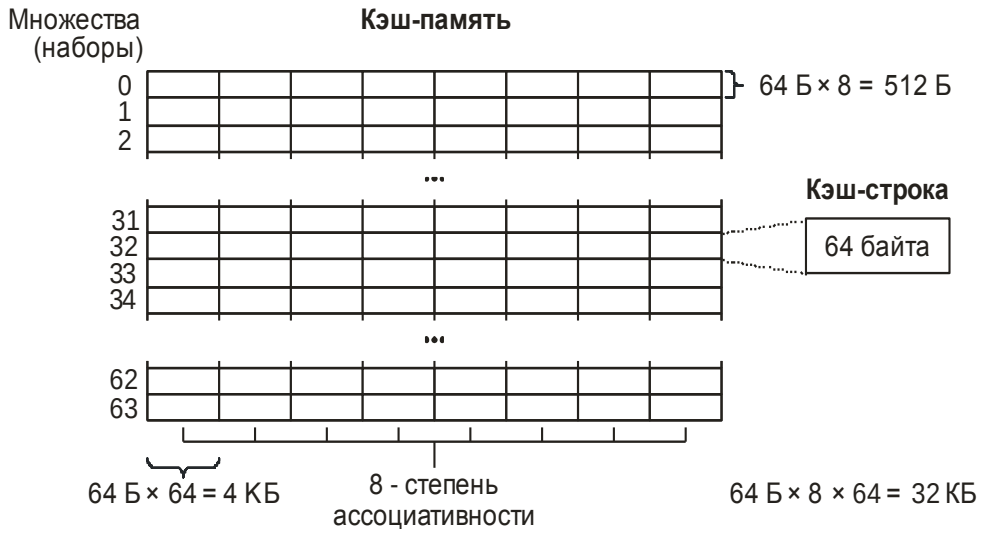

Рис. 14. Структура кэш-памяти данных 1-го уровня в процессорах с микроархитектурой Intel Sandy Bridge

2.4.1. Обработка матриц по строкам

Допустим, требуется обработать некоторую матрицу целочисленных 4-х байтовых значений (тип int), расположенную в памяти. Размер матрицы – 512×512 элементов. Каждая строка матрицы располагается в памяти непосредственно за предыдущей строкой (Рис. 15). При обходе матрицы построчно, начиная с

нулевой строки, она начинает загружаться в кэш-память. Каждая строка матрицы занимает 512 × 4 Б = 2 КБ памяти. Таким образом, для размещения одной строки матрицы потребуется 32 кэш-строки, т.к. она делится именно на такое количество 64-байтовых блоков. (Тут и далее считается, что матрица в памяти выровнена по границе 64 Б). Вслед за нулевой строкой матрицы в кэш-память будет помещена первая, которая займёт следующие 32 кэш-строки. Далее, по мере обхода матрицы, в кэш-память будут загружаться и следующие её строки. На Рис. 15 показан пример того, какое место в кэш-памяти могут занять строки матрицы. На этом рисунке каждый следующий 64-байтный блок помещён строго под предыдущим. Каждый следующий блок действительно попадёт в следующее множество в соответствии с алгоритмом отображения блоков в кэш-память. Однако выбор кэш-строки внутри этого множества может быть любым из 8 имеющихся, не обязательно тем же, что и для предыдущего блока. Строка внутри множества выбирается кэш-контроллером в зависимости от алгоритмов замещения, реализованных в нем, и его состояния на момент начала обхода массива.

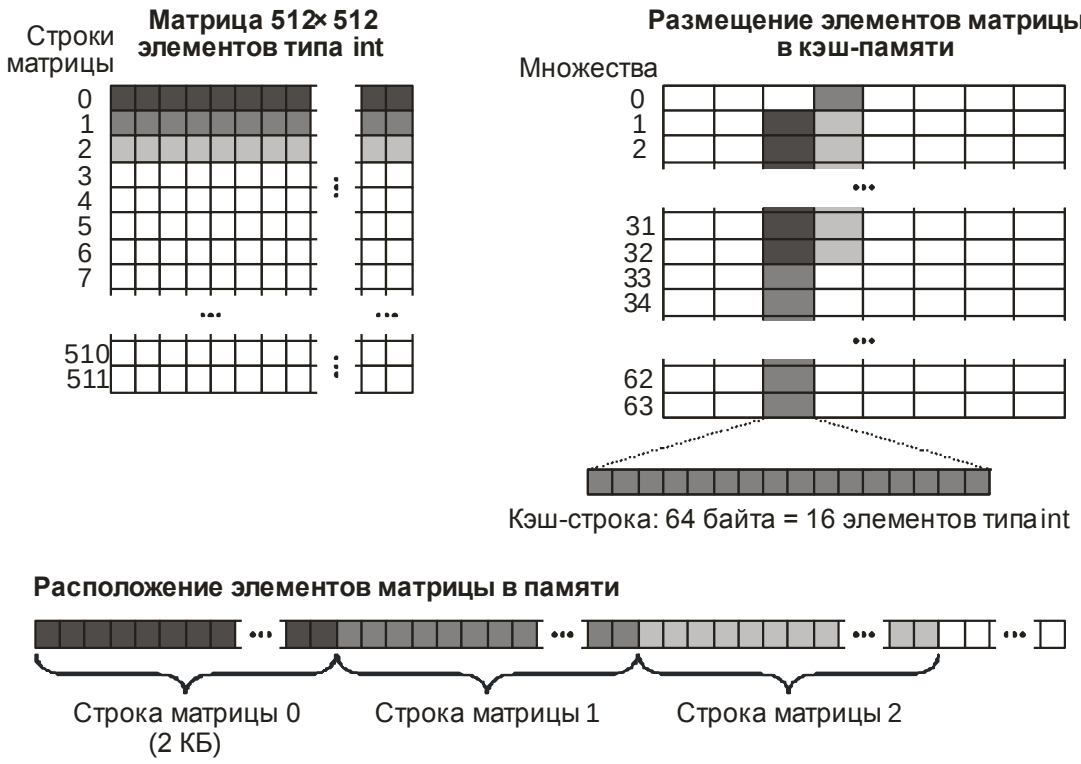

Рис. 15. Работа кэш-памяти при построчной обработке матрицы

Размер рассматриваемой матрицы в памяти равен 512 × 512 × 4 Б = 1 МБ, что превышает размер имеющейся кэш-памяти. Одновременно кэш-память может вместить лишь 16 строк матрицы. Это означает, что при обработке строки матрицы с номером 16 её загрузка в кэш-память повлечёт вытеснение оттуда уже загруженных ранее данных. Скорее всего, вытесняться будет строка матрицы с номером 0, т.к. это данные, которые не использовались дольше всего. В общем случае, какие кэш-строки будут вытесняться, определяется алгоритмом замещения. После того, как первые строки матрицы начинают вытесняться из кэш-памяти, обращение к ним опять приведёт к кэш-промаху и последующей их загрузке в кэш-память.

2.4.2. Обработка матриц по столбцам с буксованием

Теперь представим, что необходимо осуществить обработку той же матрицы, но по столбцам (Рис. 16). При обращении к элементу матрицы (0, 0) в кэш-память будет загружен 64-байтный блок, содержащий этот элемент. Т. е. при обращении к одному элементу будет загружено 64 Б / 4 Б = 16 элементов матрицы, расположенных в строке матрицы с номером 0. Тем не менее, 15 из этих элементов в настоящий момент не нужны, т.к. обработка матрицы осуществляется по столбцам. И хотя есть надежда, что позже, когда эти элементы понадобятся, то они ещё будут находиться в кэш-памяти, сейчас мы увидим, что это не так. Допустим, блок памяти был загружен в одну из кэш-строк во множестве с номером 1. Далее, при обращении к следующему элементу матрицы, в кэш-память будет загружен блок, его содержащий. В соответствии с алгоритмом отображения адресов в кэш-память можно вычислить номер множества, в который этот блок может быть загружен. Т.к. этот блок смещён относительно предыдущего в памяти на размер строки матрицы, т.е. 2 КБ или 32 блока, то номер множества будет равен (1 + 32) mod 64, т.е. 33. Тут 1 – это номер множества, в котором размещён предыдущий блок, а 64 – количество множеств в

кэш-памяти. Третий элемент будет снова загружен в множество с номером 1, и так далее. Таким образом, для хранения данных одного столбца матрицы реально из всей кэш-памяти будет использоваться лишь 2 множества суммарным объемом 2 × 8 × 64 Б = 1024 Б. Из этих 1024 Б лишь 4 Б × 16 = 64 Б заняты данными нулевого столбца, в то время как остальные 960 Б заняты элементами других столбцов матрицы. Получается, что лишь 16 элементов одного столбца могут быть одновременно загруженными в кэш-память. При обращении к следующим элементам столбца матрицы данные из кэш-памяти начнут вытесняться.

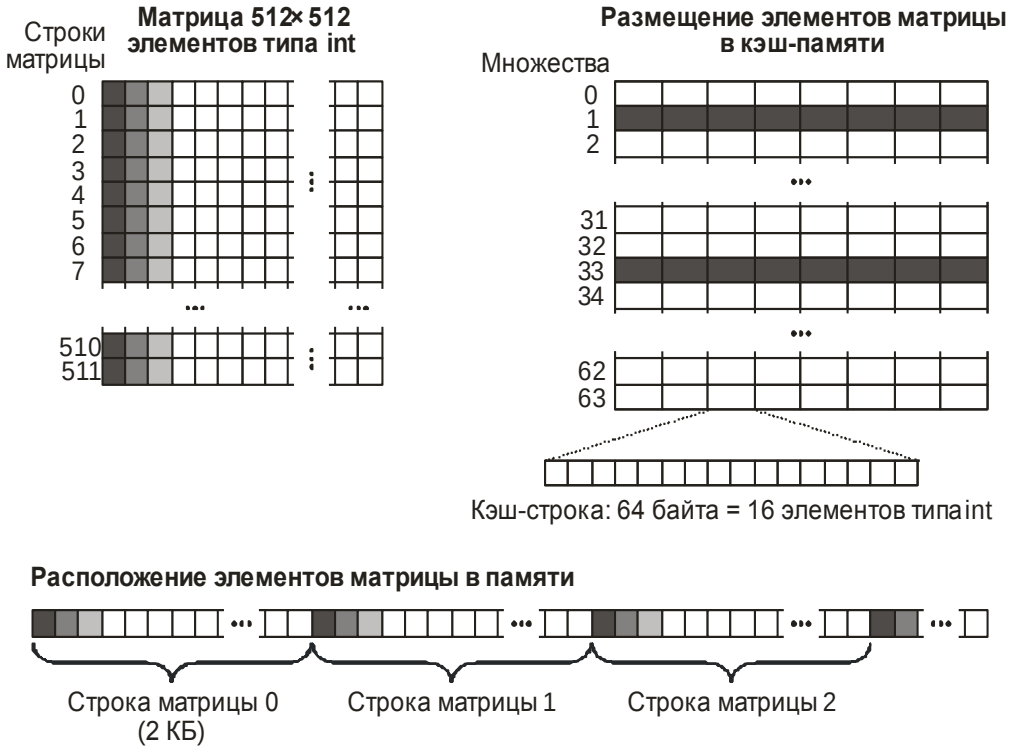

Рис. 16. Работа кэш-памяти при обработке матрицы по столбцам

Сравним количество кэш-промахов, которые возникают при обработке матрицы по строкам и по столбцам. Для простоты будем считать, что другие процессы отсутствуют и не влияют на состояние кэш-памяти. При построчной обработке матрицы при обращении к очередному её элементу в кэш-память будет загружен блок памяти размером 64 Б, содержащий, помимо текущего элемента 15 других. Обращение к этим 15 элементом последуют друг за другом

непосредственно после обработки текущего элемента. Как следствие, кэш-промах будет случаться лишь 1 раз на 16 элементов матрицы. При обработке матрицы по столбцам кэш-промах будет случаться при обращении к *каждому* её элементу. Таким образом, производительность подсистемы памяти будет такой, словно кэш-памяти отсутствует. Это пример ситуации, которая называется кэш-буксованием.

Кэш-буксование возникло из-за того, что смещение всех чётных и всех нечётных строк матрицы в памяти оказалось смещенными друг относительно друга на величину, равную 4 КБ. В соответствии с алгоритмом отображения адресов в кэш, это привело к тому, что элементы одного столбца должны загружаться в одно и то же множество, поэтому одновременная загрузка в кэш-память более чем 8 таких элементов невозможна.

2.4.3. Обработка матриц по столбцам без буксования

Одним из приёмов преодоления кэш-буксования является добавление в матрицу фиктивных, незначащих элементов так, чтобы элементы одного столбца не конкурировали за одно и то же множество в кэш-памяти (Рис. 17). В рассмотренном примере достаточно добавить 32 Б (т.е. 8 фиктивных элементов) к каждой строке массива. При этом следующие друг за другом элементы столбца в чётных строках претендуют на соседние множества в кэш-памяти. То же справедливо и для нечётных строк. В этом случае ценой небольшого перерасхода памяти мы уравниваем количество кэш-промахов с первым примером, т.е. построчным обходом матрицы.

2.5. ВИРТУАЛЬНАЯ ПАМЯТЬ

Виртуальная память – это механизм управления иерархической памятью компьютера, который позволяет размещать в памяти и одновременно выполнять несколько процессов. Виртуальная память предполагает, что пользователи имеют дело с кажущейся одноуровневой памятью, объем которой равен всему адресному пространству независимо от объема физической памяти компьютера и объема

памяти, необходимой для других программ, участвующих в мультипрограммной обработке.

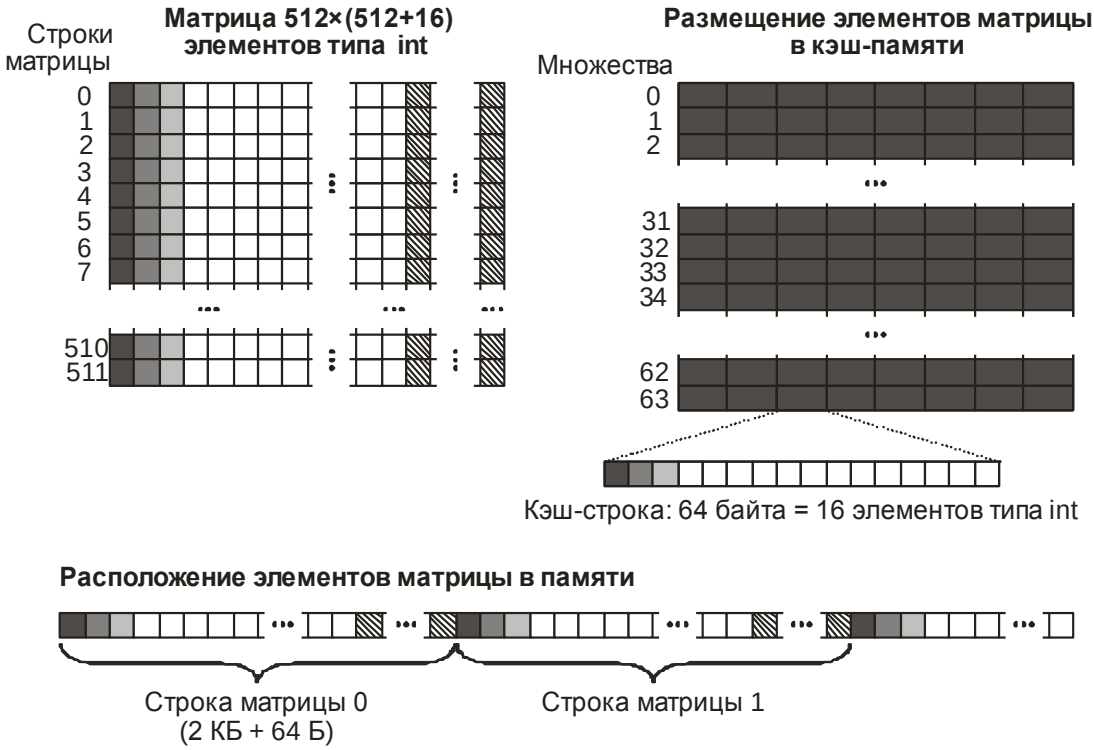

Рис. 17. Устранение эффекта кэш-буксования с помощью добавления в матрицу фиктивных элементов

Физически виртуальная память представляет собой совокупность всех ячеек памяти – оперативной и внешней (наличие ВЗУ обязательно). Она имеет сквозную нумерацию от нуля до предельного значения адреса. Адреса виртуального пространства называются *виртуальными*, адреса физического пространства (оперативной памяти) называются *физическими.*

В настоящее время наиболее распространенным способом организации виртуальной памяти является страничный способ. Виртуальная память делится на блоки фиксированного размера – виртуальные страницы. Физическая память также делится на блоки фиксированного размера – физические страницы. Размеры виртуальных и физических страниц совпадают. Физические страницы используются для хранения виртуальных страниц.

Адрес виртуальной (физической) страницы состоит из номера страницы и смещения (адреса относительно начала страницы). Страницы не имеют прямой связи с логической структурой данных и программ. Страничная организация памяти представляет память как набор страниц равного размера. В любой момент только часть страниц виртуальной памяти присутствует в оперативной памяти, т.е. та часть, которая необходима активным задачам (Рис. 18). Страничная организация сокращает объем пересылок данных между оперативной памятью и внешней памятью, поскольку

- загрузка и выгрузка страниц в оперативную память выполняется по мере необходимости (загрузка по требованию),
- отсутствует фрагментация, так как страница имеет фиксированный размер.

Кроме того, страничная организация памяти позволяет увеличить число одновременно выполняемых программ.

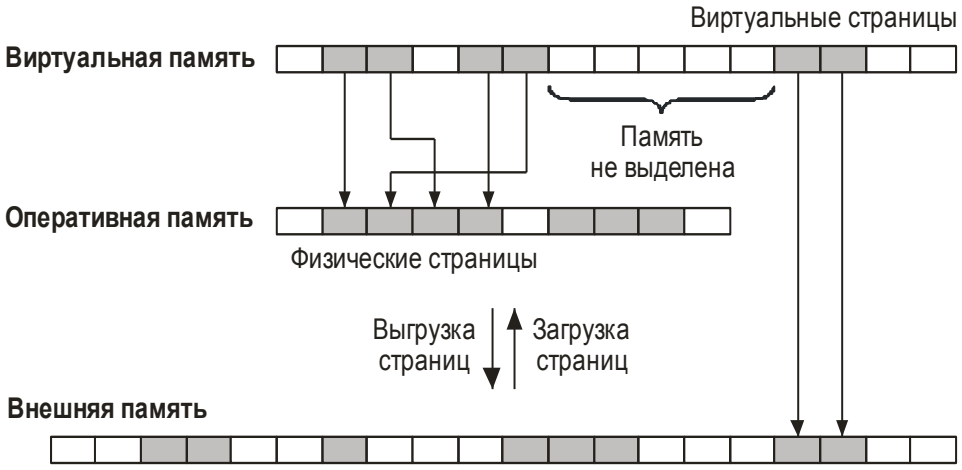

Рис. 18. Страничная организация виртуальной памяти

При использовании виртуальной памяти для каждой запущенной программы ОС создает собственное виртуальное адресное пространство (Рис. 19). Виртуальное адресное пространство описывается двумя таблицами: таблицей страниц и картой диска. Таблица страниц устанавливает соответствие виртуальных и физических адресов страниц. Карта диска содержит информацию

о расположении страниц во внешней памяти. Процесс доступа к данным по их виртуальным адресам выполняется следующим образом.

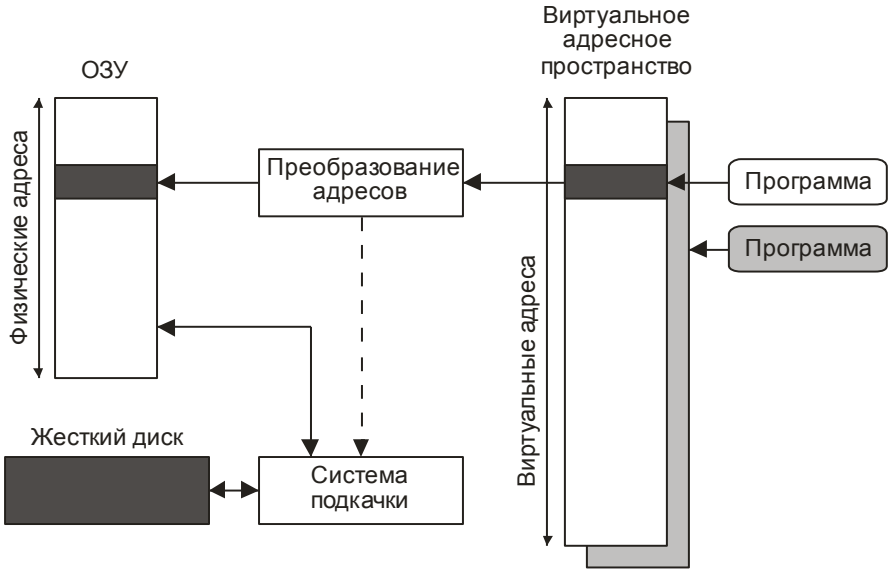

Рис. 19. Отображение виртуального адресного пространства на физическое

Процессор выставляет на шину адреса виртуальный адрес. Виртуальный адрес представлен парой чисел (p,s), где p – номер виртуальной страницы, а s – смещение внутри страницы (Рис. 20). Физический адрес аналогично представлен парой чисел (n,s), где n – номер физической страницы, а s – смещение внутри страницы. Виртуальный адрес преобразуется в физический адрес устройством управления памяти. Это устройство может находиться на микросхеме процессора или на отдельной микросхеме рядом с процессором. Преобразование осуществляется с помощью таблицы страниц по следующей схеме.

1. Если физическая страница находится в оперативной памяти, то в таблице страниц считывается строка p. Она содержит номер физической страницы n, по которому однозначно определяется физический адрес этой страницы. Искомый физический адрес вычисляется суммированием физического адреса физической страницы n и смещения s. Кроме физического адреса страницы, в строке таблицы может храниться информация о том, выделена ли оперативная память для данной страницы, происходила ли запись в

страницу после ее подгрузки, разрешено ли чтение или запись в эту страницу.

2. Если страница расположена во внешней памяти, то ее нужно подгрузить в свободную страницу оперативной памяти. Если свободной страницы нет, то по любому алгоритму вытеснения выбирается и освобождается одна из занятых страниц. Данные из этой страницы предварительно выгружаются во внешнюю память.

В современных процессорах для ускорения доступа к таблице преобразования адресов используются:

- многоуровневые таблицы страниц,
- буферы быстрого преобразования адресов (TLB).

TLB представляет собой полностью ассоциативную кэш-память или множественно-ассоциативную кэш-память с высокой степенью ассоциативности и временем доступа, сравнимым с кэш-памятью 1-го уровня. В ее памяти тэгов хранятся номера виртуальных страниц, а в памяти данных – номера физических страниц для нескольких последних операций трансляции адресов. Каждая строка таблицы содержит несколько признаков (достоверности, модификации, права доступа и т.д.)

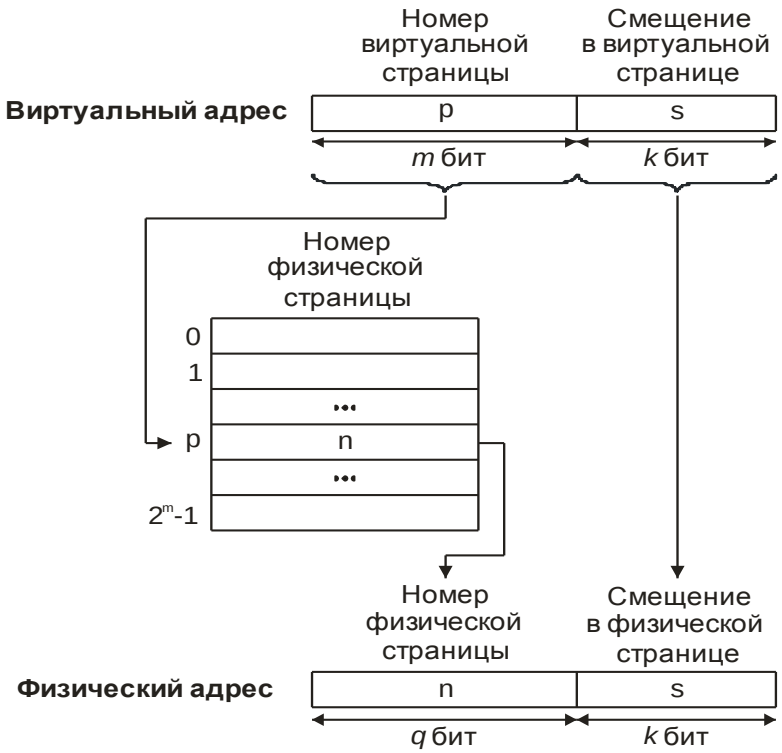

Рис. 20. Преобразование адресов в страничной виртуальной памяти

Виртуальная память и кэш-память имеют много сходств. Во-первых, главная функция, которую они реализуют, заключается в построении гибридной памяти, состоящей из нескольких уровней в иерархии памяти, которая кажется такой же быстрой, как память верхнего уровня и такой же большой, как память нижнего уровня. Как и в случае с кэш-памятью, эффективность использования виртуальной памяти в наибольшей степени определяется соблюдением свойства временной и пространственной локальностей доступа к данным. В случае ее эффективного использования она имеет быстродействие почти как у оперативной памяти. В противном случае ее быстродействие замедляется до уровня внешней памяти.

2.6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие проблемы решает иерархическая организация памяти?
2. Чем организация памяти в современном компьютере отличается от организации памяти в архитектуре фон Неймана?

3. Какие существуют соотношения между временем доступа, объемом и стоимостью памяти?

4. Всегда ли доступ к памяти с соблюдением принципа локальности осуществляется быстрее, чем когда этот принцип не соблюдается?

5. Каким образом аппаратная предвыборка уменьшает время выполнения программы?

6. Какие преимущества и недостатки имеет множественно-ассоциативная кэш-память по сравнению с другими видами кэш-памяти?

7. В чем различия между виртуальным и физическим адресом?

8. В каких случаях происходит загрузка и выгрузка страниц виртуальный памяти?

9. Ускоряет или замедляет работу компьютера виртуальная память? Почему?

10. В чем сходства и различия между механизмами кэширования и виртуальной памяти?

11. Почему в движение данных между уровнями иерархической памяти вовлечены как аппаратные, так и программные средства? Какие функции реализованы аппаратно, а какие – программно?

3. ВВЕДЕНИЕ В ПАРАЛЛЕЛЬНУЮ ОБРАБОТКУ

Для увеличения производительности компьютеров в рамках классической архитектуры использовались и используются усовершенствования в различных областях: СБИС-технологий, программного обеспечения, архитектуры и организации компьютера. Одним из источников увеличения производительности является введение микроархитектурных усовершенствований, связанных с использованием параллелизма, заложенного в машинном коде программы.

В компьютерах с классической архитектурой фон Неймана команды программы выполняются последовательно. Это означает, что в каждый момент времени выполняется только одна команда. И выполняются они в том порядке, в котором следуют в программе, за исключением случаев, когда выполняются команды условных и безусловных переходов. *Параллельное* выполнение

программы предполагает наличие в ней *независимых* команд и наличие в процессоре достаточного количества функциональных устройств, на которых они могут выполняться одновременно. Принято различать параллелизм временной и пространственный.

3.1. КОНВЕЙЕРНОЕ ВЫПОЛНЕНИЕ

Представителем *временного* параллелизма является *конвейерное* выполнение программы. Конвейерное выполнение программы предполагает, что каждая команда программы выполняется не на отдельном устройстве за один такт, как на традиционном компьютере, а на устройстве, состоящем из нескольких последовательно соединенных устройств (ступеней), называемом *конвейером*. Таким образом, выполнение команды разделяется на несколько стадий, каждая из которых выполняется на соответствующей ступени конвейера. Типичный набор стадий выполнения команды: выборка команды, декодирование команды, выборка и загрузка операндов, выполнение операции и сохранение результатов в память (Рис. 21). Выигрыш во времени достигается за счет того, что на конвейере на разных ступенях одновременно (параллельно) выполняются несколько команд. Действительно, как только очередная команда завершила свое выполнение на первой ступени конвейера и готова перейти на вторую, следующая за ней команда уже может начать своё выполнение на первой ступени конвейера (Рис. 22а) и т. д.

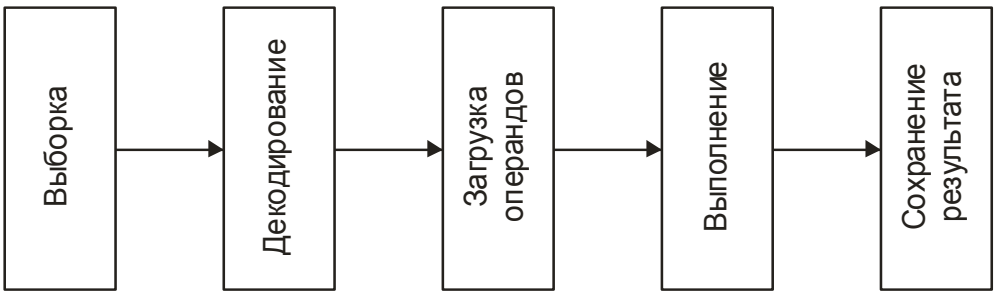

Рис. 21. Пятистадийное выполнение команды

Основными характеристиками конвейера являются пропускная способность и латентность. *Пропускная способность* (или производительность) конвейера – определяется как число команд, выполненных в единицу времени. *Латентность* конвейера – это промежуток времени между моментами попадания команды на конвейер и получения ее результата на выходе конвейера.

Если считать, что для пятиступенчатого конвейера каждая ступень срабатывает за один такт, то первая команда полностью выполнится за пять тактов. Таким образом, латентность этого конвейера составляет пять тактов. Причем, начиная с пятого такта работы конвейера, в выполнение будут вовлечены уже пять команд на всех пяти ступенях. Промежуток времени с момента запуска конвейера и до момента, когда начинают работать все ступени, называется временем разгона конвейера. Для рассматриваемого конвейера время разгона составляет четыре такта. Начиная с пятого такта, после разгона конвейера, его производительность становится равной одной команде в секунду.

Предположим, что при последовательном выполнении без конвейеризации каждая команда логически проходит те же стадии, что и при конвейеризации. Тогда последовательное выполнение можно проиллюстрировать рисунком (Рис. 22, б). Видно, что производительность такого функционального устройства будет в пять раз ниже, чем у пятиступенчатого конвейера (Рис. 22, а). Таким образом, N-ступенчатый конвейер увеличивает производительность процессора в N раз.

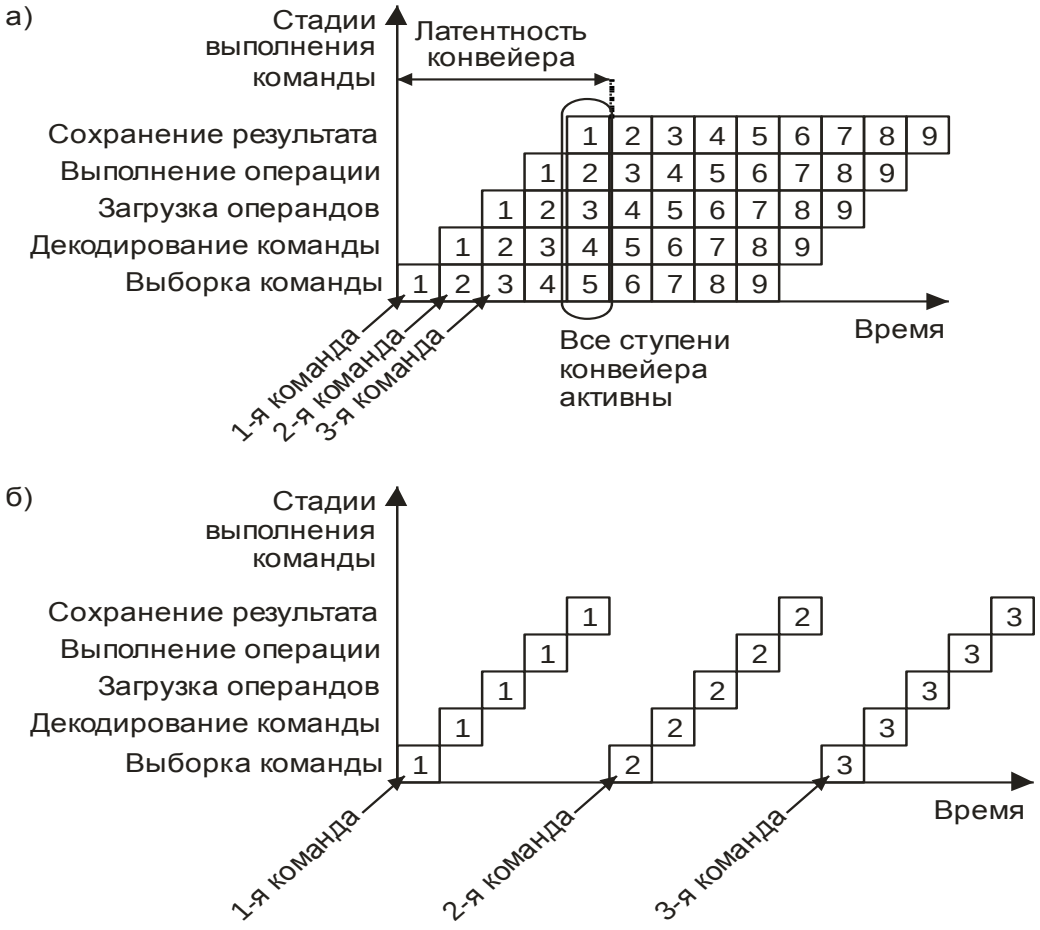

Рис. 22. Временная диаграмма выполнения последовательности команд: с конвейеризацией (а) и без конвейеризации (б)

В реальных процессорных конвейерах ситуация не такая простая. Существует факторы, которые мешают конвейеру следовать описанной выше схеме работы и постоянно сохранять максимально возможную производительность. Можно выделить три таких фактора:

- зависимости по данным,
- зависимости по управлению,
- зависимости по ресурсам.

Зависимость по данным – это ситуация, когда последующей команде требуется прочитать значение той же ячейки памяти (оперативной памяти или регистра), в которую должна записать результат предыдущая команда. Если предыдущая команда записывает значение в регистр только после выполнения стадии сохранения результата, а последующая требует его уже на стадии загрузки

операндов, то она вынуждена ждать на этой стадии дополнительно два такта. Соответственно, выполнение всех последующих команд также задерживается на два такта. Если представить, как команды продвигаются по конвейеру от начала до конца, то, начиная со стадии загрузки операндов, по конвейеру начнет продвигаться «пустое место» размером в две ступени, обычно называемое пузырьком (Рис. 23а).

Для устранения задержек из-за зависимостей по данным в конвейере используется технология ускоренной пересылки данных между ступенями. Суть этой технологии состоит в том, что результат предыдущей команды со ступени, отвечающей за выполнение команды, по специальному каналу отправляется на ступень загрузки операндов, где его ждет последующая команда (Рис. 23, б). Таким образом, простоя конвейера не происходит. Еще одним способом устранения задержек является разнесение зависимых команд в коде программы, заполняя пространство между ними другими независимыми от них командами.

Зависимость по управлению – это зависимость, вызванная командой перехода. В результате выполнения команды перехода на соответствующей стадии становится известно, какая команда должна выполняться после нее. Если это не та команда, которая следует за ней в коде программы, то процессор вынужден сделать сброс конвейера – все последующие команды, которые уже начали выполняться, должны быть удалены с конвейера. И тогда вместо них из памяти организуется загрузка потока команд по вычисленному адресу перехода. В результате на конвейере образуется пузырек размером почти с длину конвейера (Рис. 24, а).

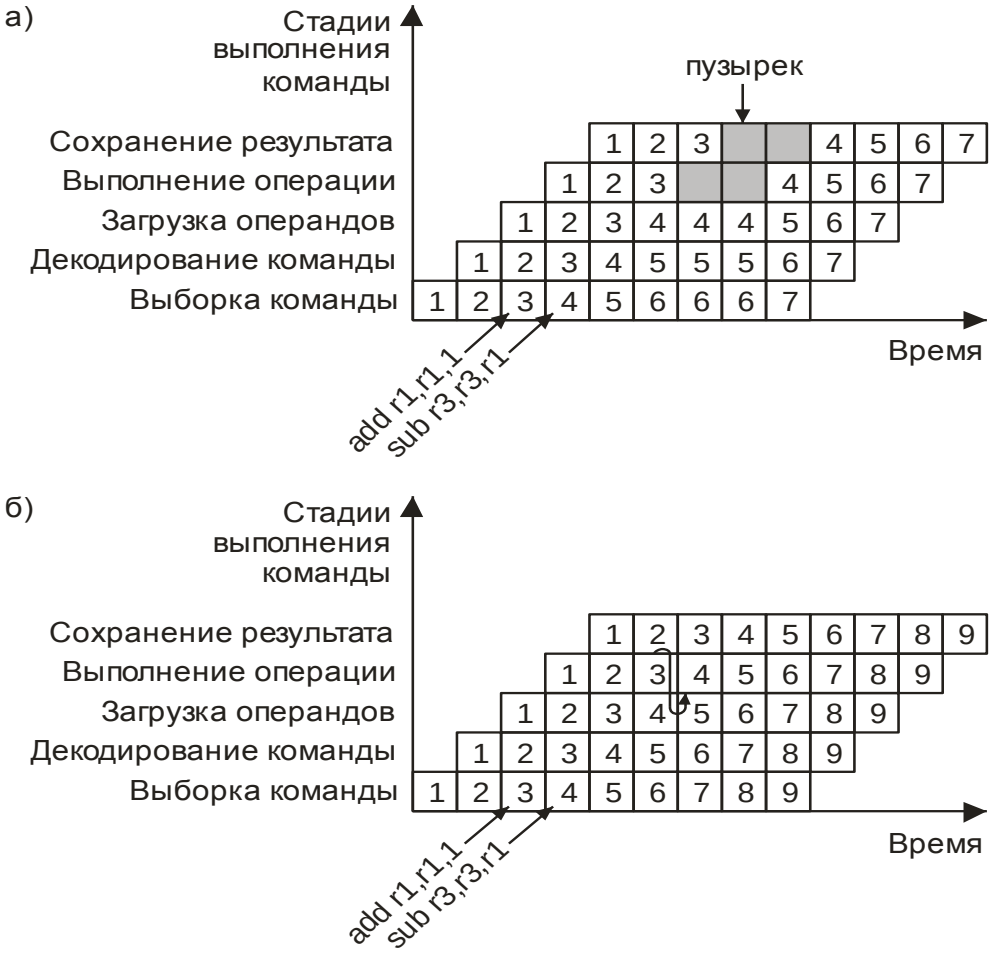

Рис. 23. Временная диаграмма выполнения на конвейере последовательности команд с зависимостью по данным: без ускоренной пересылки данных (а), с ускоренной пересылкой данных между ступенями конвейера (б)

Для устранения зависимостей, вызванных командами перехода, используется механизм раннего обнаружения и предсказания переходов. Команда перехода обнаруживается еще на начальных ступенях конвейера, после чего специальное устройство в процессоре делает предсказание, выполнится ли этот переход, и если да, то по какому адресу. По результатам этого предсказания принимается решение, какую команду загружать на конвейер следующей.

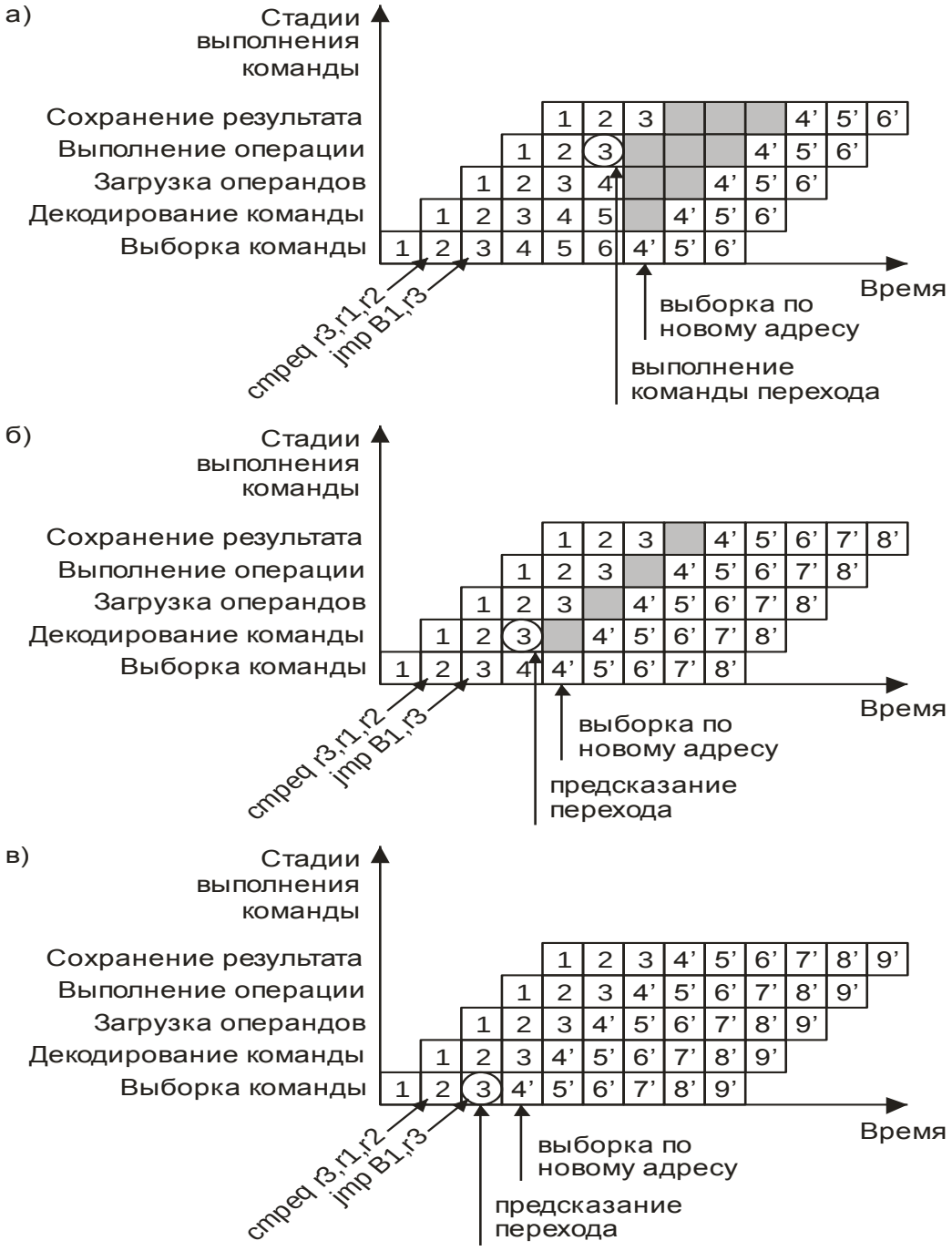

Рис. 24. Временная диаграмма выполнения на конвейере последовательности команд с зависимостью по управлению: без предсказания переходов (а), со статическим предсказанием переходов (б), с динамическим предсказанием переходов (в)

Если переход окажется предсказанным верно, то задержка будет или очень небольшой или нулевой. Если же переход окажется предсказанным неверно, то

происходит сброс конвейера, и задержка будет очень большой. Для предсказания переходов в процессорах использует как статические, так и динамические методы.

Статические методы используют информацию из кода программы, специально выработанную компилятором. При использовании этого способа процессор делает однозначный вывод о срабатывании команды перехода по ее виду. Статическое предсказание срабатывает на стадии декодирования команды (Рис. 24, б). *Динамические* способы основаны на истории срабатывания переходов, которая формируется в процессе выполнения программы. История и статистика срабатывания команд перехода сохраняется в специальных таблицах, на основе которых процессор делает предсказание об их дальнейшем поведении. Динамическое предсказание срабатывает уже на стадии выборки команды (Рис. 24, в), т.к. команды перехода в таблицах истории идентифицируются по своему адресу.

Зависимость по ресурсам – это ситуация, когда на некоторой стадии команда должна обратиться к некоторому ресурсу процессора, который занят другой командой. Таким ресурсом может быть функциональное устройство, регистровый файл, кэш-память и другие. Например, некоторые команды могут проводить на стадии выполнения более одного такта. Все это время следующая за ней команда, которой требуется данный ресурс, будет простаивать.

Для устранения зависимостей по ресурсам на уровне микроархитектуры обычно используется дублирование ресурсов. Например, в процессоре может быть несколько функциональных устройств, которые работают одновременно. Один регистровый файл может быть разбит на два (или даже продублирован), доступ к которым осуществляется независимо. На уровне кода конфликтующие по ресурсам команды обычно разносятся компилятором на некоторое расстояние, и между ними вставляются другие, независящие от них команды.

3.2. ПРОСТРАНСТВЕННЫЙ ПАРАЛЛЕЛИЗМ

Пространственный параллелизм предполагает наличие нескольких устройств, на которых одновременно могут выполняться независимые команды. Команды считаются независимыми, если они не имеют между собой зависимостей по данным, по управлению или по ресурсам. Пространственный параллелизм иногда называют истинным параллелизмом. Различают несколько видов пространственного параллелизма:

- параллелизм *на уровне данных*,
- параллелизм *на уровне команд*,
- параллелизм *на уровне потоков*.

3.2.1. Параллелизм на уровне данных

Параллелизм на уровне данных предполагает одновременное выполнение множества простых (скалярных) однотипных операций над множеством однотипных данных. Это множество операций обычно называется *векторной* операцией, а ее операнды называются векторами. Примером векторной операции может служить покомпонентное сложение двух векторов. Векторные операции часто встречаются при обработке массивов данных в задачах статистического анализа, шифрования, потоковой обработки сигналов и мультимедиа-данных и в численном моделировании.

Процессоры, способные выполнять векторные операции, называются *векторными процессорами*. В отличие от них, процессоры, способные выполнять только скалярные операции, называются *скалярными* (Рис. 25). Все современные процессоры общего назначения изначально были скалярными. Параллелизм на уровне данных был добавлен в них позднее в виде так называемых «*векторных расширений*». В литературе они часто называется *SIMD-расширениями*.

Векторное расширение – это расширение архитектуры, включающее набор векторных регистров и векторных команд. В векторных регистрах размещаются операнды векторных команд. В отличие от простых регистров, векторные

регистры хранят сразу несколько значений. Обычно размер векторного регистра составляет 64, 128 или 256 бит. Например, в регистре размером 128 бит можно разместить сразу четыре вещественных значения одинарной точности (четыре значения по четыре байта).

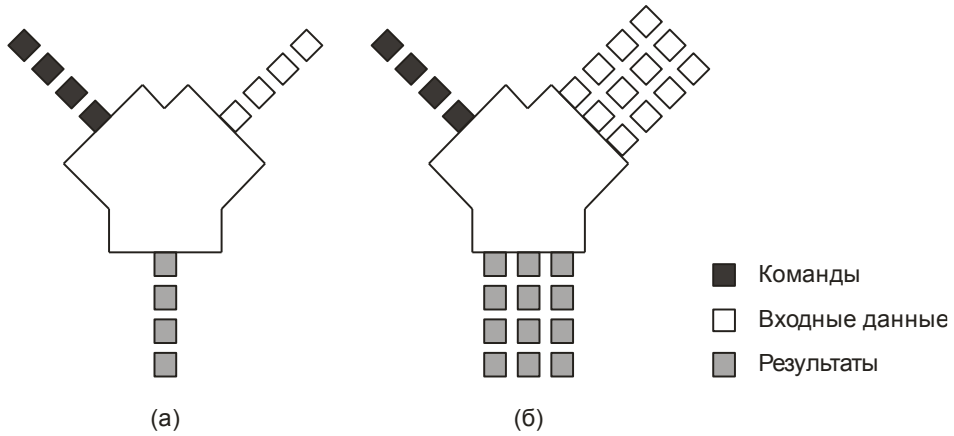

Рис. 25. Принцип работы скалярного процессора (а) и векторного процессора (б)

Векторная команда выполняет одну векторную операцию сразу над всеми значениями одного или нескольких векторных регистров. Скалярные операции, из которых состоит векторная операция, как правило, одинаковые. Выигрыш во времени здесь достигается по ряду причин. Во-первых, декодирование команды выполняется один раз для нескольких одинаковых операций с различными данными. Во-вторых, для компонентов векторной операции не требуется проверять информационные зависимости. И, в-третьих, вычислительные устройства загружаются равномерно, поскольку одна и та же операция над различными данными выполняется за равное время.

В микроархитектуре векторные расширения могут быть реализованы разными способами. Можно выделить три различных способа реализации векторных регистров в микроархитектуре.

- Векторные регистры могут быть реализованы поверх уже существующего регистрового файла и занимать несколько скалярных аппаратных регистров (как в процессорах Intel Pentium III и в процессорах AMD серии K8).

- Другой подход – это увеличение размера аппаратных регистров базовой скалярной архитектуры, чтобы вместить векторный регистр целиком (как это сделано в процессорах Intel Pentium 4 и Xeon и в процессорах AMD серии K10).
- Наконец, аппаратные векторные регистры могут быть реализованы в виде отдельного регистрового файла (как в процессорах IBM POWER6).

Векторные команды также могут выполняться в процессоре по-разному:

- разбиваться на этапе декодирования на несколько скалярных команд и выполняться на скалярных функциональных устройствах (как в процессорах Intel Pentium III и в процессорах AMD серии K8);
- в процессор может быть добавлено специальное векторное функциональное устройство, выполняющее сразу всю векторную операцию (как в процессорах Intel Xeon и в процессорах AMD серии K10).

Наличие векторных расширений в процессоре еще не гарантирует увеличение его производительности, поскольку не всякие вычисления можно векторизовать. Здесь под *векторизацией* понимается преобразование скалярной программы в векторную, т.к. использующую векторные операции. Для векторизации необходимо, во-первых, чтобы в программе было много независимых однотипных операций, и, во-вторых, чтобы эти операции были над данными одного типа. Иногда векторизацию может выполнить компилятор автоматически. Обычно легко векторизуются небольшие циклы, итерации которых не зависят друг от друга и, следовательно, могут быть выполнены параллельно. Однако компилятор может это сделать далеко не всегда и не лучшим образом. В таких случаях вычисления векторизуют вручную либо использует готовые библиотеки векторизованных подпрограмм.

Векторные расширения были введены во многие стандартные архитектуры с целью повышения скорости обработки потоковых данных. Перечислим основные векторные расширения одной из самых распространенных на сегодняшний день архитектур – архитектуры x86-64.

Первой SIMD-расширение в x86-процессор ввела фирма Intel – это было расширение **MMX**. Оно стало использоваться в процессорах Pentium MMX (расширение микроархитектуры P5) и Pentium II (расширение микроархитектуры P6). Векторное расширение MMX работает с 64-битными регистрами MM0-MM7, логически расположенными на вещественных регистрах базовой архитектуры, и включает 57 новых команд для работы с ними. 64-битные регистры логически могут представляться как одно 64-битное, два 32-битных, четыре 16-битных или восемь 8-битных упакованных целых чисел.

Одной из особенностей технологии MMX является целочисленная арифметика с насыщением, используемая, например, при обработке графики. В целочисленной арифметике с насыщением переполнение не является циклическим, как обычно, а вместо этого фиксируется минимальное или максимальное значение. Например, если 8-битное беззнаковое целое значение x=254 увеличить на 3, то в обычной арифметике с 8-битными числами результат будет 1 (в результате переполнения), а в арифметике с насыщением результат будет 255.

Технология **3DNow!** была введена фирмой AMD в процессорах K6-2. Это была первая технология, выполняющая векторную обработку вещественных данных в архитектуре x86. Данное расширение работает с 64-битными регистрами MMX, которые теперь представляются как два 32-битных вещественных числа с одинарной точностью. Базовая система команд расширена 21 новой командой, среди которых появилась команда предвыборки данных в кэш-память данных 1-го уровня. В процессорах AMD Athlon и Duron набор команд 3DNow! был дополнен новыми командами для работы с вещественными числами, а также командами MMX и командами управления кэшированием.

С процессором Intel Pentium III впервые появилось расширение SSE. Это расширение работает с новым независимым блоком из восьми 128-битных регистров XMM0-XMM7. Каждый регистр XMM представляет собой четыре упакованных 32-битных вещественных числа с одинарной точностью. Команды блока XMM позволяют выполнять как векторные (над всеми четырьмя

значениями регистра), так и скалярные операции (только над одним самым младшим значением). Кроме команд для работы с блоком XMM в расширение SSE входят и дополнительные целочисленные команды для работы с регистрами MMX, а также команды управления кэшированием.

В процессоре Intel Pentium 4 набор команд получил очередное расширение – SSE2. Это расширение не добавило новых регистров, но позволило по-новому интерпретировать существующие регистры. Расширение **SSE2** позволяет работать с 128-битными регистрами XMM как с парой упакованных 64-битных вещественных чисел двойной точности, а также с упакованными целыми числами: 16 байт, 8 слов, 4 двойных (32-битных) слова или 2 учетверенных (64-битных) слова. Соответственно, введены новые команды вещественной арифметики двойной точности и команды целочисленной арифметики: 128-разрядные для регистров XMM и 64-разрядные для регистров MMX. Ряд старых команд MMX распространили и на XMM (в 128-битном варианте). Кроме того, расширена поддержка управления кэшированием и порядком выполнения операций с памятью для многопоточных программ.

В последующих процессорах Intel и AMD происходит дальнейшее расширение системы команд на регистрах MMX и XMM – появляются расширения **SSE3**, **SSSE3**, **SSE4** и другие. В них было добавлены новые специализированные команды для ускорения обработки видео, текстовых данных. Особенно следует отметить появившуюся возможность горизонтальной работы с регистрами – выполнение операций с элементами одного векторного регистра.

В архитектуре x86-64, появившейся на смену 32-битной архитектуре x86, число регистров XMM было увеличено до 16-ти: XMM0-XMM15. Кроме того, с приходом этой архитектуры сменилось соглашение на использование вещественной арифметики. Теперь основными регистрами для хранения вещественных значений стали регистры XMM, использующиеся как скалярные или векторные, а основными вещественными операциями стали команды векторных расширений. При этом старый блок вещественной арифметики из архитектуры исключен не был и остается доступен.

В процессорах микроархитектуры Sandy Bridge от Intel и процессорах микроархитектуры Bulldozer от AMD векторные расширения сделали следующий большой шаг в развитии: появилось расширение **AVX (Advanced Vector Extensions)** с новыми векторными регистрами YMM0-YMM15 размером 256 бит. Существующие ранее регистры XMM стали занимать младшую часть новых регистров. Среди особенностей расширения AVX есть поддержка трехоперандных операций вида c = a OP b, а также менее строгие требования к выравниванию векторных данных в памяти.

3.2.2. Параллелизм на уровне команд

Параллелизм на уровне команд предполагает одновременное выполнение процессором нескольких независимых команд программы. Для этого процессоры с параллелизмом на уровне команд (ILP-процессоры) имеют несколько функциональных устройств. Кроме того, каждая стадия конвейера ILP-процессора способна обрабатывать более одной команды за такт. Максимальное число команд, обработку которых конвейер может завершать за такт, называется *шириной конвейера*. Ширина конвейера определяется шириной самой «узкой» стадии конвейера. Реальная производительность ILP-процессора на конкретной программе зависит от количества независимых команд, которые могут быть выполнены параллельно, и ограничивается шириной конвейера. Так как большинство используемых языков программирования являются последовательными (Си, Фортран, Java), то перед ILP-процессорами встает сложная задача выявления параллелизма в последовательных программах.

По способу выявления независимых команд компьютеры данного класса различаются суперскалярные процессоры и VLIW-процессоры (Very Long Instruction Word), т.е. процессоры с длинным командным словом.

Суперскалярный процессор (Рис. 26, а) получает от компилятора программу в виде последовательности команд. Специальное устройство в процессоре, называемое диспетчером, динамически выявляет в этой

последовательности независимые команды, которые затем распределяются по нескольким функциональным устройствам для параллельного выполнения.

Механизм работы диспетчера суперскалярного процессора следующий. Множество очередных последовательно идущих и ожидающих выполнения команд программы образуют так называемое «окно просмотра». В рамках этого окна диспетчер ищет готовые к выполнению команды, по несколько за такт, и отправляет их на выполнение к соответствующим функциональным устройствам. Команда считается готовой к выполнению, если все ее зависимости разрешены (все команды, от которых она зависит, уже выполнены). Типичным размером окна просмотра современного микропроцессора является 100-200 команд.

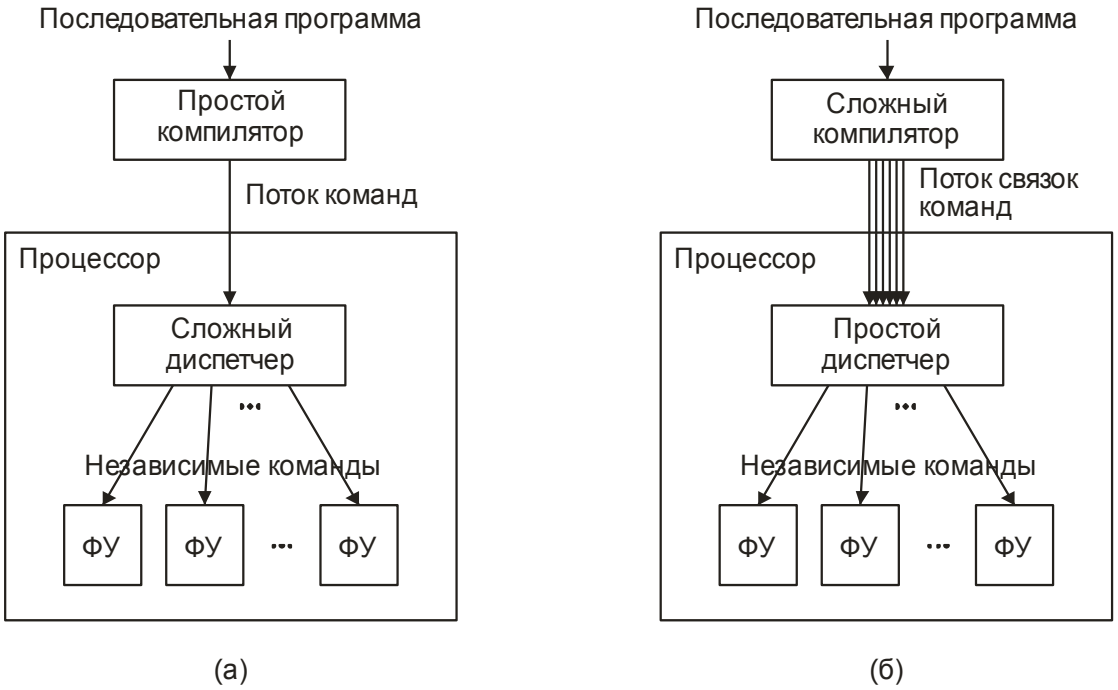

Рис. 26. Процессоры с архитектурой на уровне команд: суперскалярный процессор (а) и VLIW-процессор (б)

В зависимости от способа выбора команд на выполнение различают процессоры с упорядоченным выполнением команд и с выполнением команд вне порядка. В процессоре с *упорядоченным выполнением команд* диспетчер рассматривает команды только в том порядке, в котором они идут во входной

последовательности. Примерами суперскалярных процессоров с упорядоченным выполнением команд могут служить процессоры UltraSPARC T1, Intel Pentium, Intel Atom. В процессоре с *выполнением команд вне порядка* (OoO) диспетчер рассматривает команды в пределах всего окна просмотра. Таким образом, OoO-процессор в действительности выполняет команды не в том порядке, в котором они следуют в программе, а переупорядочивает их без нарушения логики программы. Как результат, производительность OoO-процессора обычно выше. На последней ступени конвейера правильный порядок выдачи результатов восстанавливается, чтобы обеспечить корректное выполнение программы. Примерами суперскалярных микропроцессоров с выполнением команд вне порядка являются процессоры Intel Pentium III, Intel Pentium 4, Intel Xeon, AMD Opteron.

Существенным для производительности суперскалярного процессора является наличие в программе независимых команд. Ограниченный набор архитектурных регистров приводит к появлению в коде так называемых ложных зависимостей, т. е. таких ситуаций, когда две команды логически независимы (например, две команды записи значения в регистр), но используют один и тот же архитектурный регистр. Для повышения производительности суперскалярный процессор вынужден разрешать ложные зависимости с помощью механизма переименования регистров. Переименование регистров – это динамическое отображение архитектурных регистров, использующихся в коде программы, на аппаратные регистры. Количество архитектурных регистров обычно невелико, типичными являются числа 8, 16, 32. Количество аппаратных регистров значительно больше, например, 40, 72, 80, 96. В случае ложной зависимости командам назначаются разные архитектурные регистры, в результате чего они становятся независимыми и могут выполняться параллельно.

Очевидно, что динамическое переименование регистров и выявление независимых команд увеличивают сложность аппаратного обеспечения. Значительная часть кристалла суперскалярного процессора занимает управляющая логика, в результате чего под ресурсы (регистры, функциональные

устройства, кэш-память) остается меньше места. Еще одним недостатком является то, что суперскалярный процессор в динамике не может выявить все независимые команды в программе. Это объясняется, во-первых, требованием использовать быстрые алгоритмы для поиска независимых команд, и, во-вторых, ограниченной длиной окна просмотра. Преимуществом суперскалярного процессора является использование для распараллеливания информации, которая доступна только в момент выполнения программы. Кроме того, производительность суперскалярных процессоров, в отличие от VLIW-процессоров, в меньшей степени зависит от качества кода, что обеспечивает хорошую переносимость программ и хорошую производительность «в среднем».

Компилятор для суперскалярного процессора в распараллеливании команд участвует косвенно, поскольку в коде, который он генерирует, нет указаний на независимые команды. Максимум, что может сделать такой компилятор – это расположить независимые команды в последовательности рядом, чтобы диспетчер процессора мог легко их найти.

Примерами суперскалярных архитектур являются x86/x86-64, ARM, POWER, PowerPC, Alpha, SPARC.

VLIW-процессоры, т.е. процессоры с длинным командным словом, получают от компилятора программу в виде последовательности *связок* команд (Рис. 26б). Команды в каждой связке являются независимыми друг от друга и могут выполняться параллельно. Связка простых команд образует одну составную команду, или длинное командное слово. На Рис. 27 представлена связка команд VLIW-процессора Transmeta Crusoe, состоящая из восьми независимых простых команд.

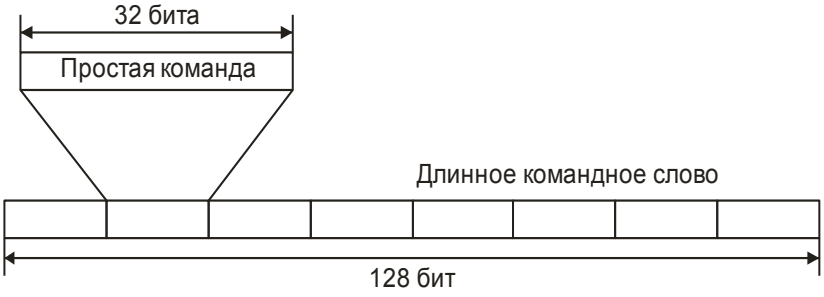

Рис. 27. Структура связки команд VLIW-процессора

Компилятор для VLIW-процессора выполняет статический анализ программы, выявляет независимые операции и формирует последовательность связок для выполнения VLIW-процессором. Число команд в связке определяется архитектурой процессора и равно ширине конвейера. Для каждой простой команды в коде указаны все необходимые аппаратные ресурсы для ее выполнения: аппаратные регистры, функциональные устройства. Если компилятор не нашел достаточное количество независимых команд для формирования связки, он дополняет ее командами NOP («нет операции»). По сути, компилятор выполняет почти полное детальное планирование выполнения потока команд на VLIW-процессоре.

Такой способ организации параллельного выполнения команд значительно упрощает логику работы процессора: нет проверки зависимостей между командами, которые компилятор объявил независимыми, нет внеочередного выполнения команд, поскольку компилятор уже определил порядок выдачи команд, нет необходимости проверять конфликты по ресурсам, поскольку все необходимые ресурсы уже назначены. Диспетчер VLIW-процессора только выполняет закодированные в программе указания. Управляющая логика занимает мало места на кристалле VLIW-процессора, оставляя больше места для ресурсов. В результате ширина конвейера типичных VLIW-процессоров (6-8 команд) больше ширины конвейера типичных суперскалярных процессоров (3-5 команд). Более того, реально достигаемая производительность VLIW-процессоров на большинстве задач выше, чем у суперскалярных процессоров. Это объясняется тем, что компилятору доступен для анализа весь код программы, а не короткое окно просмотра, и компилятор не связан при планировании жесткими временными рамками.

С другой стороны, формирование подробного плана выполнения программы на стадии компиляции приводит к тому, что часть параллелизма не может быть выявлена вообще, поскольку у компилятора нет информации о

зависимостях, которые формируются в процессе вычисления. В ряде случаев команды могут оказаться зависимыми или независимыми при разных запусках программы (например, при различных входных данных программы). Компилятор для VLIW-процессора генерирует параллельный код на самый общий случай, т.е. код, который должен правильно работать при любых входных данных программы. И если для двух команд есть возможность оказаться зависимыми, то компилятор вынужден назначить им последовательное выполнение, даже если в подавляющем большинстве случаев эти команды бы были независимыми. Кроме того, в отличие от суперскалярных процессоров, производительность VLIW-процессоров сильно зависит от качества кода, а VLIW-код жестко «привязан» к конкретной микроархитектуре процессора.

Примерами VLIW-архитектур являются Itanium, Elbrus 2000, процессоры фирмы Transmeta.

3.2.3. Параллелизм на уровне потоков

Термин «поток» в зависимости от контекста может использоваться в различных смыслах. Например, *программный поток* – это термин операционной системы, определяемый как последовательность команд, выполняемых в составе процесса. В данном разделе используется другое определение потока – как *потока команд*, поступающего на вход процессору. Процессор может выполнять поток команд независимо от других потоков команд или одновременно с ними, если у процессора есть аппаратные ресурсы для этого. Параллелизм на уровне потоков (или многопоточность) означает выполнение нескольких потоков команд, которые относятся к разным одновременно выполняющимся программам (или к разным веткам одной параллельной программы).

Различают программную и аппаратную многопоточность. Программная многопоточность – это способ выполнения нескольких потоков команд на одном процессоре путем периодического переключения его с одного потока на другой. Недостатком программной многопоточности является то, что переключение

процессора между потоками средствами операционной системы выполняется сравнительно долго (тысячи тактов процессора). Все современные операционные системы реализуют программную многопоточность, чтобы создать видимость параллельного выполнения программ.

Аппаратная многопоточность – это аппаратно поддержанный способ выполнения нескольких потоков команд на одном процессоре. Аппаратная поддержка многопоточности заключается в организации совместного использования ступеней конвейера и других ресурсов процессора (регистров, кэш-памяти) несколькими потоками команд. При одновременном выполнении команд из нескольких потоков ресурсы процессора загружаются более эффективно, чем при одном потоке, так, что они меньше простаивают. Для операционной системы один многопоточный процессор логически выглядит как несколько процессоров (по числу поддерживаемых им потоков команд).

Существует несколько способов организации аппаратной многопоточности в микроархитектуре:

- При *крупнозернистой многопоточности* конвейер в каждый момент времени выполняет команды только одного потока. Если в какой-то момент произошла длительная задержка (в результате промаха при обращении в кэш-память, зависимости по управлению или по другой причине), то процессор автоматически переключается на другой поток команд и начинает выполнять его команды. Аппаратная поддержка переключения потоков позволяет делать это быстро, без каких-либо задержек. Действительно параллельного выполнения потоков команд здесь, как и при программной многопоточности, не происходит. Ускорение достигается за счет того, что длительные задержки одного потока оказываются скрытыми за выполнением команд другого потока. Примером реализации мелкозернистой многопоточности является технология Hyper-threading в процессорах архитектуры Itanium.
- При *мелкозернистой многопоточности* каждая ступень конвейера на каждом такте переключается между потоками. Таким образом, даже

небольшие задержки в несколько тактов оказываются скрытыми. Параллельного выполнения потоков здесь также не происходит. Примером реализации мелкозернистой многопоточности является процессор UltraSPARC T1.

- При *одновременной многопоточности* ступени конвейера могут обрабатывать одновременно несколько команд с разных потоков. Таким образом, происходит действительно параллельное выполнение команд разных потоков. Ускорение здесь достигается за счет того, что недозагруженные и простаивающие при одном потоке ступени конвейера загружаются командами из других потоков. Примером реализации одновременной многопоточности является технология Intel Hyper-threading в процессорах архитектуры x86/64.

Большинство стадий процессора не различают команды нескольких потоков и могут обрабатывать их единообразно. Ресурсы процессора, отвечающие за хранение данных (кэш-память, регистровые файлы, различные таблицы и буферы), в процессорах с аппаратной поддержкой многопоточности используются одним из следующих способов: они или дублируются для всех аппаратных потоков, или делятся поровну между аппаратными потоками, или же совместно ими используются. Например, кэш-память второго уровня в ранних многопоточных процессорах Intel делилась строго на две части (по числу ядер), а в более поздних стала использоваться совместно всеми ядрами.

Для того чтобы использовать возможности аппаратной поддержки многопоточности, необходимо либо выполнять больше одной программы, либо, если выполняется одна программа, она должна иметь несколько потоков команд, т. е. быть параллельной. При выполнении одной последовательной программы многопоточный процессор работает не лучше (иногда даже хуже за счет меньшего числа ресурсов), чем однопоточный.

3.3. ПРИМЕР ПРОЦЕССОРА С РАЗЛИЧНЫМИ ВИДАМИ ПАРАЛЛЕЛИЗМА

Рассмотрим пример гипотетического процессора, реализующего все рассмотренные виды параллелизма. На Рис. 28 представлена схема конвейера двухпоточного суперскалярного процессора с внеочередным выполнением команд и векторным расширением.

На входе в конвейер стоят два блока выборки команд, каждый из которых производит выборку команд своего потока. Из кэш-памяти команд 1-го уровня в каждый блок выборки поступают блоки данных (кэш-строки или части кэш-строк), из которых блоки выборки выделяют непосредственно коды команд. Далее два потока команд поступают в декодер, который выполняет их декодирование и преобразование во внутренние команды процессора (микрооперации), чередуя потоки на каждом такте. Каждая микрооперация помечается, какому из потоков она принадлежит. На выходе из декодера микрооперации объединяются в один общий поток и попадают в диспетчер.

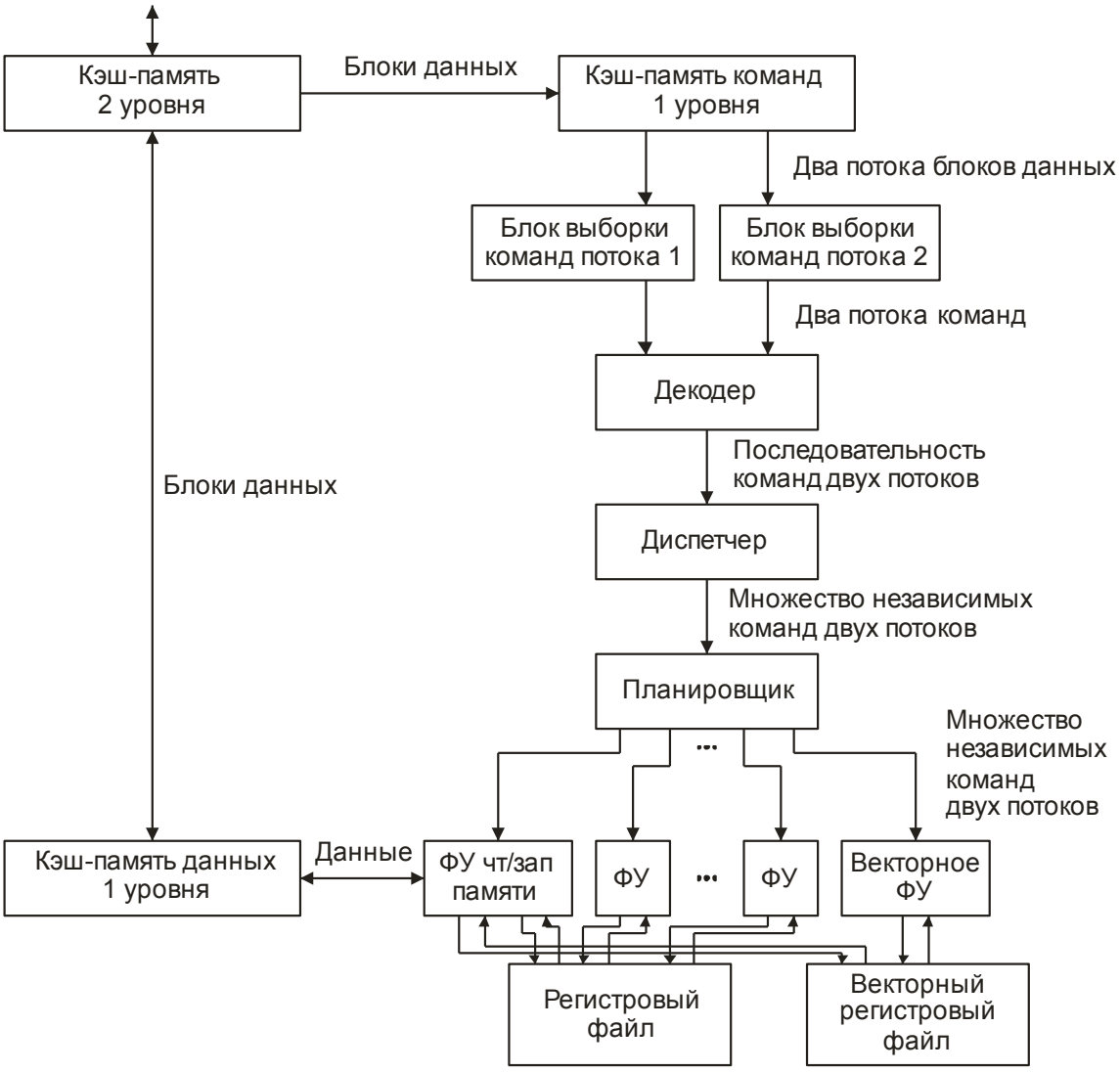

Рис. 28. Структура процессора, использующего различные виды параллелизма

Диспетчер представляет собой большой буфер микроопераций (буфер переупорядочения), организованный в виде очереди. Запись о каждой микрооперации хранится в буфере до самого ее выхода с конвейера. Диспетчер разрешает все зависимости по данным, выполняя переименование регистров и отправляя готовые для выполнения микрооперации в планировщик, по несколько штук за такт. Вновь прибывшие микрооперации попадают в начало очереди, готовые к выполнению микрооперации отправляются в планировщик с любого места очереди, а выполненные микрооперации удаляются с конца очереди. Планировщик получает микрооперации, зависимости по данным у которых разрешены, и распределяет их по функциональным устройствам, т.е. разрешает

зависимости по ресурсам. Функциональные устройства получают команды от планировщика, а данные – из регистрового файла. Два потока команд используют один и тот же файл аппаратных регистров, конкретные ячейки которого распределяет диспетчер. Результаты вычислений функциональные устройства складывают обратно в регистровый файл. По завершении выполнения каждой команды соответствующее функциональное устройство отправляет сигнал диспетчеру. Диспетчер помечает команду как выполненную и проверяет, могут ли теперь начать выполняться другие, зависимые от нее команды, т.е. все ли зависимости по данным у них разрешены. Выполненные команды, находящиеся в конце очереди, удаляются из нее, освобождая место для вновь прибывших.

Кроме того, в процессоре реализовано векторное расширение в виде отдельного регистрового файла (с векторными регистрами большого размера) и отдельного функционального устройства, выполняющего векторные операции над данными в регистровом файле. Также на схеме особо выделено функциональное устройство чтения/записи памяти, связанное с кэш-памятью данных 1-го уровня и со всеми регистровыми файлами. Оно выполняет все операции с памятью: копирование данных из памяти в регистр и из регистра в память. Кэш-память команд 1-го уровня и кэш-память данных 1-го уровня связаны с общей кэш-памятью 2-го уровня, которая в свою очередь связана с более высокими уровнями иерархии памяти (кэш-память 3-го уровня или оперативная память).

Представленный пример отражает принцип функционирования, подходящий для многих современных суперскалярных процессоров. Следует, однако, помнить, что детали микроархитектуры каждого конкретного процессора могут отличаться. В представленном примере многие важные компоненты процессора не отражены, например, функциональное устройство выполнения переходов и блок предсказания переходов, разрешающий зависимости по управлению, блок архитектурных регистров для сохранения текущего состояния процессора, разделение функциональных устройств по типу операций.

3.4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие существуют основные виды и уровни параллелизма? Какие из них используются в современных компьютерах?
2. Каким образом конвейеризация ускоряет выполнение команд?
3. Какие факторы уменьшают производительность конвейера? Как снижается негативное влияние этих факторов?
4. Всегда ли более длинный конвейер означает более высокую его производительность?
5. Какие существуют основные виды ILP-процессоров и чем они отличаются?
6. Какие сильные и слабые стороны имеет выявление ILP в суперскалярных и VLIW-процессорах?
7. Какие цели преследует внеочередное выполнение команд?
8. Какими свойствами должен обладать алгоритм, чтобы его векторизация была эффективной?
9. Чем отличается динамическое предсказание переходов от статического? В чем сильные и слабые стороны каждого из способов?
10. За счет чего ускоряется выполнение программ при использовании в каждом из видов аппаратной многопоточности?

4. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММ

При разработке программ первостепенную роль играет инструментарий, которым пользуется программист. Использование таких инструментов, как компилятор, профилировщик, отладчик позволяет на порядки увеличить результативность его работы, делая посильными и даже простыми такие задачи, которые были бы невыполнимыми для самого квалифицированного специалиста. Сейчас же владение этими инструментами является неотъемлемой частью программирования. В данном разделе представлен ряд важных инструментальных средств, полезных при разработке эффективных программ.

4.1. ОПТИМИЗИРУЮЩИЙ КОМПИЛЯТОР

Компилятор – это программный инструмент, который автоматически преобразует текст программы на некотором языке программирования в машинный код. Они играют важную роль в обеспечении переносимости программ, так как позволяют получать из одной и той же программы машинные коды для выполнения на различных операционных системах и/или архитектурах. Компиляторы также выполняют проверку программы на ошибки и могут оптимизировать программы по различным критериям, таким как общий размер машинного кода или время его выполнения на заданном классе архитектур. Все это делает компилятор важным инструментом программиста.

Традиционно программа, записанная на языке программирования высокого уровня, подается компилятору на вход в виде одного или нескольких текстовых файлов, а на выход компилятор вырабатывает выполняемый файл в формате, который поддерживается операционной системой и программной архитектурой компьютера. Как правило, компиляторы генерируют машинный код для той архитектуры компьютера и той операционной системы, на которых выполняется компиляция. Но возможна и ситуация, когда компилятор генерирует код для другой операционной системы и/или архитектуры компьютера. Это называется *кросс-компиляцией*.

Компилятор может быть реализован как отдельная программная утилита, запускаемая из командной строки, или как часть интегрированной среды разработчика. В настоящем пособии будет рассматриваться компилятор GNU Compiler Collection (GCC), реализованный как утилита командной строки. Строго говоря, GCC является коллекцией компиляторов, объединенных одной оболочкой. Эта оболочка интерпретирует аргументы командной строки и запускает один или несколько своих компиляторов в зависимости от того, на каких языках программирования описаны входные тексты программ, и от целевой архитектуры. Компилятор GCC позволяет компилировать программы, написанные на ряде языков высокого уровня, включая Си и Си++, в бинарный код

для разных архитектур, в том числе для традиционных 32- и 64-битных архитектур (x86, x86-64).

Управлять работой компилятора GCC можно с помощью аргументов командной строки, в которых указываются имена входных файлов с текстами программ и ключи компиляции, управляющие настройками компилятора и режимом его работы. Ключ компиляции – это аргумент командной строки, состоящий из одного или нескольких слов. Первое слово задает конкретную настройку и обычно начинается с одинарного или двойного символа дефиса. Остальные слова (если есть) могут задавать параметры этой настройки.

Рассмотрим примеры использования параметров командной строки для управления работой компилятора GCC. Скомпилировать файл myprog.c можно следующей командой:

```
gcc myprog.c
```

При этом сгенерируется исполняемый файл с именем по умолчанию (например, a.out). Следующая команда скомпилирует файл myprog.c в исполняемый файл с именем prog.exe:

```
gcc -o prog.exe myprog.c
```

Следующая команда выполнит то же самое, но к программе будет применена оптимизирующая компиляция:

```
gcc -O -o prog.exe myprog.c
```

Обратите внимание, что регистр ключей важен. В третьем примере используется два ключа: `-O`, который не имеет параметров и включает оптимизацию при компиляции, и `-o`, который имеет один параметр и указывает имя выходного файла.

Помимо аргументов командной строки работой компилятора GCC можно управлять с помощью директив и атрибутов, располагаемых в исходном тексте программы.

Компилятор GCC поддерживает большое количество режимов работы, выбираемых с помощью соответствующих ключей командной строки. Эти ключи

можно разделить на несколько групп, в зависимости от того, чем они ключи управляют.

- Глобальные настройки, задающие режим работы компилятора, например, выбор языка программирования, выбор имени генерируемого исполняемого файла и т.д.
- Настройки, специфичные для языков Си/Си++, например, выбор диалекта или стандарта языка программирования.
- Настройки сообщений об ошибках и предупреждениях, например, предупреждения об объявленных, но неиспользуемых переменных.
- Настройки отладки, например, нужно ли включать отладочную информацию в генерируемый исполняемый файл.
- Настройки оптимизации (явное указание необходимых оптимизирующих преобразований и задание так называемых уровней оптимизации, рассматриваемых далее).

4.1.1. Уровни оптимизации компилятора GCC

Число оптимизирующих преобразований, поддерживаемых современными компиляторами, велико. Выбор конкретного набора способов оптимизации, которые следует применить в каждом конкретном случае, выходит за рамки компетенции компилятора и должен управляться программистом. Практически каждый отдельный способ оптимизации можно включить или выключить с помощью некоторого ключа командной строки или других средств, но поиск оптимального набора способов, также как и их перечисление в аргументах командной строки, было бы сложным и утомительным. Поэтому в компиляторе GCC и многих других современных компиляторах предлагаются уже готовые наборы настроек компиляции и оптимизации, которые ориентированы на типичные ситуации, часто встречающиеся на практике. Чтобы задействовать тот или иной набор настроек оптимизации требуется указать соответствующий ключ при компиляции. От версии к версии и от одного компилятора к другому эти

наборы могут отличаться, но в целом разработчики компиляторов придерживаются вполне определенного подхода. А именно, компиляторы имеют несколько режимов оптимизации, называемых также уровнями оптимизации. Каждый уровень включает способы оптимизации предыдущего уровня и добавляет новые. При этом на нулевом уровне отсутствуют или практически отсутствуют какие-либо способы оптимизации, а на максимальном уровне (обычно 3 или 4) используются практически все, которые поддерживает эта версия компилятора. Рассмотрим эти уровни.

- На уровне **O0** (ключ компиляции `-O0`) оптимизация кода не производится. Исходный текст программы тривиальными преобразованиями перерабатывается в машинный код. Благодаря этому уменьшается время компиляции программы, что важно, например, при частых сборках больших проектов, где не требуется эффективность кода. Кроме того, полученный таким образом машинный код прост в отладке, так как его структура близко повторяет структуру исходного кода.
- На уровне **O1** (ключ компиляции `-O1`) применяются самые простые и очевидные способы оптимизации, например, размещение переменных программы на регистрах процессора, а не в оперативной памяти. Эти способы оптимизации, как правило, существенно ускоряют программу и крайне редко приводят к ее замедлению. Кроме того, оптимизационные преобразования этого уровня не особенно замедляют процесс компиляции.
- На уровне **O2** (ключ компиляции `-O2`) применяются практически все доступные компилятору способы оптимизации, кроме тех, что ускоряют вычисления за счет увеличения размера кода, или тех, которые в некоторых случаях могут наоборот замедлить выполнение программы. Этот уровень является самым подходящим для большинства программ и обычно используется по умолчанию.
- На уровне **O3** (ключ компиляции `-O3`) обычно применяются все доступные компилятору способы оптимизации, включая те, которые работают не во всех

случаях и могут приводить к увеличению размера исполняемого файла. Так как относительно высока вероятность, что включение уровня **O3** может замедлить программу, то этот уровень не используется по умолчанию.

Часто встречается ситуация, когда оптимизация следующего уровня не дает выигрыша по скорости в сравнении с предыдущим. По умолчанию компиляторы, обычно, используют уровень оптимизации **O2**, т. е. отсутствие ключей компиляции не означает отсутствие оптимизации. Узнать какой именно уровень оптимизации используется компилятором по умолчанию можно из документации к этому компилятору.

Чтобы определить, какой уровень или режим оптимизации является наиболее подходящим для конкретной программы и вычислителя, обычно достаточно скомпилировать программу в нескольких вариантах и с помощью тестирования определить, какая из них является более эффективной (например, по времени выполнения). Тем не менее, следует учитывать, что на других наборах входных данных, а также в иных условиях запуска программы оптимальным может оказаться другой вариант.

Кроме уровней оптимизации в различных версиях компилятора GCC реализованы и другие режимы оптимизации. Рассмотрим некоторые из них.

В режиме **Ofast** применяются все способы оптимизации уровня **O3**, а также добавляется ряд других, таких как использование более быстрых, но приближенных алгоритмов для вещественной арифметики. Этот режим является потенциально опасным, так как снижение точности вычислений может сказаться на эффективности программы или даже ее работоспособности. Там же, где некоторая потеря точности вычислений не является важной, этот режим может существенно ускорить программу.

В режиме **Og** применяются только те способы оптимизации, которые не сильно затрудняют отладку машинного кода. В частности, при использовании данного режима сохраняется общая структура кода и данных программы. Скомпилированную в этом режиме программу программист может отлаживать, имея в полной мере возможность просмотра стека вызовов, фрагментов исходного

текста программы, относящихся к разным уровням этого стека, возможность приостановки программы для каждой строки исходного текста, содержащей операторы и т.п.

В режиме **Os** применяются только те способы оптимизации, которые уменьшают не время выполнения программы, а размер машинного кода. В этом режиме используются некоторые способы оптимизации из уровня **O2.**

4.1.2. Примеры оптимизирующих преобразований в компиляторе GCC

Все оптимизирующие преобразования кода, выполняемые компиляторами, можно разбить на две группы:

- платформенно независимые и
- специфичные преобразования для конкретной платформы.

Если известна архитектура компьютера, на котором будет запускаться программа, то можно включить оптимизацию под эту конкретную архитектуру. Компилятор будет использовать дополнительные команды и другие возможности этой архитектуры, а также учитывать ее особенности для получения более эффективного кода. В обычном режиме компилятор не может этого делать из соображений совместимости.

Список всех ключей оптимизации с аннотациями можно получить с помощью ключа `--help=optimizers`. Рассмотрим примеры некоторых оптимизирующих преобразований, применяемых в GCC.

Удаление мертвого кода – преобразование, удаляющее фрагменты кода, которые не влияют на результат программы. К мертвому коду относят код, который не выполняется ни при каких условиях, и код, изменяющий значения переменных, которые никогда не используются. Это преобразование уменьшает размер выполняемого кода и иногда уменьшает время выполнения, так как исключает выполнение команд, не влияющих на результат.

Преобразование включается ключами `-fdce`, `-fdse`, `-ftree-dce`, `-ftree-builtin-call-dce`, последнее из которых активно на уровнях оптимизации **O2**, **O3**, а остальные – на всех уровнях оптимизации кроме **O0**.

Отображение переменных на регистры процессора. При компиляции программы без оптимизации компилятор отображает все данные программы в оперативную память. В таком случае при каждой операции чтения или записи данных происходит доступ к памяти. Если же данные имеют небольшой размер, то компилятор может отобразить их на регистры, доступ к которым осуществляется значительно быстрее. Компилятор GCC может отображать на регистры только локальные переменные, тогда как компилятор Compaq C Compiler для архитектуры Alpha может также отображать на регистры небольшие массивы. Данное оптимизационное преобразование доступно на уровнях оптимизации **O1**, **O2**, **O3**.

Раскрутка циклов включается ключами GCC `-funroll-loops`, `-funroll-all-loops`. В результате этого преобразования исходный цикл преобразуется в другой цикл, в котором тело цикла содержит несколько тел старого цикла (листинг 1). При этом счетчик цикла меняется соответственно. Этот способ оптимизации может уменьшить время выполнения за счет того, что уменьшается количество команд проверки условия выхода из цикла и команд условного перехода, которые могут приводить к приостановке конвейера команд. Однако иногда раскрутка цикла приводит к увеличению времени выполнения программы. Другой недостаток преобразования – увеличение размера результирующего кода.

Листинг 1. Пример раскрутки циклов

// Исходный цикл	// Цикл после раскрутки на 4 итерации

`for (i=0;i<N;i++)`
`  x[i]=f(i);`
	`for (i=0;i<N;i+=4) {`
	`  x[i  ]=f(i  );`
	`  x[i+1]=f(i+1);`
	`  x[i+2]=f(i+2);`

	`  x[i+3]=f(i+3);` `}`

Встраивание функций. При использовании этого преобразования вместо вызова функции в код встраивается тело функции. При этом ценой увеличившегося размера кода устраняются временные издержки на вызов функции и передачу аргументов. Встраивание функций, размер кода которых меньше размера кода их вызова, или приблизительно равен ему, включается ключом `-finline-small-functions` (включено по умолчанию на уровнях оптимизации **O2**, **O3**). Встраивание более крупных функций включается с помощью ключей `-finline-functions` (включено на **O3**), `-finline-functions-called-once` (не включено только на **O0**), `-findirect-inlining` (включено на **O2**, **O3**). Кроме этих основных ключей есть дополнительные ключи для настройки параметров преобразования. Например, ключ `-finline-limit` позволяет задать максимальный размер функций, которые следует встраивать.

Переупорядочение команд. Команды переупорядочиваются с учетом информационных зависимостей таким образом, чтобы более равномерно и полно загружать вычислительные устройства процессора. Например, две зависимые команды компилятор постарается разнести в коде на некоторое расстояние, если выполнение первой из них занимает длительное время.

Использование расширений процессора (группы специфичных для платформы преобразований). При генерации кода используются дополнительные команды процессора, специфичные для данной архитектуры. В результате код может получиться более быстрым, особенно при векторизации вычислений, однако может потерять переносимость, т. е. не будет функционировать на процессорах с другими микроархитектурами.

Вынос инвариантных вычислений за циклы. Если в цикле присутствуют вычисления, которые не зависят от итерации цикла, то они выносятся за цикл и тем самым многократно не повторяются.

«Перепрыгивание» переходов. Если в программе имеется цепочка последовательных переходов (условных или безусловных), она заменяется на единственный переход, который ведет сразу в окончательный пункт назначения, минуя промежуточные переходы. Преобразование включается ключом `-fcrossjumping` и активно на уровнях **O2**, **O3**.

Устранение несущественных проверок указателей на NULL. Считается, что обращение по нулевому указателю всегда приведет к исключению (и аварийной остановке программы). Поэтому если в коде встречается проверка указателя на ноль после обращения по этому адресу, то такая проверка из кода исключается, так как указатель заведомо не нулевой, если выполнение дойдет до этой точки. Данное оптимизирующее преобразование включается ключом `-fdelete-null-pointer-checks` и выключается ключом `-fno-delete-null-pointer-checks`. Для платформ x86, x86-64 это преобразование активно на всех уровнях, начиная с уровня **O1**.

4.2. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА ОТЛАДКИ ПРОГРАММ

Отладкой называют процесс устранения в программе ошибок, которые были найдены на этапе тестирования. Первая задача, решаемая при отладке, это локализация ошибки, т.е. выявление места в программе, где она была допущена. Степень локализации варьируется от всей программы до модулей, подпрограмм, и, наконец, до конкретных операторов, содержащих ошибку. Вторая задача – это исправление локализованной ошибки.

4.2.1. Принципы работы отладчика

Отладчик – основное инструментальное средство для отладки программ. Он служит для поиска ошибок самого разного вида путем прогона программы в особом, отладочном режиме. Отладочный режим отличается от обычного тем, что в нем выполнение программы можно приостанавливать и продолжать в любом

месте программы и в любой момент, выполнять программу по шагам, смотреть состояние ячейки памяти или регистра. При указании собрать программу для выполнения в отладочном режиме, компилятор добавляет в ее бинарный файл информацию, позволяющую отладчику оперировать с объектами данных и процедурами программы по их именам в исходном тексте.

Отладчик может быть как независимой утилитой (например, GNU GDB), так и частью интегрированной среды разработки программ (Microsoft Visual C++). Базовый набор функций у отладчиков приблизительно одинаков. Отладчики позволяют:

- инициировать выполнение программы в отладочном режиме;
- останавливать выполнение программы;
- приостанавливать и продолжать выполнение программы;
- производить пошаговое выполнение;
- просматривать и изменять значения переменных;
- просматривать и изменять значения ячеек памяти;
- просматривать иерархию вызовов подпрограмм;
- устанавливать и убирать точки останова (англ. breakpoint) по строкам исходного текста программы;
- устанавливать и убирать точки останова по командам процессора в выполняемой программе.

Выполнение программы в отладочном режиме может приостанавливаться в следующих случаях:

- по команде пользователя;
- при достижении в отлаживаемой программе команды завершения выполнения;
- при генерации процессором прерывания, которое сигнализирует о возникновении события или ошибки некоторого вида;
- при достижении точки останова.

Некоторые виды ошибок в процессе отладки могут быть обнаружены

благодаря тому, что процессор при их обнаружении может сгенерировать аппаратное прерывание. Конкретный набор обнаруживаемых таким образом ошибок зависит от архитектуры процессора. Обычно в такой набор входят следующие виды ошибок:

- переполнение стека;
- ошибка доступа к памяти;
- целочисленное деление на нуль;
- неверный код операции.

Пошаговое выполнение программы может производиться с разной степенью детализации:

- по строкам исходного текста программы без захода в вызываемые подпрограммы;
- по строкам исходного текста программы с заходом в вызываемые подпрограммы;
- по командам процессора в выполняемой программе.

Хотя отладчик позволяет локализовать широкий спектр ошибок реализации, глубокие причины их возникновения он не покажет. Для этого после сеанса отладки разработчику необходимо анализировать спецификации, проектную документацию и текст программы.

Существуют ошибки, которые не могут быть найдены с помощью отладчика. Например, бывают ошибки, которые происходят при выполнении обычной версии бинарного выполняемого файла, а при запуске отладочной версии они не проявляются. В таком случае можно использовать подход, называемый трассировкой, который заключается в печати выбранных ячеек памяти (например, глобальных переменных или параметров подпрограмм) в заданные моменты времени. Также бывают ситуации, когда нет возможности использовать отладчик в интерактивном режиме. Например, некоторая программа должна запускаться асинхронно, в непредсказуемые моменты времени. Это типичный режим работы для процедур обработки прерываний или для серверных программ, обрабатывающих клиентские запросы с удаленных машин. Другой

пример – программа должна выполняться в реальном времени, и задержки являются недопустимыми. В подобных случаях также можно пользоваться трассировкой. После каждого запуска трассировочный вывод сохраняется в файл, называемый журнальным. Если некоторый запуск сопровождался ошибками, то можно анализировать соответствующий ему журнальный файл. Для подобных специальных случаев существуют специализированные отладчики, например, отладчики для параллельных программ (TotalView) или отладчики для систем реального времени.

Кроме собственно отладчиков при отладке полезны утилиты, показывающие состояние системы. Они отвечают на такие вопросы, как:

- какие процессы выполняются в данный момент;
- сколько памяти (физической, виртуальной) занимает процесс;
- сколько процессорного времени потребляет процесс;
- сколько потоков у процесса;
- сколько данных принято и получено процессом по сети;
- сколько данных было записано и прочитано с внешней памяти.

Подобные утилиты имеются, например, в ОС Windows. Иногда достаточно использовать Менеджер задач Windows. В ОС Linux ответы на подобные вопросы можно получить с помощью многочисленных утилит, например:

- `top` – информация о выполняемых системой процессах,
- `ifconfig` – информация о сетевых интерфейсах и их загрузке,
- `lsmod` – список подгружаемых модулей ядра, и какими процессами они в текущий момент используются.

4.2.2. Отладчик GNU GDB

GNU GDB – свободно распространяемый и доступный в исходных текстах отладчик. GDB поддерживает широкий круг аппаратных платформ (Intel X86, X86-64, IBM Cell, ARM, MIPS, Atmel AVR и т.д.) и операционных систем, включая, конечно, Linux и Windows. GDB поддерживает отладку для программ,

написанных на широком круге языков высокого уровня: C, C++, Objective C, Fortran, Pascal, Modula-2, Ada. Отладка программ с помощью GDB возможна и для других языков, позволяющих генерировать бинарные исполняемые файлы. В этом случае отлаживаемая программа будет представляться не в виде исходного текста на этом языке, а в виде команд процессора из ее исполняемого файла.

Хотя сам по себе отладчик GDB работает в текстовом режиме, для него разработано множество графических оболочек. Отладчик GDB допускает написание скриптов, позволяющих свести часто повторяющиеся последовательности действий пользователя при отладке, к выполнению одной макрокоманды. Набор команд GDB несколько шире, чем приведенный в предыдущем разделе набор для типичного отладчика. В частности, GDB позволяет не только выполнять команды исходного текста программы, но и формировать фрагменты программы на языке высокого уровня непосредственно при отладке.

Чтобы отлаживать программу с использованием GDB, нужно скомпилировать ее с включением в исполняемый файл отладочной информации. В случае использования компилятора GCC это указывается с помощью ключа `-g`. Запуск отладчика производится командой:

`gdb <имя исполняемого файла>`

GDB предоставляет пользователю следующий базовый набор команд.

- инициировать выполнение программы в отладочном режиме (команда `run`);
- завершить выполнение приостановленной программы и выйти из отладчика (команда `quit`);
- продолжить выполнение программы (команда `continue`);
- произвести пошаговое выполнение по строкам исходного текста программы (команды `step`, `next`);
- произвести пошаговое выполнение по инструкциям процессора (команды `stepi`, `nexti`);
- просмотреть значения переменных и регистров (команды `print`, `info`

registers, `info all-registers`);

- изменить значение переменной или регистра (команда `set`);
- просмотреть иерархию вызовов подпрограмм (команда `backtrace`);
- напечатать ассемблерный листинг, сгенерированный компилятором для указанной функции (команда `disassemble`);
- установить точку останова по строкам исходного текста программы (команда `break`).

Рассмотрим пример отладки программы sample1.c на языке Си с использованием GDB (листинг 2).

Листинг 2.Программа sample1.c

```
01 #include <stdio.h>
02 void perform_division(int a, int b){
03         int c;
04         c = a/b;
05         printf("%d/%d = %d\n", a, b, c);
06 }
07 main(){
08         perform_division(1, 1);
09         perform_division(1, 0);
10 }
```

Команда сборки бинарного образа программы:
`gcc -g -o sample1.bin sample1.c`

В листинге 3 приведена сессия отладки. Жирным шрифтом обозначен текст, вводимый оператором, курсивом – программный вывод на текстовую консоль. Вывод может отличаться в несущественных деталях в зависимости от установленного программного обеспечения и его настроек.

Листинг 3. Пример отладочной сессии GDB

```
01   $ GDB
02   GNU GDB Fedora (6.8-17.fc9)
03   Copyright (C) 2008 Free Software Foundation, Inc.
04   License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
05   This is free software: you are free to change and
```

```
redistribute it.
06   There is NO WARRANTY, to the extent permitted by
law. Type "show copying"
07   and "show warranty" for details.
08   This GDB was configured as "x86-64-redhat-linux-gnu"
09   (GDB)file sample1.bin
10   Reading symbols from /tmp/test1/sample1.bin...done.
11   (GDB)run
12   Starting program: /tmp/test1/sample1.bin
13   1/1 = 1
14   Program received signal SIGFPE, Arithmetic
exception.
15   0x00000000004004e8 in perform_division (a=1, b=0) at
sample1.c:4
16                c = a/b;
17   (GDB)backtrace
18   #0  0x00000000004004e8 in perform_division (a=1,
b=0) at sample1.c:4
19   #1  0x000000000040052e in main () at sample1.c:9
20   (GDB)quit
21   The program is running.  Exit anyway? (y or n) y
22   $
```

В строке 01 пользователь запустил отладчик. В строке 09 пользователем запускается команда загрузки бинарного образа программы. В строке 11 пользователь запускает процесс на выполнение в режиме отладки. В строке 13 отлаживаемая программа осуществляет консольный вывод при первом вызове perform_division. В строке 14 отладчик GDB печатает сообщение о том, что возникло исключение при целочисленном делении на нуль. В строке 15 отладчик напечатал имя функции, внутри которой приостановлено выполнение программы, а также значения параметров вызова этой функции. В строке 16 отладчик напечатал на консоль строку из исходного текста, в которой приостановлено выполнение программы. В строке 17 пользователь запускает команду печати стека вызовов, а в строке 20 – команду выхода из отладчика.

В результате отладки выяснилось, что при втором вызове функции perform_division происходит целочисленное деление на нуль. Для исправления ошибки необходимо в теле данной функции вставить проверку деления на нуль

(текст sample2.c на листинге 4)

Листинг 4. Программа sample2.c

```
01 #include <stdio.h>
02 void perform_division(int a, int b){
03         int c;
04         if(b == 0)
05                 printf("abs(%d/0) = inf\n", a);
06         else{
07                 c = a/b;
08                 printf("%d/%d = %d\n", a, b, c);
09         }
10 }
11 main(){
12         perform_division(1, 1);
13         perform_division(1, 0);
14 }
```

4.3. СРЕДСТВА ДЛЯ ИЗМЕРЕНИЯ ВРЕМЕНИ ВЫПОЛНЕНИЯ ПРОГРАММЫ

4.3.1. Методика измерения времени выполнения прикладной программы

Измерение времени выполнения прикладной программы или ее частей является одним из основных способов контроля характеристик аппаратного и программного обеспечения с точки зрения быстродействия. Такой контроль, с одной стороны, полезен для определения «узких мест» в алгоритме или программе, которые нуждаются в оптимизации, с другой стороны, позволяет судить о реальной производительности компьютера (тесты производительности). Известно, что на время выполнения программы влияют разные факторы:

- характеристики самой программы,
- архитектура и конфигурация компьютера,
- операционная система,
- совместно работающие процессы,
- состояние компьютера на момент старта программы,
- влияние измерителя времени.

Отделить влияние интересующих характеристик от прочих далеко не всегда просто. Для этого существуют приемы, позволяющие снизить влияние нежелательных факторов, а также интерпретировать полученные временные характеристики запусков программ.

Время выполнения программы или ее частей (далее – программы), как правило, определяется разницей показаний некоторого таймера, которые снимаются перед началом выполнения программы и после ее завершения.

Всегда, когда требуется измерить время выполнения программы, его требуется измерить с некоторой точностью. Даже если это явно не оговаривается, то всегда подразумевается. Мерой точности измерения является погрешность измерения. Погрешность измерения – это оценка отклонения измеренного значения величины от ее истинного значения. Однако действительное время выполнения программы неизвестно, поэтому погрешность измерения также неизвестна. Тем не менее, погрешность часто можно оценить. Говорят, что измерение времени выполнено с некоторой точностью, если известно, что погрешность измерения не превышает этой величины.

Погрешность бывает абсолютной и относительной. Абсолютная погрешность изменения времени измеряется в секундах. Относительная погрешность измеряется в безразмерных единицах или процентах и определяется как отношение абсолютной погрешности к действительному времени выполнения программы. Например, абсолютная погрешность в 1 мс даст относительную погрешность в 50%, если весь интервал времени был 2 мс и 0,1%, если интервал был 1 с.

Любой измерительный прибор функционирует с некоторой точностью, и временны́е таймеры – не исключение. Каждый таймер обладает определенной точностью, которая обычно определяется размером временного интервала между последовательными изменениями значения (шага) таймера и специфицируется в его документации. Точность использованного таймера может служить оценкой точности измерения времени выполнения программы. Например, если абсолютная погрешность таймера равна 1 мс, то можно считать, что время

выполнения программы было измерено с абсолютной погрешностью не меньше 1 мс.

В современном компьютере имеется несколько таймеров с разной точностью. Некоторые из них будут рассмотрены далее. Для измерения времени работы программы можно использовать и внешние измерительные приборы, например, секундомер. Однако необходимо помнить, что даже если точность выбранного таймера заведомо выше, чем требуется для заданного измерения, это еще не гарантирует, что полученный результат измерения выполнен с требуемой точностью, т.к., возможно, измерение было искажено влиянием посторонних факторов. Для уменьшения этого возможного влияния существует ряд приемов. Ниже приведена процедура измерения времени выполнения программы.

Процедура измерения времени выполнения программы

1. Пусть заданы допустимые величины абсолютной и относительной погрешности.
2. Выбирается наиболее подходящий таймер исходя из допустимой величины абсолютной погрешности и предполагаемой загрузки компьютера посторонними процессами. Если замеряемый интервал времени достаточно большой (несколько минут, часов), или компьютер загружен посторонними процессами, то наилучшим выбором будет таймер времени работы процесса. Если замеряемый интервал времени очень мал (меньше кванта времени, выделяемого операционной системой одному процессу), то лучше использовать счетчик тактов процессора. В остальных случаях подходящим выбором будет таймер монотонного времени. При выборе должен учитываться тот факт, что абсолютная погрешность таймера не должна превышать допустимую величину абсолютной погрешности.
3. Оценивается относительная погрешность измерения. Для этого производится серия измерений, и в качестве оценки относительной погрешности берется отношение абсолютной погрешности таймера к минимальному полученному значению времени. Чем больше число

экспериментов в серии, тем ближе оценка приближается снизу к точному значению.

4. Если полученная оценка превышает допустимую величину относительной погрешности, то необходимо повторить пункт 3, увеличив замеряемый интервал времени, например, путем многократного запуска интересующего участка программы. Как следствие, относительная погрешность измерения уменьшится пропорционально числу повторений.
5. Если полученная оценка относительной погрешности не превышает требуемой величины, то минимальное полученное значение времени будет результатом измерения.

Существует ряд посторонних факторов, влияние которых может уменьшить точность измерения времени выполнения прикладной программы. Ниже приведен ряд приемов, позволяющих уменьшить их влияние.

1. **Исключение из измерения стадий инициализации и завершения**. Если требуется измерить время работы некоторого фрагмента кода, например, некоторой процедуры, то целесообразно вынести весь код программы, предшествующий вызову процедуры, за первое измерение времени работы программы, а второе измерение выполнять сразу после завершения ее работы. Особенно это касается команд работы с устройствами ввода-вывода (чтение с клавиатуры, вывод на экран, работа с файлами) и в меньшей степени команд работы с оперативной памятью.
2. **Уменьшение влияния кода измерения времени.** Сам код снятия показаний того или иного таймера выполняется не мгновенно. Это означает, что в измеряемый интервал времени частично попадает и время его выполнения. Это влияние следует по возможности уменьшать. Во-первых, не следует снимать показания времени часто, особенно в циклах. В идеале код замера времени должен быть вызван лишь дважды – в начале работы и в конце работы измеряемого фрагмента кода. Во-вторых, необходимо следить за тем, чтобы измеряемый интервал был существенно больше, чем время работы функции замера времени.

3. **Уменьшение влияния посторонних процессов.** В случаях, когда процессор загружен другими процессами (например, системными процессами или задачами других пользователей), для получения более точного результата целесообразно устранить посторонние «тяжелые» процессы или дождаться, когда система будет менее загружена.
4. **Сброс буфера отложенной записи на диск**. В современных операционных системах, как правило, используется механизм кэширования при работе с внешней памятью, например, жесткими дисками. Этот механизм заключается в том, что при записи данных на диск они попадают в буфер отложенной записи, который располагается в оперативной памяти, а фактическая запись содержимого буфера на диск происходит позже. Этот механизм служит для ускорения доступа к файлам и уменьшению износа аппаратного обеспечения. В таких ОС возможна ситуация, когда во время работы вашей программы ОС решит сгрузить накопленные данные на диск, что наверняка повлияет на чистоту эксперимента. Поэтому перед проведением замеров времени следует освобождать буфер отложенной записи на диск. В ОС Linux/UNIX это делается с помощью утилиты `sync`. Эту команду можно набрать перед запуском своей программы из командной строки либо вызвать ее прямо из кода своей программы с помощью системного вызова `system` (см. system (3) man page).

Примечание: Man pages (от англ. manual pages, «страницы документации») – это вид документации, распространенный в системах семейства UNIX/Linux. Для получения справки используется утилита командной строки `man`:

```
man 3 system
```

Содержимое этих документов можно также найти в Интернете через поисковые системы.

4.3.2. Системные таймеры

Компьютер имеет несколько аппаратных и программных таймеров, отражающих течение времени с различных точек зрения. Далее будут рассмотрены некоторые из них.

Системное время

Системное время (system time, wall-clock time) – отражение астрономического времени в системе. Системное время измеряется в количестве временных тактов (не путать с тактами в процессоре) от некоторой выбранной точки во времени. Выбор этой точки и продолжительности такта определяется архитектурой ЭВМ и установленной ОС. В ОС UNIX начальной точкой выбрано 01.01.1970 00:00. На современных компьютерах с архитектурой x86-64 и при использовании современной версии ОС GNU Linux/UNIX шаг изменения системного таймера составляет порядка 1 нс (10^{-9} с). Однако время получения значения системного таймера составляет примерно 0.5 мкс (5×10^{-7} с).. Счет системного времени в компьютере ведет энергонезависимый аппаратный таймер RTC (real time clock). Системное время может устанавливаться программно пользователем или ОС. При получении более точного времени извне (например, с сервера времени), ОС автоматически корректирует системное время, т.е. «старается» уменьшить рассогласование системного времени на компьютере с астрономическим временем. Системное время может сдвигаться как в большую, так и в меньшую сторону, т.е. его монотонность не обеспечивается. По этой причине использовать системное время для измерения времени выполнения программ или их частей не рекомендуется.

Достоинства: позволяет получить астрономическое время.

Недостатки: значение таймера может меняться посторонними процессами в ходе измерения; замеряется время работы всех процессов в системе.

Монотонное время

Монотонное время измеряется в таких же тактах, как и системное время, но выбор начальной точки отсчета произволен. Например, ею может быть время запуска ОС. Это означает, что монотонным временем нельзя пользоваться для получения астрономического времени. Программным способом изменить монотонное время нельзя, поэтому оно в наибольшей степени подходит для замера времени выполнения программы или ее части.

Достоинства: значение таймера не может меняться программным способом.

Недостатки: замеряется время работы всех процессов в системе.

Время выполнения процесса

Таймер времени процесса (process time) – программный счетчик, который отражает использование процессорного времени только конкретным процессом. Таймер времени процесса позволяет определить время работы данного процесса в многозадачной ОС, где каждый процессор (или ядро) выполняет несколько процессов (программ) в режиме разделения времени (Рис. 29). Этот показатель особенно важен, когда процессор сильно загружен другими процессами. В этом случае показания, например, системного таймера могут быть очень далеки от действительного времени работы программы. В каждый момент времени процессор (или аппаратный поток, если процессор поддерживает многопоточность) может выполнять только один процесс. Каждому процессу планировщик процессов выделяет некоторый квант времени, т.е. процессор переключается между процессами с некоторой периодичностью. В обычных Linux/UNIX системах квант времени работы процесса составляет около 10 мс (10^{-2} с), в Windows – 0.5-15.6 мс. Шаг изменения таймера времени процесса может быть меньше кванта времени работы процесса.

Достоинства: замеряется время работы только интересующего процесса.

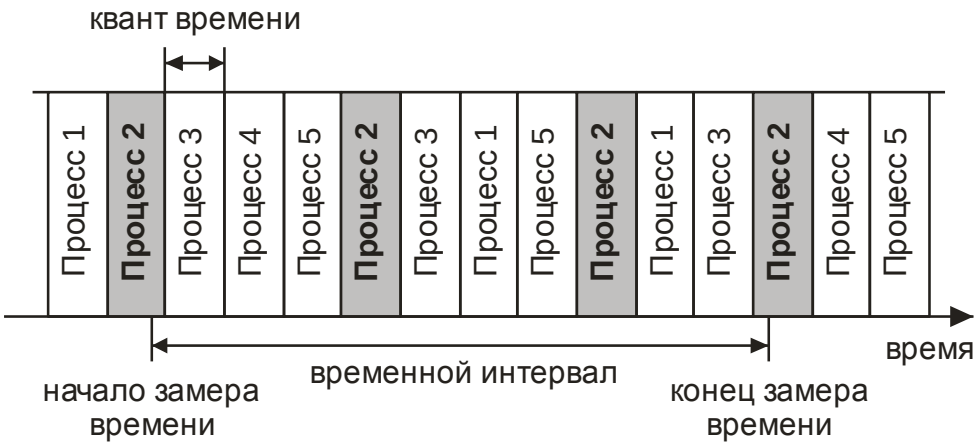

Рис. 29. Измерение времени в многозадачной операционной системе

Время выполнения потока

Таймер времени потока (thread time) – программный счетчик, который отражает использование процессорного времени только конкретным программным потоком. Программные потоки работают в рамках одного процесса в режиме разделения времени. ОС распределяет между ними кванты времени, выделенные на процесс. Параметры таймера времени потока аналогичны параметрам таймера времени процесса.

Достоинства: замеряется время работы только интересующего потока.

Счетчик тактов процессора

Счетчик тактов процессора (CPU time stamp counter) – аппаратный счетчик, значение которого увеличивается на каждом такте процессора. Такт процессора – самый малый интервал времени в вычислительной системе, который теоретически может быть замерен. Поэтому счетчик тактов позволяет с большой точностью измерять малые промежутки времени (вплоть до нескольких команд процессора). Счетчик тактов процессора имеет смысл использовать только для измерения интервалов времени меньших кванта времени, выделяемого процессу/потоку.

Достоинства: минимально возможный шаг изменения таймера.

Недостатки: на процессорах с динамически изменяемой частотой величина такта процессора может меняться, что затрудняет получение величины временного интервала в секундах; на многоядерных системах выполняющаяся программа может мигрировать с одного ядра на другое, счетчики тактов у которых никак не связаны.

4.3.3. Способы получения показаний некоторых таймеров

Утилита `time`

Утилита `time` измеряет время работы приложения во многих конфигурациях ОС GNU Linux/UNIX (листинг 5). Точность, обеспечиваемая утилитой, относительно низкая: стандартный вывод утилиты обеспечивает точность до 10^{-3} с. Утилита `time` выдает следующие временные характеристики работы программы:

- *real* – общее время работы программы согласно системному таймеру;
- *user* – время, которое работал процесс (кроме выполнения системных вызовов);
- *sys* – время, затраченное процессом на выполнение системных вызовов программы.

Пример запуска утилиты `time`:

```
user@host:~$ time ./program.exe
real 0m37.053s
user 0m36.950s
sys  0m0.000s
```

Дополнительная информация: time (1) man page.

Достоинство: не требуется вносить изменения в программу.

Недостаток: измеряется только время работы всей программы, нет возможности измерить время работы отдельных ее частей.

Библиотечная функция `clock_gettime`

Библиотечная функция `clock_gettime` позволяет получить значения различных таймеров в ОС Linux/UNIX. В зависимости от значения параметра, функция позволяет получить значение:

- системного времени,
- монотонного времени,
- времени выполнения процесса,
- времени выполнения потока.

Листинг 5. Пример использования функции clock_gettime

```
01  #include <stdio.h>
02  #include <time.h> // for clock_gettime
03  int main(){
04    struct timespec start, end;
05    clock_gettime(CLOCK_MONOTONIC_RAW, &start);
06    // some work
07    clock_gettime(CLOCK_MONOTONIC_RAW, &end);
08    printf("Time taken: %lf sec.\n",
09      end.tv_sec-start.tv_sec +
10      0.000000001*(end.tv_nsec-start.tv_nsec));
11    return 0;
12  }
```

Функция `clock_gettime` сохраняет значение указанного таймера в структуру `struct timespec`. Структура состоит из двух полей: `tv_sec` и `tv_nsec`, задающих количество секунд и наносекунд (10^{-9} с), прошедших с некоторого неспецифицированного момента времени в прошлом. В приведенном примере (листинг 5) сохраняется значение таймера перед выполнением некоторого кода и после него, затем разница показаний преобразуется в секунды и выводится на экран. Реализация функции `clock_gettime` находится в библиотеке `rt`, поэтому при компиляции программы на некоторых системах необходимо в конце команды компиляции добавить ключ `-lrt`. Пример команды компиляции:

```
gcc –o prog prog.c -lrt
```

Дополнительная информация: clock_gettime (3) man page.

Достоинство: позволяет получить с хорошей точностью значения многих таймеров.

Библиотечная функция `times`

Функция измерения времени работы процесса `times` позволяет определить, сколько квантов времени выполнялся процесс (листинг 6). Если перевести количество этих квантов во время, то можно определить, какое время процесс реально выполнялся. Количество квантов в секунду позволяет узнать системный вызов `sysconf` с параметром `_SC_CLK_TCK`. Типичным размером кванта является 10^{-2} с.

Листинг 6. Пример использования функции times

```
01  #include <stdio.h> // for printf
02  #include <sys/times.h> // for times
03  #include <unistd.h> // for sysconf
04  int main(){
05    struct tms start, end;
06    long clocks_per_sec = sysconf(_SC_CLK_TCK);
07    long clocks;
08
09    times(&start);
10    // some work
11    times(&end);
12
13    clocks = end.tms_utime - start.tms_utime;
14    printf("Time taken: %lf sec.\n",
15      (double)clocks / clocks_per_sec);
16    return 0;
17  }
```

Дополнительная информация: times(2) man page, sysconf (3) man page.

Недостаток: низкая точность.

Машинная команда `rdtsc`

Машинная команда `rdtsc` (Read Time Stamp Counter) используется в архитектуре x86/x86-64 для получения значения счетчика тактов. В других

архитектурах могут существовать аналогичные машинные команды. Команда `rdtsc` возвращает результат в виде 64-разрядного беззнакового целого числа, равного количеству тактов, прошедших с момента запуска процессора. Время в секундах получается делением количества тактов на тактовую частоту процессора. В примере в листинге 7 используется ассемблерная вставка команды процессора `rdtsc`, результат выполнения которой записывается в объединение (union) `ticks` (старшая и младшая части). Разница показаний счетчика тактов преобразовывается в секунды в зависимости от тактовой частоты. Узнать тактовую частоту процессора в ОС Linux/UNIX можно, например, распечатав содержимое системного файла `/proc/cpuinfo`. Но необходимо помнить, что на современных микропроцессорах тактовая частота может динамически изменяться.

Листинг 7. Пример использования машинной команды rdtsc

```
01  #include <stdio.h> // for printf
02  int main(){
03    union ticks{
04      unsigned long long t64;
05      struct s32 { long th, tl; } t32;
06     } start, end;
07    double cpu_Hz = 3000000000ULL; // for 3 GHz CPU
08
09  asm("rdtsc\n":"=a"(start.t32.th),"=d"(start.t32.tl));
10    // some work
11    asm("rdtsc\n":"=a"(end.t32.th),"=d"(end.t32.tl));
12
13    printf("Time taken: %lf sec.\n",
14      (end.t64-start.t64)/cpu_Hz);
15    return 0;
16  }
```

Достоинство: минимально возможный шаг изменения таймера позволяет замерять самые маленькие временные интервалы.

Недостатки: непереносимость – команда используется только в архитектуре x86/x86-64, необходимо использовать ассемблерные вставки в программе.

4.4. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Для каких целей предназначены различные уровни оптимизирующей компиляции?
2. Какова область применения отладчиков? Какие виды ошибок удобно выявлять с помощью отладчика, а какие – нет?
3. Какие есть способы получения показаний различных таймеров? В чем их достоинства и недостатки?
4. Какие гарантии дает использование методики измерения времени?
5. Почему в компьютере существуют различные таймеры?

5. РЕКОМЕНДАЦИИ ПО ЭФФЕКТИВНОМУ ПРОГРАММИРОВАНИЮ С УЧЕТОМ ОСОБЕННОСТЕЙ ПОДСИСТЕМЫ ПАМЯТИ

В разработке и выполнении программы, как правило, участвуют три действующих лица: программист, компилятор и компьютер (иногда обходятся без компилятора, но в этом случае просто программист или компьютер перенимают часть его работы). Такое разделение не случайное, и понимать роль каждого из них необходимо для эффективного использования компьютера.

5.1. РОЛИ ПРОГРАММИСТА, КОМПИЛЯТОРА И КОМПЬЮТЕРА В ОПТИМИЗАЦИИ ПРОГРАММ И ИХ ВЫПОЛНЕНИЯ

Оптимизация времени выполнения программы производится на нескольких этапах ее разработки и выполнения:

- при разработке алгоритма;
- при написании текста программы;
- при компиляции программы оптимизирующим компилятором;
- при выполнении программы на микропроцессоре с современной микроархитектурой.

Каждый из этапов имеет свои особенности и разную степень влияния на конечный результат.

Изначально программист имеет некоторую спецификацию программы, то есть описание того, что программа должна делать. Такая постановка задачи не ограничивает программиста в выборе алгоритма её решения. Программист изучает спецификацию программы и разрабатывает алгоритм, удовлетворяющий этой спецификации. При этом одной и той же спецификации программы может удовлетворять множество различных алгоритмов. Например, если требуется разработать программу, сортирующую элементы массива по возрастанию значений (спецификация), то программист может применить любой алгоритм сортировки: сортировку «пузырьком», подсчётом элементов, быструю сортировку и так далее. Все эти алгоритмы удовлетворяют одной и той же спецификации: они сортируют элементы массива по возрастанию, но имеют разные оценки временной сложности.

Когда программист разрабатывает алгоритм или создаёт новый, он выбирает такой алгоритм, программа которого будет удовлетворять заданным требованиям по времени выполнения программы, расходу памяти, точности. Процесс выбора оптимального алгоритма относительно заданного критерия называется *алгоритмической оптимизацией*. Одним из лучших трудов по алгоритмической оптимизации являются работы [7–10].

Выбрав алгоритм, программист записывает его на каком-либо языке программирования, и в результате получается программа. Программа – это выполняемое представление алгоритма, т. е. такая запись алгоритма, по которой возможно его выполнить автоматически. Очевидно, что один и тот же алгоритм можно реализовать различными программами. Способность эффективно запрограммировать тот или иной алгоритм также определяется квалификацией программиста. Обычно самая первая реализация алгоритма оказывается не самой эффективной. Поэтому, после проверки правильности работы программы, программист обычно старается повысить ее эффективность, выполняя определенные преобразования. Изменение программы с целью повышения её

эффективности (без изменения алгоритма) называется *программной оптимизацией*.

Разработанную программу компилятор перерабатывает в машинный код. Машинный код – это такое представление программы, которое может поступать на вход процессору для выполнения. Поскольку большинство современных компиляторов оптимизирующие, они, кроме преобразования программы в машинный код, еще и оптимизируют программу, то есть улучшают её характеристики посредством тех или иных преобразований. Такие оптимизации относится к программным. Программист должен знать, какие оптимизирующие преобразования способен выполнять компилятор, поскольку чрезмерная ручная оптимизация может помешать компилятору сгенерировать более эффективный код. Хорошим справочником по программной оптимизации, а также источником информации о возможностях компиляторов по оптимизации программ, может служить [28].

Следует понимать, что алгоритмическая оптимизация, как правило, даёт бóльшую выгоду, чем программная оптимизация. Практика показывает, что выбор более эффективного алгоритма может сократить время выполнение программы на порядки, тогда как программные оптимизации, как правило, сокращают время выполнения программы всего лишь в несколько раз. Поэтому эффективность программы зависит в первую очередь от выбора алгоритма. Более того, правильный выбор алгоритма позволяет в принципе получить саму возможность выполнения программы за конечное время [27].

После того, как программа скомпилирована, к её выполнению приступает компьютер. Компьютер также старается сократить время выполнения программы, выполняя различные оптимизирующие преобразования. Такие оптимизации называются оптимизациями времени выполнения (runtime-оптимизации). Цели компьютера и компилятора по сокращению времени выполнения программы совпадают, однако, условия, в которых они работают, имеют несколько существенных отличий.

- Во-первых, компилятор при генерации кода не так ограничен во времени, как микропроцессор при его выполнении. Все возможные оптимизации, которые можно выполнять заранее, следует выполнять именно во время компиляции. Более того, поскольку скомпилированная программа запускается много раз, то для получения более эффективного машинного кода имеет смысл тратить на компиляцию больше времени.
- Во-вторых, микропроцессор выполняет программу на конкретном наборе входных данных, в то время как компилятор этой информации не имеет и должен предусмотреть варианты выполнения программы на любых входных данных. Например, одна из оптимизаций, выполняемых суперскалярным процессором – это переупорядочение команд с целью перераспределения вычислительной нагрузки во времени. Но определить, действительно ли команды можно менять местами, иногда можно только во время выполнения, когда известно, по каким адресам в память будет осуществляться доступ. Компилятор в такой ситуации не имеет права выполнять перестановку, поскольку эти адреса ему неизвестны, и при запуске на некоторых данных команды могут оказаться зависимыми одна от другой, и их перестановка станет недопустимой. Кроме того, процессор «знает», какие его ресурсы (память, функциональные устройства и т.д.) сейчас перегружены, а какие наоборот простаивают без загрузки.
- В-третьих, компилятору доступен код всей программы. Он может анализировать его и выполнять глобальные оптимизации – оптимизации, при которых учитывается взаимное влияние различных частей программы друг на друга. Компьютер во время выполнения программы этой информации не имеет. Хотя машинный код и находится где-то в его памяти, но анализировать его, пытаясь выявить структуру программы, нет ни времени, ни удовлетворительных способов.

5.2. ЭФФЕКТИВНОЕ ПРОГРАММИРОВАНИЕ С УЧЕТОМ ПАМЯТИ

Практика показывает, что программа, написанная без учета свойств архитектуры компьютера, обычно позволяет достичь лишь малой доли от его теоретически достижимой производительности. Одна из основных причин – простои процессора в ожидании данных и команд из оперативной памяти. Сократить эти простои до минимума (даже при оптимальном выборе алгоритма решения задачи) на архитектурном уровне сложно, поскольку время доступа в оперативную память превосходит время преобразования данных в процессоре, и этот разрыв имеет тенденцию к увеличению. Поэтому проблема сокращения простоев процессора решается на программном уровне, т. е. силами компилятора и программиста.

Простои процессора в ожидании данных из оперативной памяти говорят о том, что не все данные находятся в кэш-памяти в момент обращения процессора к ним. Иногда причиной этого становится объективная невозможность уместить все необходимые данные в кэш-памяти (например, если размер активно используемых данных превышает объем кэш-памяти или суть решаемой задачи такова, что ней отсутствует локальность ссылок в оперативную память в пространстве и времени). В этом случае исправить ситуацию невозможно. Но часто причины простоев обусловлены тем, что программа написана без учёта особенностей работы кэш-памяти, принципов организации компьютера. В этом случае изменение программы таким образом, чтобы она лучше эксплуатировала возможности кэш-памяти, может значительно ускорить её выполнение.

Рассмотрим типичные ситуации, которые могут встречаться в программах, и которые приводят к неэффективной работе кэш-памяти.

Данные на границе блоков памяти. Если некоторая переменная разместились в памяти на границе двух блоков, то для её загрузки в кэш-память понадобится загрузить оба этих блока, хотя размер переменной может быть меньше одного блока. Это означает, что в кэш-памяти будет занято больше места, чем требуется, а сама загрузка потребует вдвое больше времени.

Разреженное размещение данных в памяти. Если данные, с которыми работает программа, хаотично рассредоточены по различным адресам, то обработка таких данных будет малоэффективна, так как при их загрузке в кэш-память блоками вместе с запрошенными переменными туда будут попадать и ненужные данные, занимая место в кэш-памяти и нагружая канал связи между кэш-памятью и оперативной памятью.

Размещение и обработка данных в памяти с кэш-буксованием. Если данные располагаются или обрабатываются в памяти со сдвигом, кратным размеру банка кэш-памяти, то возникает буксование, что означает увеличение количества кэш-промахов.

Нарушение принципа локальности ссылок во времени. Если обращения к некоторому элементу данных в памяти рассредоточены во времени, то к моменту очередного обращения к ним высока вероятность, что данные уже были вытеснены из кэш-памяти другими данными, что приводит к кэш-промахам.

Неупорядоченная обработка данных. Если данные в памяти обрабатываются непоследовательно, бессистемно, то такие механизмы кэш-памяти как блочная загрузка и аппаратная предвыборка данных не помогают, а мешают эффективной работе кэш-памяти. Как следствие, такой обход памяти является неэффективным.

Для преодоления этих проблем применяют различные оптимизирующие приемы, которые позволяют во многих случаях добиться ускорения выполнения программы. Далее более подробно рассматриваются описанные проблемы и способы их устранения.

5.2.1. Данные на границе блоков памяти.

С точки зрения отображения данных из оперативной памяти на кэш-память, первая имеет блочную структуру, где размер блока равен размеру кэш-строки. По адресу переменной можно определить, в каком блоке памяти она находится. Переменные и структуры данных, имеющие размер более одного байта, могут

располагаться в памяти на стыке двух блоков, т.е. первые несколько байт переменной могут располагаться в одном блоке, а несколько – в следующем (Рис. 30а). Таким образом, во-первых, время доступа к такой переменной практически удваивается, так как вместо одного блока требуется загрузить два. Во-вторых, для хранения переменной в кэш-памяти потребуется две строки вместо одной. Кроме того, в некоторых архитектурах (например, вычислительные ядра процессора Cell) доступ в память, например загрузка переменной в регистр, может осуществляться только по адресу, выровненному относительно размера переменной. Если же требуется загрузить переменную, расположенную по невыровненному адресу, то для этого потребуется загружать её в несколько шагов, сначала загрузив одну часть, потом другую, и затем объединив их в одном регистре. Всё это негативно сказывается на скорости доступа в память.

Рис. 30. Примеры невыровненных (а) и выровненных (б) переменных в памяти

Для решения этой проблемы применяют *выравнивание* данных. Рассмотрим понятие выровненных и невыровненных данных. Говорят, что элемент данных (переменная, массив и т.п.) выровнен относительно смещения *S*, если адрес первого байта этого элемента кратен *S*. Например, если переменная *x* расположена в памяти по адресу 70, то она является выровненной относительно смещения 10 и невыровненной относительно смещения 16. Также, из сказанного в разделе 2.3 про блочную структуру памяти следует, что блоки в памяти выровнены относительно своего размера.

Очевидно, что если переменная выровнена относительно размера блока памяти, то она не попадёт на границу двух блоков. Также, если размер переменной делит нацело размер блока, то выравнивание переменной по её размеру также означает, что переменная не пересечет границу двух блоков.

Выравнивание данных – это изменение положения переменных в памяти таким образом, чтобы они были выровнены относительно некоторой величины. Для массивов и больших структур данных применяют выравнивание относительно размера блока. Для обычных переменных применяют выравнивание по размеру переменной. На Рис. 30 приведены примеры невыровненных (а) и выровненных (б) переменных, причём переменная a выровнена относительно своего размера, а переменная b выровнена, кроме этого, еще и относительно размера блока.

Выравниванием данных в оперативной памяти для обычных переменных занимается компилятор, и, как правило, человеку не нужно об этом заботиться. Внимание следует уделить пользовательским структурам данных и динамически выделяемой оперативной памяти. Чтобы проверить выровненность данных, можно преобразовать адрес переменной в число и проверить его делимость на интересующее смещение, например, на размер строки кэш-памяти.

Возможен особый случай нарушения выровненности данных. В языках C/C++ допускается явная работа с указателями. Сдвиг выровненного указателя на число байт, не кратное размеру элемента, и последующее его использование приведет к обращению в оперативную память по невыровненному адресу (листинг 8). Таких случаев следует избегать.

Листинг 8. Пример доступа к невыровненным данным

```
1   double x[N+1], *p=(double*)((int)x+3);
2   for (i=0;i<N;i++) p[i]=1.;
```

Управлять выравниванием данных можно с помощью ключей и директив компилятора, которые позволяют запретить и разрешить выравнивание, установить размер блока, относительно которого выравнивать.

Для динамического выделения выровненной памяти можно воспользоваться такими функциями, как `_mm_malloc` или `posix_memalign` (см. posix_memalign(3) man page). Можно также выделить область памяти большего размера с помощью стандартной функции `malloc` и сделать в начале области необходимый для выравнивания отступ (листинг 9).

Листинг 9. Пример выравнивания данных вручную

```
1   ptr = malloc(size + 64);
2   ptr = (ptr % 64) ? (ptr & 0xffffffc0) + 64 : ptr;
```

5.2.2. Разреженное размещение данных в памяти

Ещё одним случаем, когда данные располагаются в памяти неэффективно с точки зрения подсистемы памяти, является ситуация разреженного размещения данных. Речь здесь идёт не об одиночных элементах в памяти, а о группах переменных, связанных локальностью обращений во времени. Такая группа переменных может размещаться в памяти плотно или разреженно (Рис. 31). Плотное размещение означает, что некоторая область памяти заполнена в основном этими переменными, и в этой области отсутствуют вовсе или встречаются редко неиспользуемые ячейки, или используемые под хранение других переменных. Разреженное размещение, напротив, означает, что данные «разбросаны» по памяти на значительные расстояния друг от друга (больше размера кэш-строки).

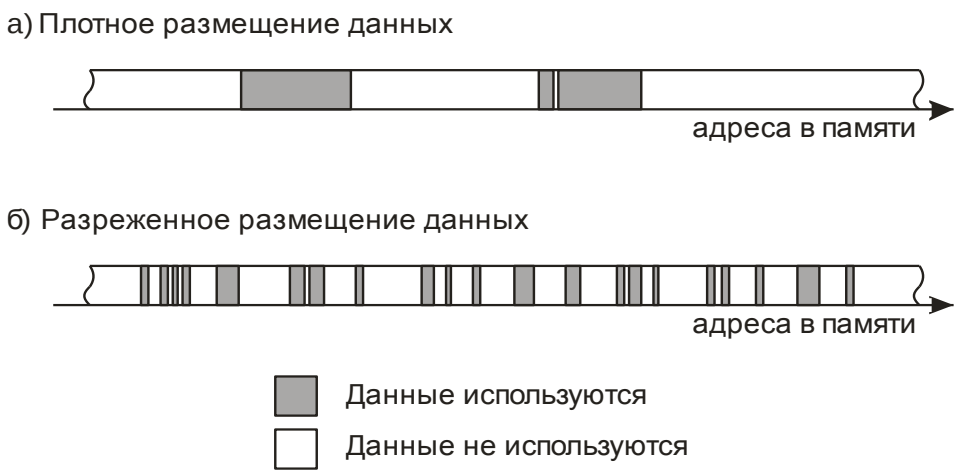

Рис. 31. Плотное (а) и разреженное (б) размещение данных в памяти

С точки зрения подсистемы памяти важно, чтобы данные располагались плотно, так как обращения к плотно расположенным данным, как правило, выполняются быстрее, чем к разреженным. Это обусловлено двумя основными причинами – блочной загрузкой и аппаратной предвыборкой данных в кэш-память.

Плотным размещением простых переменных (глобальных и локальных) в оперативной памяти занимается, как правило, компилятор. Однако он не может изменить размещение и структуру данных в оперативной памяти, которую задаёт программист. Например, пусть в некоторой задаче имеется множество точек на плоскости, представленных своими координатами – парами вещественных чисел. Программист может создать массив пар чисел (вариант 1 в листинге 10), а может задать два массива с координатами точек по осям x и y отдельно (вариант 2 в листинге 10). Расход памяти при этом не изменится, но изменится размещение переменных в оперативной памяти. Если при обработке точек требуются обе их координаты, то первый способ размещения данных в оперативной памяти будет более эффективным за счёт того, что часто обе координаты точки будут находиться в одном и том же блоке оперативной памяти (Рис. 32а). Особенно заметной разница будет при обработке точек в случайном порядке. Если же обработка точек осуществляется по каждой координате независимо от другой, то

предпочтительным будет второй вариант (Рис. 32б), особенно при обработке точек по порядку (листинг 10).

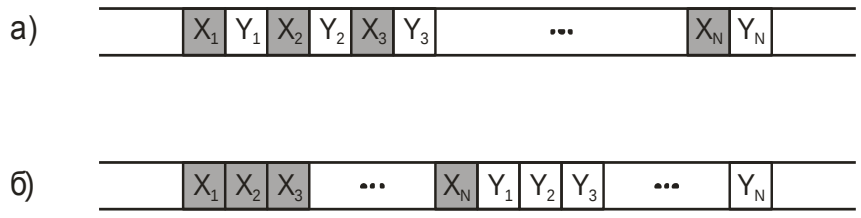

Рис. 32. Два способа размещения данных: массив структур (а) и несколько массивов (б)

Листинг 10. Представление в программе множества точек на плоскости

Вариант 1	Вариант 2
`struct point { float x, y; };` `struct point points[N];`	`float x[N];` `float y[N];`

Другой пример, когда плотность размещения данных в оперативной памяти может быть повышена, связан с выравниванием данных внутри структур данных, которое выполняет компилятор. Рассмотрим листинг 11.

Листинг 11. Выравнивание данных внутри структур

```
1   struct Item1
2   { int a;      // 4 байта
3     double x;   // 8 байт
4     int b;      // 4 байта
5   };
```

```
1   struct Item2
2   { double x;   // 8 байт
3     int a;      // 4 байта
4     int b;      // 4 байта
5   };
```

Структура `Item1` в памяти будет автоматически размещена так, как показано на Рис. 33а. Видно, что два поля по 4 байта в ней не используются, а только занимают место: первое поле обеспечивает выравнивание элемента `x`, второе поле обеспечивает выравнивание размера всей структуры.

Компилятор дополняет структуру до размера, кратного размеру наибольшего ее элемента. Если использовать в программе большой массив таких

структур, то потери оперативной памяти и места в кэш-памяти могут оказаться существенными.

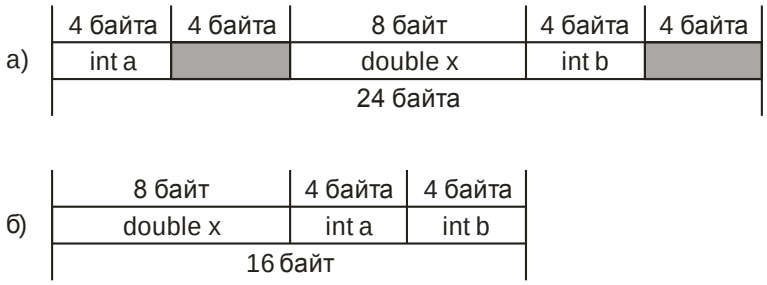

Рис. 33. Размещение данных внутри структур: неоптимальное (а) и оптимальное (б)

Более плотным является размещение в оперативной памяти структуры `Item2`, которое показано на Рис. 33б. Элементы этой структуры упорядочены по убыванию их размеров, что позволяет эффективно разместить их в оперативной памяти.

Можно выделить ряд рекомендаций, которые позволяют избежать проблем, связанных с плотностью размещения данных в оперативной памяти.

1. Разрешить компилятору производить выравнивание данных (обычно он это делает по умолчанию).
2. Избегать особых ситуаций явного сдвига адреса элемента на значение, не кратное размеру элемента (как в листинге 8).
3. Группировать данные в соответствии с их использованием.
4. Размещать данные в структурах как можно более плотно.

5.2.3. Данные, смещённые на величину, кратную размеру банка кэш-памяти

Существует особенный случай размещения данных, когда вследствие организации кэш-памяти скорость работы программы может упасть особенно сильно за счёт эффекта буксования кэш-памяти. Как известно, этот эффект появляется, когда в программе многократно происходят обращения к нескольким

элементам, отстоящим в памяти на величину, кратную размеру банка ассоциативности кэш-памяти, а число таких элементов превышает степень ассоциативности этой кэш-памяти. Например, в процессорах DEC Alpha 21264 и AMD Opteron 840 кэш-память первого уровня – множественно-ассоциативная со степенью ассоциативности равной 2 и размером 64 КБ. Размер банка кэш-памяти равен 64 КБ / 2 = 32 КБ. Значит, обращения к трем или более элементам, отстоящим друг от друга на расстояние, кратное 32 КБ, вызовут эффект буксования кэш-памяти.

На Рис. 34 показано время работы программы численного моделирования, в которой выполнялся обход трехмерного массива 8-байтовых элементов размером N×N×N. При обращении к каждому элементу во время обхода массива также выполнялись обращения ко всем его соседям по трем измерениям, отстоящим в памяти в обе стороны на расстояния в 1, N и N×N элементов. Видно, что при N=128, происходят три обращения в память на расстоянии N×N×8Б = 128 КБ, что приводит к кэш-буксованию и существенному замедлению работы программы.

Рис. 34. Влияние эффекта буксования на время выполнения программы

Далее будут рассмотрены несколько типичных примеров ситуаций, в которых может появляться эффект «буксования» кэш-памяти.

Рассмотрим пример из листинга 12. Во внутреннем цикле происходит перебор элементов массива `a` с шагом 4096 элементов, т.е. 4096×8 Б = 32768 Б = 32 КБ, что приводит к буксованию кэш-памяти.

Листинг 12. Буксование кэш-памяти при обработке одного массива

```
1   double a[4096000], sum[4096];
2   int i, j;
3   for(i=0; i<4096; i++) {
4     sum[i]=0;
5     for(j=0; j<1000; j++)
6       sum[i] += a[i+j*4096];
7   }
```

Листинг 13. Буксование кэш-памяти при обработке нескольких массивов

```
1   double a[4096], b[4096], c[4096];
2   int i;
3   for(i=0; i<4096; i++)
4      c[i]=a[i]+b[i];
```

В примере из листинга 13 массивы размером 4096 элементов, т.е. 32 КБ, расположены в памяти последовательно друг за другом. Элементы этих массивов с одинаковыми индексами отстоят друг от друга в памяти как раз на размер массива, т.е. на 32 КБ. Значит, в приведенном цикле будут обращения к 3-м элементам, отстоящим на 32 КВ, что приведет к «буксованию» 2-ассоциативной кэш-памяти (Рис. 35).

Для того, чтобы избежать эффекта «буксования» кэш-памяти, обычно используются два приёма:

1. изменение порядка обхода элементов;
2. изменение расстояния между элементами.

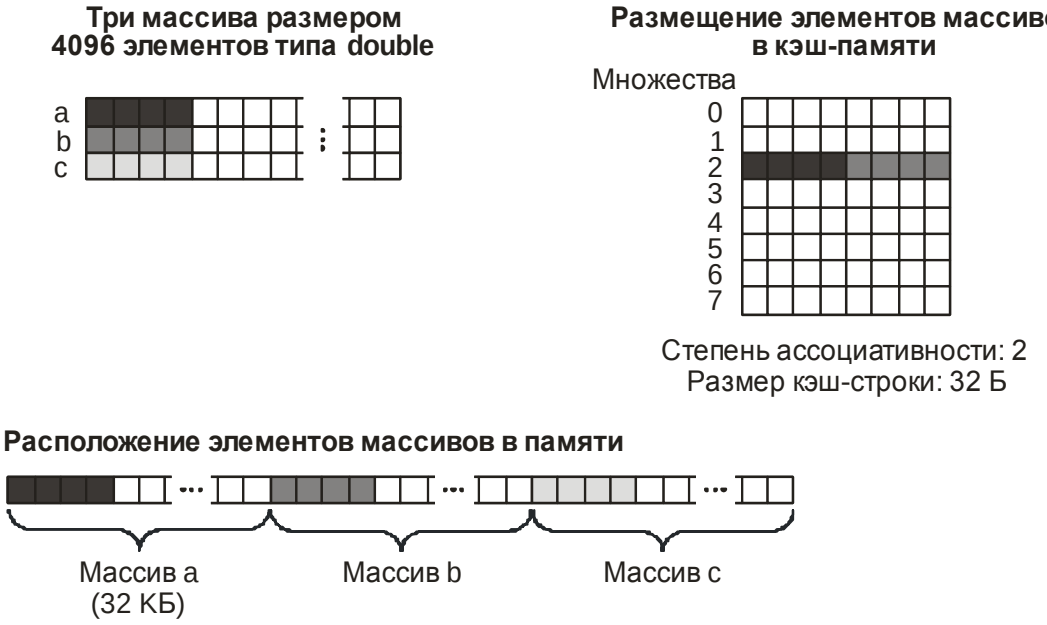

Рис. 35. Эффект кэш-буксования при работе с одномерными массивами

В ряде случаев избавиться от «буксования» кэш-памяти можно, изменив порядок обработки элементов массива. Способы изменения обхода массива будут рассмотрены в разделе 5.2.5.

В некоторых случаях порядок обработки данных изменить нельзя, т.к. он зафиксирован алгоритмом. Тогда единственным вариантом улучшения является изменение размещения данных в оперативной памяти, а именно, расстояния между конфликтующими элементами. Если такие элементы находятся в разных массивах, то можно изменить расстояние между ними, не изменяя сами массивы. Например, изменения расстояния между массивами `a`, `b` и `c` из листинга 13 можно добиться так, как показано в листинге 14.

Листинг 14. Изменение расположения массивов в памяти

```
1    double a[4096], tmp1[8], b[4096], tmp2[8], c[4096];
```

С тем же эффектом можно увеличить размер массива фиктивными (неиспользующимися) элементами, например, до 4096+8 (Рис. 36). Заметим, что для наилучшего результата величина прироста размера массива должна быть не меньше размера кэш-строки. Если добавленные элементы занимают

значительный объем памяти, то их можно использовать под хранение других данных задачи.

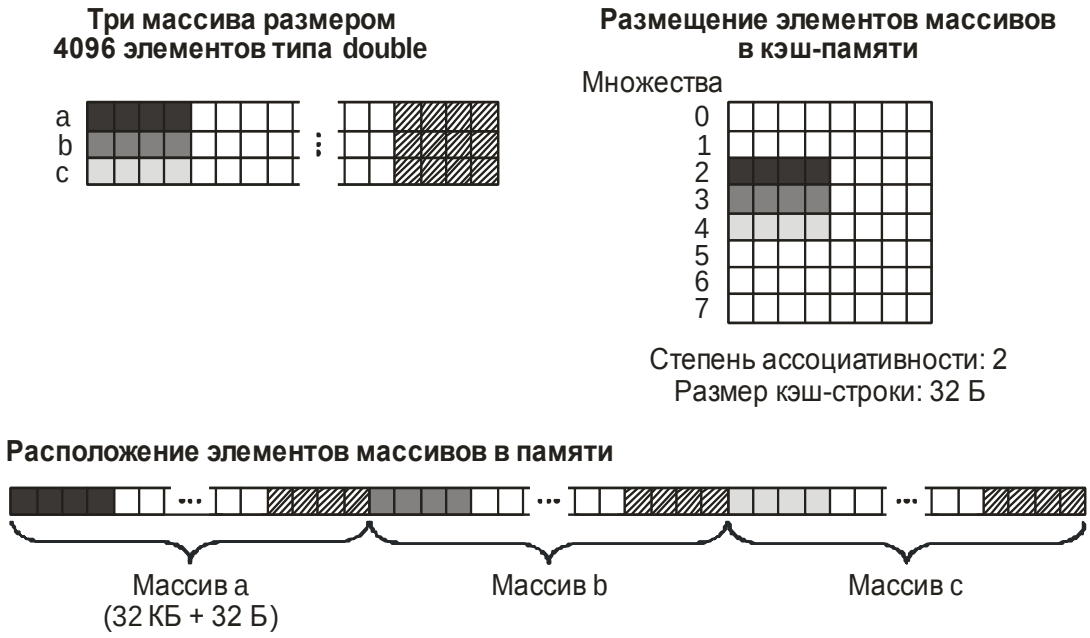

Рис. 36. Изменение расположения массивов в памяти для устранения эффекта кэш-буксования

5.2.4. Нарушение локальности во времени обращений в память

В программах довольно часто один и тот же блок памяти многократно загружается в кэш-память. Такая ситуация возникает, если к тому времени, как блок оперативной памяти понадобился снова, он уже был вытеснен из кэш-памяти другими блоками. Часто это свидетельствует о неэффективном использовании кэш-памяти, ведь значительно лучше было бы загрузить переменную в кэш-память лишь один раз и выполнить над ней все необходимые вычисления. Количество кэш-промахов при этом сократится, а значит и программа ускорится.

Для решения этой проблемы программу перестраивают таким образом, чтобы если некоторая переменная была загружена из оперативной памяти в кэш-память, то над ней выполняются все возможные вычисления, пока она ещё находится в кэш-памяти. Этот приём реализуется в блочном алгоритме умножения матриц (листинг 16).

Листинг 15. Обычный алгоритм умножения матриц

```
1    for (i=0;i<n;i++)
2     for (j=0;j<n;j++)
3      for (k=0;k<n;k++)
4       c[i][k] += a[i][j]*b[j][k];
```

Листинг 16. Блочный алгоритм умножения матриц

```
1    for (ii=0;ii<n;ii=ii+nb)
2     for (jj=0;jj<n;jj=jj+nb)
3      for (kk=0;kk<n;kk=kk+nb)
4       for (i=ii;i<ii+nb;i++)
5        for (j=jj;j<jj+nb;j++)
6         for (k=kk;k<kk+nb;k++)
7           c[i][k] += a[i][j]*b[j][k];
```

В блочном алгоритме тот же набор операций, что и в обычном алгоритме (листинг 15), но они сгруппированы таким образом, как если бы матрицы `a` и `b` были разделены на блоки (подматрицы), и умножение производится над блоками последовательно (листинг 16). Размеры блоков выбираются таким образом, чтобы в кэш-памяти одновременно могли поместиться данные обоих перемножаемых блоков и блок частичного результата. Как следствие, до окончания перемножения блоков данные будут находиться в кэш-памяти, что положительно скажется на времени выполнения программы.

5.2.5. Неупорядоченный обход данных в памяти.

Неупорядоченный доступ в оперативную память по произвольным адресам является неэффективным, так как с высокой вероятностью запрашиваемых данных не окажется в кэш-памяти и произойдёт кэш-промах. Доступ же в память последовательно, в порядке возрастания (или убывания) адресов с фиксированным шагом является наиболее предпочтительным. Выгода такого обхода памяти обуславливается двумя основными причинами – блочной загрузкой и аппаратной предвыборкой данных в кэш-память.

Рассмотрим Рис. 37. При случайном обходе некоторого массива после обращения к некоторому элементу содержащий его блок памяти загружается в кэш-память и находится там некоторое время. Некоторое время спустя этот блок вытесняется из кэш-памяти другими, вследствие чего повторное обращение к элементам этого блока снова приводит к кэш-промахам. При последовательном же обходе памяти после первого кэш-промаха идет обращение к другим элементам того же блока. В это время блок ещё находится в кэш-памяти, поэтому кэш-промахов не происходит. Таким образом, уменьшается количество кэш-промахов, и повышается эффективность использования канала кэш-память – оперативная память.

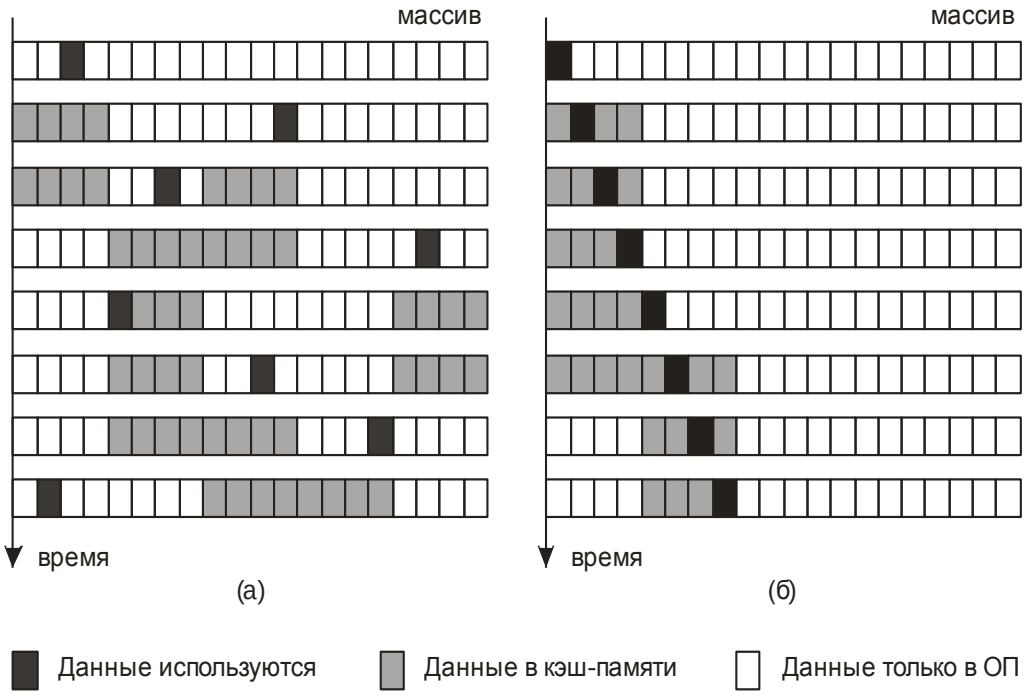

Рис. 37. Работа кэш-памяти при случайном (а) и последовательном (б) обходах памяти

На практике для повышения эффективности прикладных программ идея последовательного обхода оперативной памяти может быть использована следующим образом. Если требуется обработать элементы некоторого массива, то, по возможности, его элементы следует перебирать от первого элемента до

последнего элемента с шагом в один элемент. Неоднозначность возникает при обходе многомерных массивов. Очевидно, что порядок обхода определяется порядком вложенности циклов, перебирающих индексы массива.

По стандарту языка C/C++ массивы располагаются в памяти по строкам (Рис. 38). Соответственно, последовательный обход в языке C/C++ будет осуществляться, если порядок вложенности циклов установить от первого (левого) индекса к последнему (правому). Самый вложенный цикл – по последнему индексу (листинг 17).

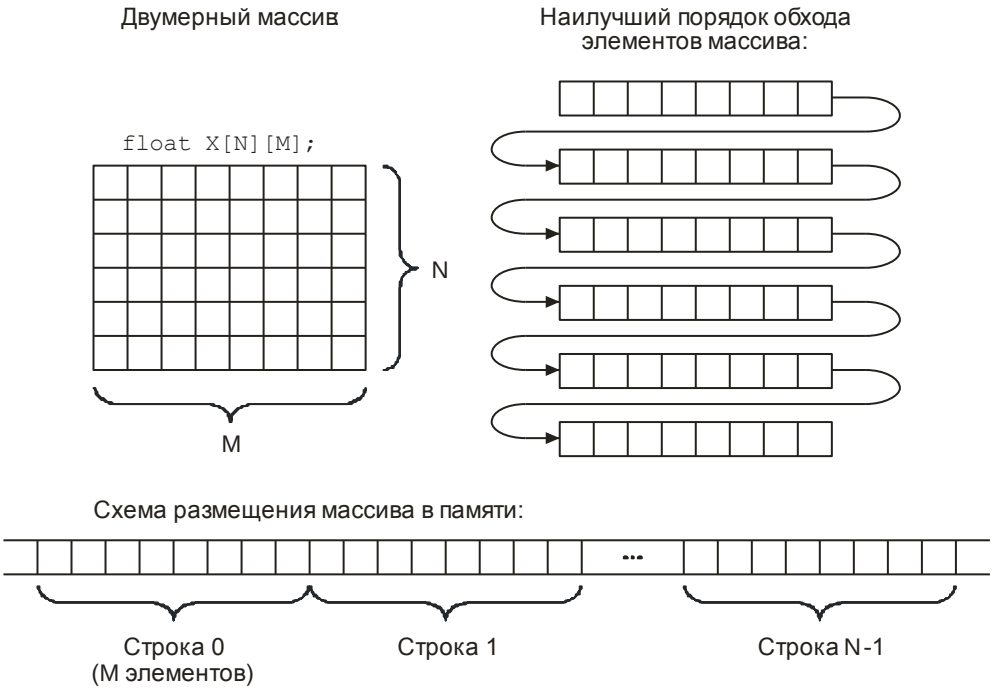

Рис. 38. Расположение в памяти двумерного массива

Листинг 17. Последовательный обход двумерного массива в языке C/C++

```
1    int X[N][M];
2    for (i=0;i<N;i++)
3      for (j=0;j<M;j++)
4        X[i][j]=1;
```

Приведем несколько вариантов действий, позволяющие достигнуть последовательного обхода памяти в программе, где таковой нарушается.

1. **Перестановка циклов.** Если алгоритм допускает изменение порядка вложенности циклов, то можно расположить их в том порядке, в котором будет происходить последовательный обход массива (Рис. 38). Это самый простой вариант, обычно требующий меньше всего изменений в исходном коде.
2. **Изменение порядка следования размерностей массива**. Если перестановка циклов невозможна вследствие особенностей алгоритма, то может помочь предварительное переупорядочение элементов массива таким образом, чтобы в циклах вычислений он обходился последовательно. Например, можно поменять порядок следования размерностей многомерного массива.

На Рис. 39 приведено время выполнения программы умножения плотных матриц в зависимости от порядка вложенности циклов. Как видно, при различных порядках обхода данных время может отличаться на порядок.

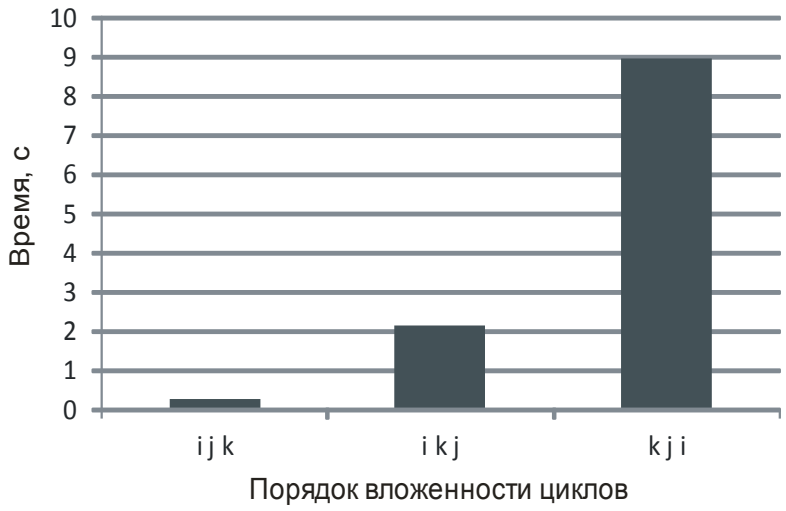

Рис. 39. Влияние порядка вложенности циклов на время выполнения программы умножения матриц

5.3. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. На каких этапах построения программы возможна ее оптимизация? В чем заключаются особенности оптимизации на каждом из этапов?
2. Зачем требуется делать выравнивание данных?

3. Как выбрать оптимальное расположение данных в памяти? Какие факторы это определяют?

4. Почему последовательный обход является наиболее оптимальным?

5. Каковы причины эффективности плотного размещения данных в памяти?

6. В каких ситуациях удобно устранять буксование кэш-памяти изменением порядка обхода элементов, а в каких – изменением их расположения в памяти?

ПРИЛОЖЕНИЕ 1. СРЕДСТВА РАЗРАБОТКИ ПРОГРАММ

1.1. ОПРЕДЕЛЕНИЕ ВРЕМЕНИ РАБОТЫ ПРИКЛАДНЫХ ПРОГРАММ

Цели практической работы:

1. Изучение методики измерения времени работы подпрограммы.
2. Изучение приемов повышения точности измерения времени работы подпрограммы.
3. Изучение способов измерения времени работы подпрограммы.
4. Получение практических навыков измерения времени работы программы.

Задание к практической работе

1. Написать программу на языке C или C++, которая реализует выбранный алгоритм из задания.
2. Проверить правильность работы программы на нескольких тестовых наборах входных данных.
3. Выбрать значение параметра N таким, чтобы время работы программы было порядка 15 секунд.
4. По приведенной в разделе 4.3.1 методике определить время работы подпрограммы тестовой программы с относительной погрешностью не менее 1%.
5. Составить отчет по работе. Отчет должен содержать следующие пункты.
    - Титульный лист.

- Цель работы.
- Вариант задания.
- Результат измерения времени работы программы.
- Полный компилируемый листинг реализованной программы и команду для ее компиляции.
- Вывод по результатам работы.

Варианты заданий

1. Алгоритм вычисления числа Пи с помощью разложения в ряд по N первых членов ряда:

$$\pi = 4\sum_{n=0}^{\infty}\frac{(-1)^n}{2n+1} = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \ldots + 4\frac{(-1)^n}{(2n+1)} + \ldots \qquad (2)$$

2. Алгоритм вычисления числа Пи метом Монте-Карло. Алгоритм состоит в следующем. Сначала в квадрат с центром в начале координат и со стороной два вписывается круг с единичным радиусом. Затем в этом квадрате случайным образом с равномерным распределением генерируются N точек. Точка может попасть в окружность или нет (условие попадания $x^2 + y^2 \le 1$). Далее определяется число M точек, попавших в круг. При достаточно большом числе бросков N, по значениям M и N вычисляется число Пи:

$$\pi \approx \frac{4M}{N}. \qquad (3)$$

3. Алгоритм вычисления определенного интеграла сложной функции методом трапеций:

$$\int_a^b f(x)dx = \frac{b-a}{N}\sum_{k=0}^{N-1}\frac{f(x_k) + f(x_{k+1})}{2}, \qquad (4)$$

где $f(x) = e^x \sin x$, $a = 0$, $b = \pi$, N - число интервалов.

4. Алгоритм вычисления функции sin *x* с помощью разложения в ряд по первым N членам этого ряда:

$$\sin x = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots + \frac{(-1)^{n-1}}{(2n-1)!} x^{2n-1} + \ldots \quad (5)$$

Область сходимости ряда: $-\infty \le x \le \infty$.

5. Алгоритм вычисления функции e^x с помощью разложения в ряд по первым N членам этого ряда:

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = 1 + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^n}{n!} + \ldots \quad (6)$$

Область сходимости ряда: $-\infty \le x \le \infty$.

6. Алгоритм вычисления функции ln(1+*x*) с помощью разложения в ряд по первым N членам этого ряда:

$$\ln(1+x) = \sum_{n=1}^{\infty} (-1)^{n+1} \frac{x^n}{n} = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots + (-1)^{n+1} \frac{x^n}{n} + \ldots \quad (7)$$

Область сходимости ряда: $-1 < x \le 1$.

7. Алгоритм сортировки методом пузырька. Дан массив случайных чисел длины N. На первой итерации попарно упорядочиваются все соседние элементы; на второй – все элементы, кроме последнего элемента; на третьей – все элементы, кроме последнего элемента и предпоследнего элемента и т. п.

Контрольные вопросы

1. Назовите цели измерения времени.
2. В чем состоит методика измерения времени работы программы?
3. Назовите способы измерения времени работы программы. Перечислите их особенности.
4. Каким способом лучше измерять большие промежутки времени (порядка нескольких часов)?

5. Каким способом лучше измерять малые промежутки времени (порядка времени работы нескольких команд процессора)?
6. Перечислите возможные причины понижения точности измерения времени и способы их устранения.

1.2. ИЗУЧЕНИЕ ОПТИМИЗИРУЮЩЕГО КОМПИЛЯТОРА

Цели практической работы:

1. Изучение основных функций оптимизирующего компилятора и способов его использования.
2. Получение базовых навыков работы с компилятором GCC.
3. Исследование влияния оптимизирующих преобразований компилятора GCC на время выполнения программы.

Задание к практической работе

1. Написать программу на языке C или C++, которая реализует выбранный алгоритм из задания.
2. Проверить правильность работы программы на нескольких тестовых наборах входных данных.
3. Выбрать значение параметра N таким, чтобы время работы программы было примерно 30-60 секунд.
4. Программу скомпилировать компилятором GCC с ключами оптимизации `-O0`, `-O1`, `-O2`, `-O3`, `-Os`, `-Ofast`, `-Og`.
5. Программу скомпилировать компилятором GCC с ключами оптимизации `-O3` под две различные микроархитектуры в рамках архитектуры имеющегося процессора.
6. Для каждого из вариантов компиляции измерить время работы программы при нескольких значениях N.
7. Составить отчет по практической работе. Отчет должен содержать следующие пункты.

- Титульный лист.
- Цель практической работы.
- Вариант задания.
- Графики зависимости времени выполнения программы с указанными уровнями оптимизации от параметра N.
- Полный компилируемый листинг реализованной программы и команды для ее компиляции.
- Вывод по результатам практической работы.

Варианты заданий

Варианты заданий взять из практической работы «Определение времени работы прикладных программ».

Контрольные вопросы

1. Каковы основные функции оптимизирующего компилятора?
2. Приведите примеры характеристик программы, по которым осуществляется оптимизация?
3. Приведите примеры оптимизирующих преобразований кода. Каково их предназначение? В чем их суть?
4. Всегда ли увеличение уровня оптимизации позволяет уменьшить время работы программы?
5. Чем отличается общая оптимизация от оптимизации под архитектуру?
6. Какие имеются группы ключей в GCC?
7. Какие уровни оптимизации есть в GCC, и чем они характеризуются?

ПРИЛОЖЕНИЕ 2. ЭФФЕКТИВНОЕ ПРОГРАММИРОВАНИЕ

2.1. ВЛИЯНИЕ КЭШ-ПАМЯТИ НА ВРЕМЯ ОБРАБОТКИ МАССИВОВ

Цели практической работы:

1. Исследование зависимости времени доступа к данным в памяти от их объема.
2. Исследование зависимости времени доступа к данным в памяти от порядка их обхода.

Влияние кэш-памяти на скорость доступа к данным

Обработка массивов данных встречается в самых разных задачах – от численного моделирования до обработки мультимедиа-данных. Время, затрачиваемое на обработку массивов, как правило, составляет существенную долю от общего времени выполнения программы. Основными особенностями организации памяти, которые играют важную роль при обработке массивов данных, являются:

- наличие нескольких уровней памяти разного размера и скорости доступа;
- блочная загрузка данных в кэш-память из оперативной памяти;
- аппаратная предвыборка данных в кэш-память.

Влияние уровней кэш-памяти

Иерархия памяти включает несколько уровней кэш-памяти разного размера и с разным временем доступа. Допустим, некоторая программа производит многократную обработку элементов массива. Известно, что зависимость среднего времени доступа к одному элементу массива от размера массива имеет нелинейный характер. При малых размерах массива, когда все данные умещаются в кэш-памяти первого уровня, среднее время доступа к элементу будет наименьшим, и оно не будет меняться при увеличении размера массива. Если размер массива превысит размер кэш-памяти первого уровня, то весь массив целиком уже не сможет в нем разместиться. Поэтому при обращении к некоторым элементам массива в кэш-памяти первого уровня будут случаться кэш-промахи, и элементы будут загружаться из кэш-памяти второго уровня (или оперативной памяти). Чем больше кэш-промахов происходит, тем больше будет среднее время доступа к элементу, вплоть до времени доступа к следующему уровню иерархии

памяти. В результате, с увеличением размера массива среднее время доступа к элементу будет ступенчато возрастать. Анализ графика зависимости времени от размера массива может показать, каковы объемы различных уровней кэш-памяти, имеющихся в системе.

Влияние блочной передачи данных

Данные из оперативной памяти в кэш-память считываются целыми блоками, которые затем размещаются в строках кэш-памяти. Размер блока равен одной или нескольким кэш-строкам. Если элементы в массиве обрабатываются последовательно один за другим, то попытка чтения первого элемента блока вызывает сравнительно долгое копирование всего блока из оперативной памяти в кэш-память. Чтение нескольких последующих элементов того же блока выполняется намного быстрее, так как они уже находятся в быстрой кэш-памяти.

Влияние аппаратной предвыборки данных

В большинстве современных микропроцессоров реализована аппаратная предвыборка данных. Она устроена таким образом, что при последовательном обходе очередные данные считываются из оперативной памяти еще до того, как к ним произошло обращение. Кэш-контроллеры с высокой вероятностью распознают последовательный обход памяти и обеспечивают эффективную предварительную загрузку данных в кэш-память. Как следствие, вероятность кэш-промахов значительно снижается. Если же элементы массива обрабатываются в более сложном порядке, то либо кэш-контроллер его не распознает, и тогда аппаратная предвыборка работать не будет, либо может распознать неправильно, что повлечёт вытеснение ещё нужных данных из кэш-памяти и увеличение среднего времени доступа к элементу массива.

Исследование влияния кэш-памяти

Для изучения влияния кэш-памяти на скорость доступа к данным в данной практической работе требуется построить программу, выполняющую обработку данных различного размера тремя характерными способами: последовательно в

сторону увеличения адресов, последовательно в сторону уменьшения адресов и в случайном порядке. Исследование времени выполнения программы в зависимости от порядка обхода и объема обрабатываемых данных позволит пронаблюдать влияние кэш-памяти.

Перебор размеров массива

Для определения среднего времени доступа к данным, программа должна многократно выполнять чтение элементов массива заданного размера (N) в заданном порядке. Интересующий нас диапазон изменения размеров массива определяется следующими границами:

- минимальное значение N_{min} выбирается заведомо меньше, чем размер кэш-памяти данных 1-го уровня (например, 1 Кбайт – 256 элементов типа `int`).
- максимальное значение N_{max} выбирается заведомо больше, чем размер кэш-памяти последнего уровня (например, 32 МБайта).

Программа должна перебирать размеры массива от N_{min} до N_{max}, для каждого конкретного N определяя среднее время доступа к элементу массива. Шаг изменения N должен быть достаточно маленьким, чтобы увидеть все существенные изгибы на графике, и достаточно большим, чтобы время выполнения теста не было слишком большим. Можно использовать переменный шаг, который растет вместе с возрастанием N.

Выполнение обхода

Для каждого конкретного размера массива N программа должна выполнить многократное чтение элементов массива в заданном порядке. Чтобы во время обхода не тратить время на вычисление индекса каждого следующего элемента, используется следующий приём. Значения элементов массива заполняются таким образом, чтобы сформировать односвязный циклический список, в котором значение очередного элемента представляет собой номер следующего. Обход тогда может быть выполнен циклом следующего вида:

```
for (k=0, i=0; i<N*K; i++) k = x[k];
```

Здесь `N` – размер массива, `K` – число обходов массива.

Таким образом, для каждого конкретного N программа должна выполнять следующие действия:

1. Заполнить значения элементов массива, чтобы они образовали циклический односвязный список в соответствии с требуемым порядком обхода.
2. Выполнить многократный обход массива. Замерить время обхода.

Способы обхода элементов массива

В рамках практической работы предлагается сравнить время доступа к данным для трех характерных способов обхода:

- прямой обход – в сторону увеличения адресов;
- обратный обход – в сторону уменьшения адресов;
- обход элементов в случайном порядке.

Для каждого способа обхода в программе необходимо заполнить значения элементов массива таким образом, чтобы они образовали односвязный циклический список, включающий все элементы массива (Рис. 40). Особое внимание следует обратить на заполнение массива для случайного обхода. Например, если в варианте для случайного массива, приведённом на Рис. 40в, поменять местами значения ячеек 5 и 6, то в обходе будут участвовать только элементы с номерами 0, 6 и 7. Необходимо, чтобы при обходе массива участвовали все элементы.

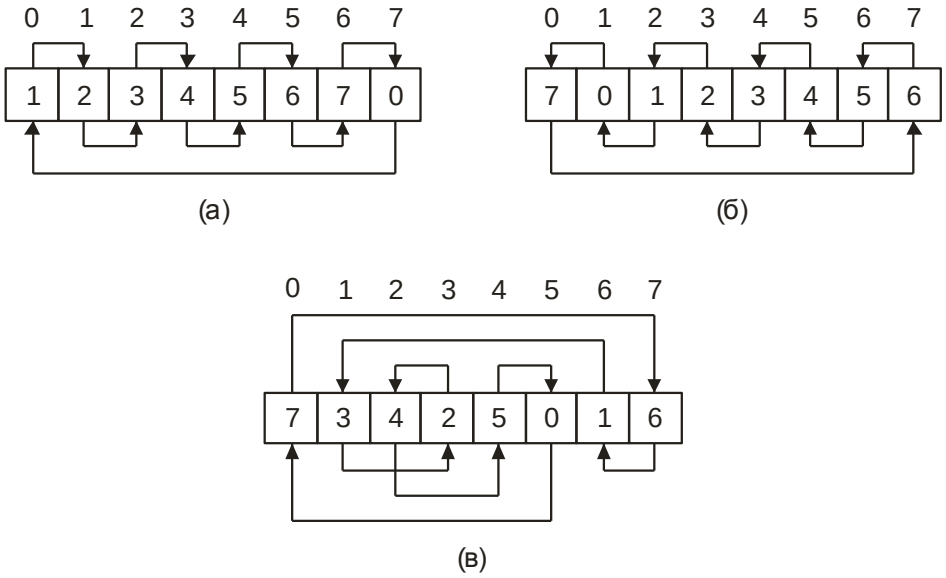

Рис. 40. Примеры способов заполнения элементов массива для выполнения обходов: а - прямого, б - обратного и в - в случайном порядке.

На Рис. 41 приведен пример графиков, полученных на процессоре Intel Xeon E5420 (L1: 32 KB, L2: 6 MB). Видна разница в скорости последовательного и случайного обходов массива. На соответствующих местах графика случайного обхода видно возрастание времени обращения к элементам.

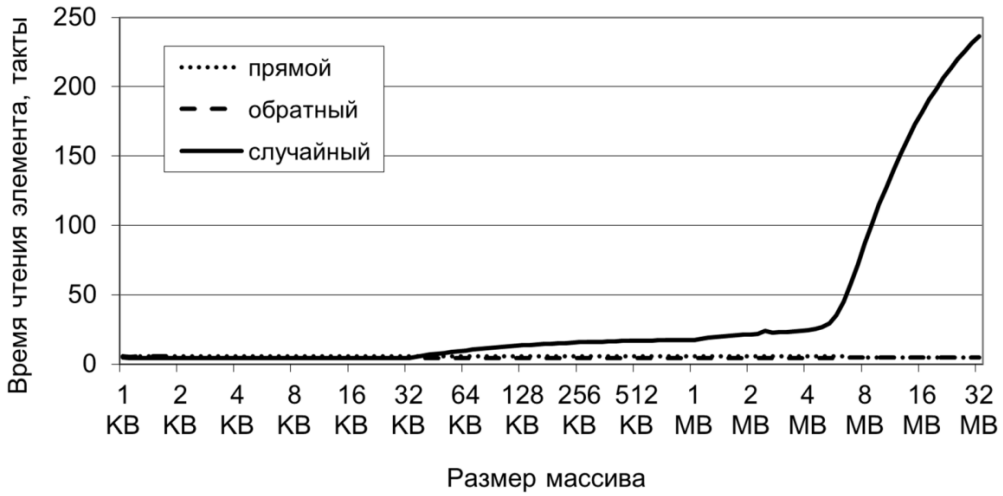

Рис. 41. Зависимость среднего времени чтения элемента массива от размера массива при различных способах обхода для процессора Intel Xeon E5420

Примечание 1. Компилировать программу нужно с ключом оптимизации –O1 чтобы исключить лишние обращения в память за счёт размещения переменных на регистрах.

Примечание 2. Перед измерением времени для каждого размера N необходимо осуществить однократный обход массива, чтобы «прогреть кэш-память», то есть выгрузить из кэш-памяти посторонние данные, разместив там (по возможности) необходимые нам данные.

Примечание 3. Результирующий график может оказаться сильно "замусоренным" высокими пиками. Такое бывает, если на машине работают посторонние "тяжелые" программы. В этом случае рекомендуется для каждого

конкретного размера массива и каждого конкретного способа обхода выполнить тест несколько раз, а в качестве результата взять минимальное время.

Задание к практической работе

1. Написать программу, многократно выполняющую обход массива заданного размера тремя способами.
2. Для каждого размера массива и способа обхода измерить среднее время доступа к одному элементу (в тактах процессора). Построить графики зависимости среднего времени доступа от размера массива.
3. На основе анализа полученных графиков:
   - определить размеры кэш-памяти различных уровней, обосновать ответ, сопоставить результат с известными реальными значениями;
   - определить размеры массива, при которых время доступа к элементу массива при случайном обходе больше, чем при прямом или обратном; объяснить причины этой разницы во временах.
4. Составить отчет по практической работе. Отчет должен содержать следующие пункты.
   - Титульный лист.
   - Цель практической работы.
   - Описание способа заполнения массива тремя способами.
   - Графики зависимости среднего времени доступа к одному элементу от размера массива и способов обхода.
   - Полный компилируемый листинг реализованной программы и команду для ее компиляции.
   - Вывод по результатам практической работы.

Контрольные вопросы

1. Что такое кэш-память? Какую проблему она решает?
2. Какой способ обхода данных в памяти самый быстрый? Почему?

3. Какой способ обхода данных в памяти самый медленный? Почему?
4. Для каких программ наличие кэш-памяти дает выигрыш во времени работы?
5. Назовите две основные причины, по которым случайный обход массива дольше, чем прямой и обратный.
6. Почему тип обхода не сказывается на времени доступа к памяти, если массив умещается в кэш-памяти первого уровня?
7. Чем отличаются программная и аппаратная предвыборка данных?

2.2. ИЗМЕРЕНИЕ СТЕПЕНИ АССОЦИАТИВНОСТИ КЭШ-ПАМЯТИ

Цели практической работы

1. Экспериментальное определение степени ассоциативности кэш-памяти.
2. Сравнение скорости работы подсистемы памяти при «буксовании» кэш-памяти и без него.

Организация теста

Для экспериментального определения степени ассоциативности кэш-памяти в программе организуется обход данных в памяти, который вызывает «буксование» кэш-памяти. Для этого предлагается организовать доступ в память по адресам, отображаемым на одно и то же множество в кэш-памяти. Будем увеличивать количество таких конфликтующих обращений. Когда оно превысит степень ассоциативности кэш-памяти, то данные начнут вытесняться из кэш-памяти, и возникнет кэш-буксование. Буксование кэш-памяти приводит к увеличению времени выполнения программы, благодаря чему можно определить степень ассоциативности кэш-памяти.

Например, для кэш-памяти, представленной на Рис. 42, размер банка составляет 4 КБ, а степень ассоциативности равна четырем. Таким образом, многократно выполнять обращения по пяти и более адресам, отстоящим друг от друга на расстояние, кратное 4 КБ, то мы получим эффект кэш-буксования.

Рассмотрим, как может быть организован обход памяти в соответствии с описанной идеей эксперимента. Будем выполнять обход нескольких фрагментов данных в памяти, отстоящих друг от друга на смещение, кратное размеру банка. Суммарный объем фрагментов данных возьмем равным размеру исследуемой кэш-памяти, предполагая, что для всех запрашиваемых данных в кэш-памяти должно найтись место.

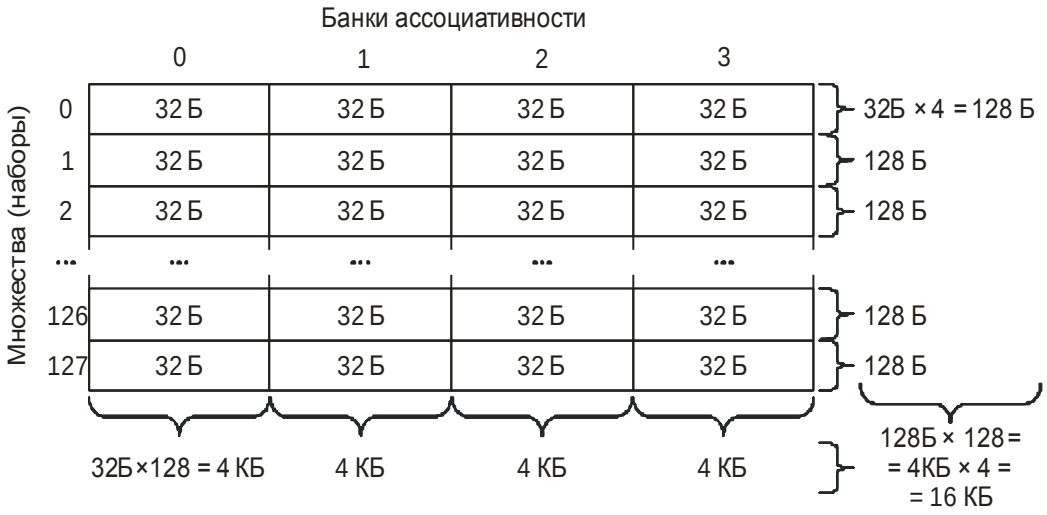

Рис. 42. Пример множественно-ассоциативной кэш-памяти

Чтобы адреса отображались на одно и то же множество в кэш-памяти, необходимо, чтобы смещение между ними было кратно размеру банка кэш-памяти. Так как размер банка кэш-памяти не всегда известен, то рассмотрим, как его можно оценить. Размер кэш-памяти кратен размеру банка, поэтому, если известен размер кэш-памяти, то расстояние между началами фрагментов можно взять равным размеру кэш-памяти. Если же размер кэш-памяти неизвестен, то можно использовать тот факт, что размеры банков кэш-памяти практически всегда являются степенью двойки. Для современных процессоров подходящим смещением между фрагментами будет 2^{24} Б = 16 МБ.

Обход фрагментов данных организуем таким образом, чтобы подряд происходили обращения к элементам разных фрагментов, отстоящим на заданное смещение. Сначала последовательно производится чтение всех первых элементов

во всех фрагментах, затем всех вторых, затем – всех третьих и т.д. На Рис. 43 показано размещение фрагментов данных в памяти (Рис. 43а) и порядок обхода элементов (Рис. 43б). Здесь Size – объем кэш-памяти, N – число фрагментов, Offset – смещение между началами соседних фрагментов.

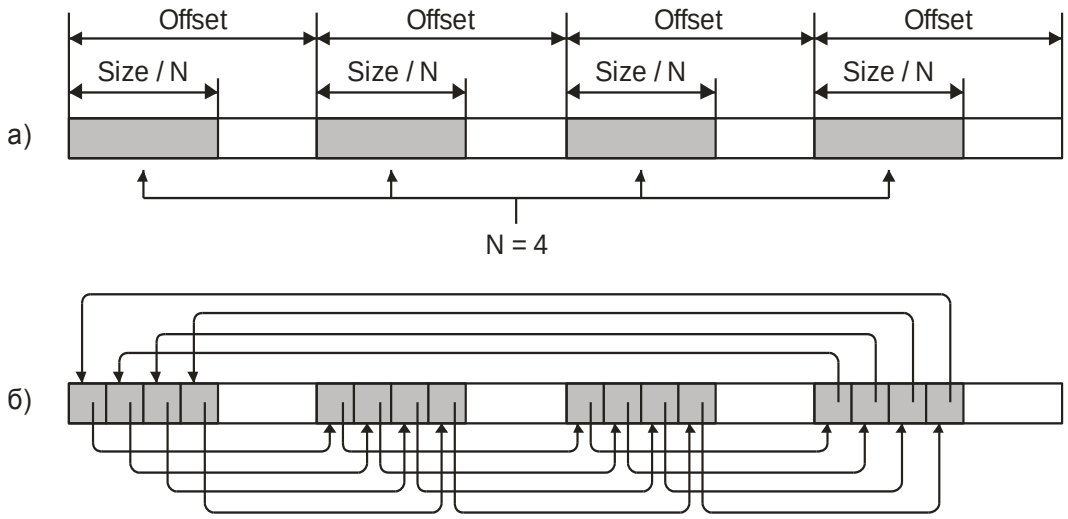

Рис. 43. Представление фрагментов данных в оперативной памяти и их обход: а - схема расположения фрагментов данных для обхода,
б - порядок обхода элементов

Для проведения эксперимента предлагается выделить массив достаточно большого размера, чтобы вместить все необходимые фрагменты с требуемым смещением. Элементы массива образуют связанный список, где значение каждого элемента является индексом следующего (как в предыдущей практической работе).

На Рис. 44 приведен график зависимости времени чтения фрагментов от их числа. График построен для процессора Intel Xeon X5660 (степень ассоциативности кэш-памяти 1-го и 2-го уровня: 8, степень ассоциативности кэш-памяти 3-го уровня: 16). На графике видно замедление после 8-ми фрагментов. Это соответствует степени ассоциативности кэш-памяти 1-го и 2-го уровня. Второе замедление происходит после 16-ти фрагментов. Оно соответствует степени ассоциативности кэш-памяти 3-го уровня. Увеличение времени после 4-х

фрагментов тоже соответствует степени ассоциативности, но уже не кэш-памяти, а TLB.

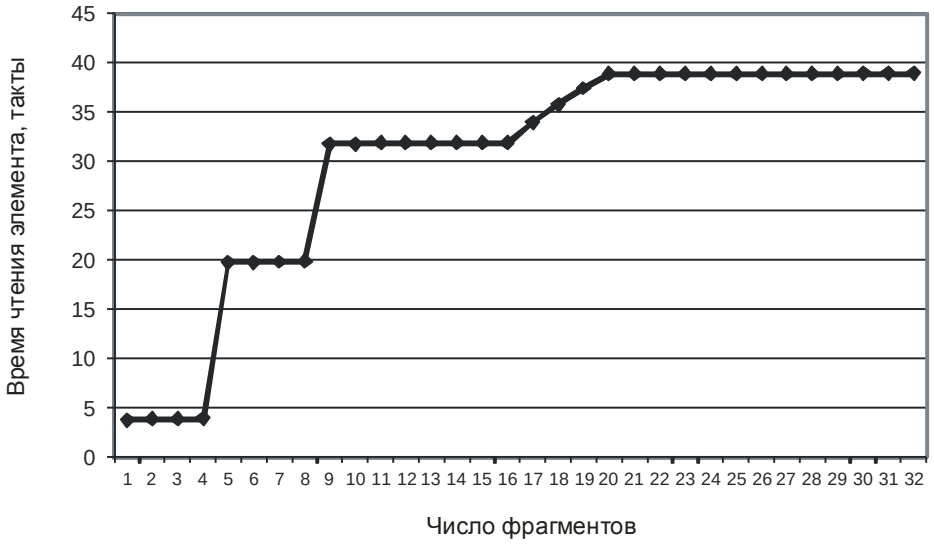

Рис. 44. Среднее время чтения одного элемента массива в зависимости от числа фрагментов для процессора Intel Xeon X5660

Задание к практической работе

1. Написать программу, выполняющую обход памяти в соответствии с заданием.
2. Измерить среднее время доступа к одному элементу массива (в тактах процессора) для разного числа фрагментов: от 1 до 32. Построить график зависимости времени от числа фрагментов.
3. По полученному графику определить степень ассоциативности кэш-памяти, сравнить с реальными характеристиками исследуемого процессора.
4. Определить, во сколько раз увеличилось время доступа к элементам массива при кэш-буксовании.
5. Составить отчет по практической работе. Отчет должен содержать следующее.
    - Титульный лист.

- Цель практической работы.
- Параметры теста: размер фрагментов, величина смещения.
- График зависимости среднего времени доступа к элементу массива от числа фрагментов.
- Реальные и полученные в ходе тестирования значения степени ассоциативности кэш-памятей процессора.
- Результат сравнения времени доступа к элементам массива при кэш-буксовании и без него.
- Полный компилируемый листинг реализованной программы и команды для ее компиляции.
- Вывод по результатам практической работы.

Контрольные вопросы

1. Что такое степень ассоциативности кэш-памяти?
2. Назовите достоинства и недостатки множественно-ассоциативной кэш-памяти?
3. Что такое эффект кэш-буксования? Как его вызвать? Как его избежать?
4. Какое минимальное число элементов требуется, чтобы организовать эффект «буксования» кэш-памяти (при «наихудшем» размещении этих элементов в оперативной памяти)?
5. Какой график получится в результате тестирования, если величину смещения между фрагментами взять равной половине размера банка?

2.3. УСТРАНЕНИЕ БУКСОВАНИЯ КЭШ-ПАМЯТИ

Цели практической работы:

1. Исследование влияния эффекта буксования кэш-памяти на время выполнения программы.
2. Освоение способов устранения буксования кэш-памяти.

Буксование кэш-памяти при обработке массивов

Рассмотрим пример программы заполнения элементов массива, которая приводит к буксованию кэш-памяти (листинг 18). Размеры массива в программе установлены в расчете на кэш-память размером 32 КБ (Рис. 14).

Листинг 18. Пример программы, вызывающей эффект буксования кэш-памяти

```
1   #define N (32768 / sizeof(int))
2   int array[N][N];
3   main(){
4     for (i=0; i<N; i++)
5       for (j=0; j<N; j++)
6         array[j][i] = i*j;
7   }
```

При некоторых размерах массива array в процессе выполнения данной программы будет возникать буксование кэш-памяти. Эмпирическое определение таких размеров основано на переборе различных значений параметра N, измерении времени выполнения программы и поиске таких размеров, при которых время существенно возрастает по сравнению со временем выполнения для близких размеров.

Способы устранения буксования кэш-памяти

Две основных группы способов устранения кэш-буксования основаны:

- на смене представления обрабатываемых данных в оперативной памяти и
- изменении порядка доступа к ячейкам памяти.

Первый способ применительно к двумерному массиву заключается в увеличении каждой строки массива на несколько элементов. В листинге 19 показана соответствующая модификация программы листинга 18. Общий размер добавляемых к строке элементов должен быть равным или превышать размер строки кэш-памяти. В таком случае ячейки из одной колонки массива

располагаются в блоках памяти, которые проецируются на разные множества кэш-памяти.

Листинг 19. Устранение эффекта буксования кэш-памяти, способ 1

```
1   #define N (32768 / sizeof(int))
2   int array[N][N + 64 / sizeof(int)];
3   main(){
4     for (i=0; i<N; i++)
5       for (j=0; j<N; j++)
6         array[j][i] = i*j;
7   }
```

Примером второго способа устранения является перестановка циклов. Соответствующий модифицированный пример программы приведен в листинге 20.

Листинг 20. Устранение эффекта буксования кэш-памяти, способ 2

```
1   #define N (32768 / sizeof(int))
2   int array[N][N];
3   main(){
4     for (j=0; i<N; i++)
5       for (i=0; j<N; j++)
6         array[j][i] = i*j;
7   }
```

Задание к практической работе

1. Написать программу, реализующую алгоритм из выбранного варианта заданий.
2. Определить значение параметра N, при котором в программе происходит буксование кэш-памяти.
3. Модифицировать программу из п. 1 с целью устранить «буксование» кэш-памяти, изменив представление массива в памяти путем добавления элементов.
4. Модифицировать программу из п. 1 с целью устранить «буксование» кэш-памяти, изменив обход элементов по столбцам на обход по строкам.

5. Для определенного в п. 2 значения параметра N и при нескольких других значениях в его окрестности измерить времена выполнения трех программ из пп. 1, 3 и 4.
6. На основе проведенных измерений сделать выводы об эффективности каждого из способов устранения буксования кэш-памяти.
7. Составить отчет по практической работе. Отчет должен содержать следующие пункты.
    - Титульный лист.
    - Цель практической работы.
    - График зависимости времени выполнения исходной программы от размера массива в окрестности размера, при котором наблюдается буксование кэш-памяти.
    - Полный компилируемый листинг одной исходной и двух модифицированных программ и команды для их компиляции.
    - Вывод по результатам практической работы.

Варианты заданий

1. Алгоритм умножения двух квадратных матриц размером $N \times N$: $C=A \times B$, где элемент матрицы C вычисляется по формуле: $c_{i,j} = \sum_{k=1}^{N} a_{i,k} b_{k,j}$. Обход матрицы A должен производиться по строкам, а обход матрицы B – по столбцам. Тип элементов матриц: 8-байтовый вещественный.
2. Алгоритм итеративного пересчета элементов двумерного массива размером N×N. Тип элементов: 4-байтовый целочисленный. На каждой новой итерации вычисляются новые значения элементов массива, используя значения элементов массива на предыдущей итерации. Новое значение элемента массива $q_{i,j}$ вычисляется из старых значений $p_{i,j}$ по формуле:

$$q_{i,j} = \frac{1}{49}\left[\sum_{k=-12}^{12}\left(p_{i+k,j} + p_{i,j+k}\right)\right] - p_{i,j} \qquad (8)$$

Все элементы массива, значения которых используются при подсчете нового значения данного элемента, будем называть окрестностью этого элемента. Если у некоторого элемента окрестность выходит за края массива, то вычисления для него не проводить. Для реализации алгоритма требуются два массива одинакового размера. Исходные значения всех четных итераций (включая нулевую) считываются из первого массива, а новые значения записываются во второй массив. На всех нечетных итерациях исходные значения считываются из второго массива, а записываются – в первый. Предлагается использовать следующие параметры алгоритма: размер массива взять такой, при котором размер одной строки будет превышать размер кэш-памяти 1-го уровня (например, N=8192), а число итераций взять порядка 10-20.

Контрольные вопросы

1. В каких случаях может возникать может возникать буксование кэш-памяти при обработке массивов? Приведите пример.
2. Как экспериментально обнаружить буксование кэш-памяти?
3. Какими способами можно устранить буксование кэш-памяти?

2.4. СОВМЕСТНЫЙ ДОСТУП НЕСКОЛЬКИХ ПОТОКОВ К ДАННЫМ ОБЩЕЙ ПАМЯТИ

Цели практической работы:

1. Знакомство с организацией согласованности данных в многоядерных компьютерах с общей памятью.
2. Исследование особенностей доступа нескольких параллельно работающих потоков к данным в общей памяти.

Организация согласованности данных в многоядерных компьютерах с общей памятью

Кэш-память хранит копии блоков данных из оперативной памяти. В параллельных вычислительных системах с общей памятью (например, в многоядерных процессорах) у каждого ядра может быть своя кэш-память, между которыми, как правило, поддерживается согласованность (когерентность). Так, если копии одного и того же блока данных содержатся в кэш-памятях разных процессорных ядер, то при изменении одной из копий остальные либо тоже изменяются, либо становятся «недействительными». Когерентность поддерживается с помощью специального протокола поддержки когерентности. Протокол поддержки когерентности – это система состояний блоков данных (измененный, эксклюзивный, недействительный, …) и правил изменения состояний в результате различных событий (чтение, запись, …). Блоком данных, с которым работает протокол, обычно является кэш-строка.

Рассмотрим пример работы протокола MESI (по первым буквам состояний кэш-строк: Modified, Exclusive, Shared, Invalid). Когда некоторый блок данных оперативной памяти запрашивается на чтение одним из ядер, тогда он попадает в кэш-память этого ядра, в некоторую кэш-строку. Если копий этого блока данных нет ни в какой другой кэш-памяти, то кэш-строка получает состояние **Exclusive** (эксклюзивный). Когда затем этот же блок данных будет запрошен на чтение другим ядром, тогда его копия будет помещена в кэш-память и того ядра. При этом состояние обеих кэш-строк, хранящих копии одного и того же блока данных, будут изменены на **Shared** (разделяемый). Если теперь одно из этих ядер совершит запись в рассматриваемый блок данных, то соответствующая кэш-строка этого ядра получит состояние **Modified** (измененный), а кэш-строки всех других ядер, хранящих копии того же блока данных, получат состояние **Invalid** (недействительный). Таким образом, если ядро с недействительной кэш-строкой снова запросит данные по тому же адресу, то данные снова должны быть переданы из оперативной памяти или из кэш-памяти других ядер.

Передача данных между кэш-памятью различных ядер занимает некоторое время. Это время обычно меньше времени доступа в оперативную память, но больше времени доступа в свою кэш-память. Если несколько ядер будут

одновременно многократно модифицировать одну и ту же ячейку оперативной памяти, то актуальное содержимое этой ячейки будет постоянно передаваться от одного ядра к другому и обратно, т. е. скорость их работы будет меньше, чем, если бы каждое ядро работало со своей отдельной ячейкой оперативной памяти.

При создании многопоточных программ встречается ситуация, когда несколько потоков работают с различными ячейками памяти, но скорость работы такая же, как при работе с одной и той же ячейкой. Эта ситуация называется ложное разделение кэш-строк (false sharing) и заключается в том, что различные ячейки данных, с которыми работают потоки, находятся на одной кэш-строке. Как следствие, для протокола когерентности эти ячейки неразличимы. На Рис. 45 приведена зависимость времени однократной модификации ячейки памяти при одновременной обработке двух ячеек памяти двумя разными потоками от смещения одной ячейки относительно другой. Она получена на двухядерном процессоре AMD Athlon64 X2 3600+ и демонстрирует ситуацию ложного разделения кэш-строк. На диаграмме приведена зависимость времени однократной модификации ячейки памяти при одновременной обработке двух ячеек памяти двумя разными потоками от смещения в памяти одной ячейки относительно другой. Видно, что единицей поддержания когерентности является блок размером 64 байта – одна кэш-строка.

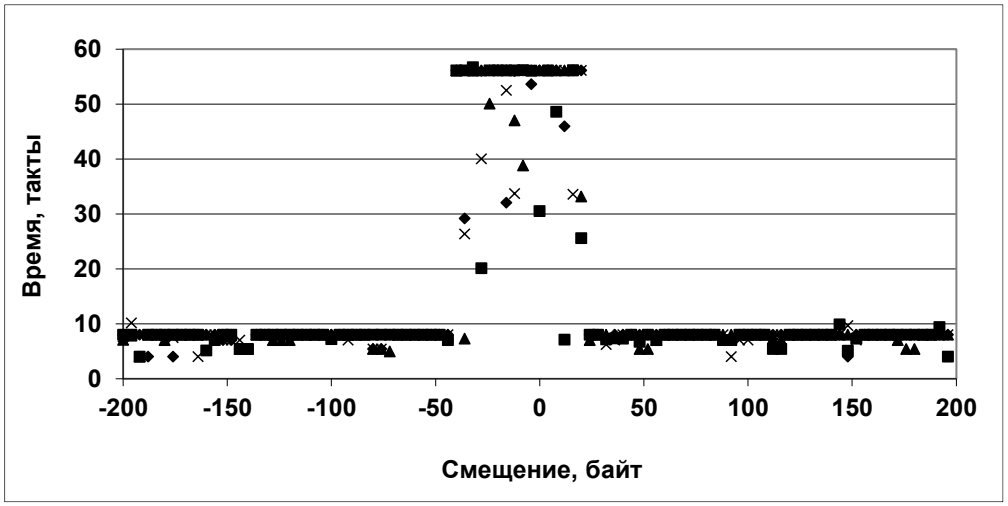

Рис. 45. Зависимость времени однократной модификации ячейки памяти при одновременной обработке двух ячеек памяти двумя разными потоками от смещения одной ячейки относительно другой

Решением проблемы ложного разделения кэш-строк является разнесение ячеек, к которым требуется одновременный доступ из нескольких потоков, на расстояние большее, чем кэш-строка.

Организация теста

Идея теста заключается в том, чтобы сравнить скорость работы подсистемы памяти при совместном доступе в память двух ядер в случае, когда разделение данных имеет место (ложное или истинное), и когда разделения нет. Для этого организуется обработка элементов массива двумя ядрами. Первое ядро всегда обрабатывает один и тот же элемент массива, а второе ядро обрабатывает различные элементы того же массива. Когда элементы, обрабатываемые двумя ядрами, оказываются в одном и том же блоке памяти, то возникает ситуация ложного разделения данных, что сказывается увеличением времени обработки одного элемента массива.

Создание многопоточной программы с помощью OpenMP

Параллельные потоки можно организовать с помощью средства параллельного программирования OpenMP. Следующая конструкция в последовательной программе на языке C/C++ позволяет создать два независимых потока (листинг 21).

Листинг 21. Шаблон многопоточной программы в OpenMP

```
01  #pragma omp parallel num_threads(2) private(i,j,k)
02  {
03    // <код, выполняемый всеми потоками>
04    #pragma omp sections
05    {
06      #pragma omp section
```

```
07      {
08        // <код 1-го потока>
09      }
10      #pragma omp section
11      {
12        // <код 2-го потока>
13      }
14    }
15    // <код, выполняемый всеми потоками >
16  }
```

Здесь директива `#pragma omp parallel` задает в последовательной программе параллельную секцию, внутри которой работают несколько потоков. Параметр `num_threads(2)` устанавливает число потоков равное двум. Параметр `private(…)` задает список переменных, которые должны быть индивидуальными для каждого потока (для каждого параллельного потока создается отдельная копия переменной). Все остальные данные будут общими для всех потоков. Директива `#pragma omp sections` задает разделение работ по потокам.

Задание к практической работе

1. Написать программу, осуществляющую одновременный доступ двух потоков к элементам одного массива длиной N элементов целочисленного типа (4 байта). Первый поток всегда обрабатывает элемент с номером N/2. Второй поток обрабатывает i-й элемент, где индекс i пробегает по всем элементам массива. Под обработкой элемента массива будем понимать M-кратную операцию его инкремента.
2. Построить график зависимости времени одной операции инкремента во втором потоке от номера обрабатываемого элемента, при этом использовать замер времени с помощью подсчета числа тактов процессора.
3. По графику определить размер блока, который является единицей данных для протокола поддержания когерентности.
4. Составить отчет по практической работе. Отчет должен содержать:

- Титульный лист.
- Цель практической работы.
- Полный компилируемый листинг реализованной программы и команды для ее компиляции.
- График зависимости времени обработки элемента массива вторым потоком от его смещения относительно элемента, обрабатываемого первым потоком.
- Вывод по результатам практической работы.

Примечание 1. Параллельные потоки следует создать один раз в начале тестирования, чтобы избежать накладных расходов на их создание и удаление.

Примечание 2. Чтобы нивелировать возможное расхождение во времени запуска параллельных задач для каждого нового значения индекса, рекомендуется поступить следующим образом. Первый поток выполняет 3M операций инкремента, а второй поток сначала выполняет M операций инкремента, затем начинает замер времени, потом выполняет M операций инкремента и завершает замер времени. Рекомендуемые значения параметров тестирования: N = 100, M = 100000.

Примечание 3. Чтобы исключить влияние посторонних обращений в память, необходимо использовать хотя бы минимальный уровень оптимизации компилятором. Чтобы не допустить выкидывания оптимизирующим компилятором значимого кода, рекомендуется массив, к которому выполняются обращения, объявить с ключевым словом volatile (в языке Си). При необходимости можно также использовать другие способы борьбы с «излишней оптимизацией», например, фиктивное использование результатов вычислений.

Контрольные вопросы

1. Как поддерживается согласованность данных в кэш-памяти различных ядер многоядерных процессоров?
2. Что такое протокол когерентности?

3. Как несколько потоков, одновременно работающих на различных ядрах с различными данными в одном адресном пространстве, могут влиять на скорость работы друг друга?
4. Что такое ложное разделение кэш-строк? Как оно устраняется?

ПРИЛОЖЕНИЕ 3. ВВЕДЕНИЕ В АРХИТЕКТУРУ X86/X86-64

Цели практической работы:

1. Знакомство с программной архитектурой x86/x86-64.
2. Анализ ассемблерного листинга программы для архитектуры x86/x86-64.

Краткое описание архитектуры x86

Архитектура x86 в настоящий момент является самой распространенной программной архитектурой настольных и серверных вычислительных систем. Она используется с 1978 года, когда был выпущен 16-разрядный процессор i8086. Впоследствии архитектура была естественным образом расширена до 32-разрядной (IA-32), а затем и до 64-разрядной (x64, AMD64, x86-64, IA-32e, EM64T, Intel 64). При каждом очередном расширении сохранялась программная совместимость с предыдущим поколением архитектуры. При этом набор команд расширялся, а новые регистры большего размера включали в качестве своих частей регистры предыдущего поколения архитектуры.

Регистры. Программисту доступны следующие группы регистров:

- регистры общего назначения,
- регистры сопроцессора,
- регистры векторных расширений,
- регистр флагов,
- счетчик команд.

Регистры общего назначения используются для хранения произвольных целочисленных данных. В большинстве команд в качестве операндов эти регистры полностью взаимозаменяемы. На Рис. 46 представлены регистры

общего назначения архитектур x86 и x86-64. У каждого регистра отдельно доступна младшая часть размером 8, 16 или 32 бита. Например, младшая часть регистров Rx доступна по именам RxD (32 бита), RxW (16 бит) и RxB (8 бит), где x – число от 8 до 15.

Сопроцессор. Сопроцессор (FPU – Floating Point Unit) предназначен для выполнения операций над вещественными числами. С программной точки зрения сопроцессор содержит несколько управляющих регистров, а также блок из восьми регистров данных разрядностью 80 бит, организованных в стек (Рис. 47). Номер регистра, являющегося текущей вершиной стека, хранится в специальном поле регистра состояния (указателе вершины стека). Операция push уменьшает значение указателя на единицу и помещает данные в регистр, являющийся новой вершиной стека. Операция pop записывает данные с вершины стека в память или регистр и увеличивает указатель на единицу.

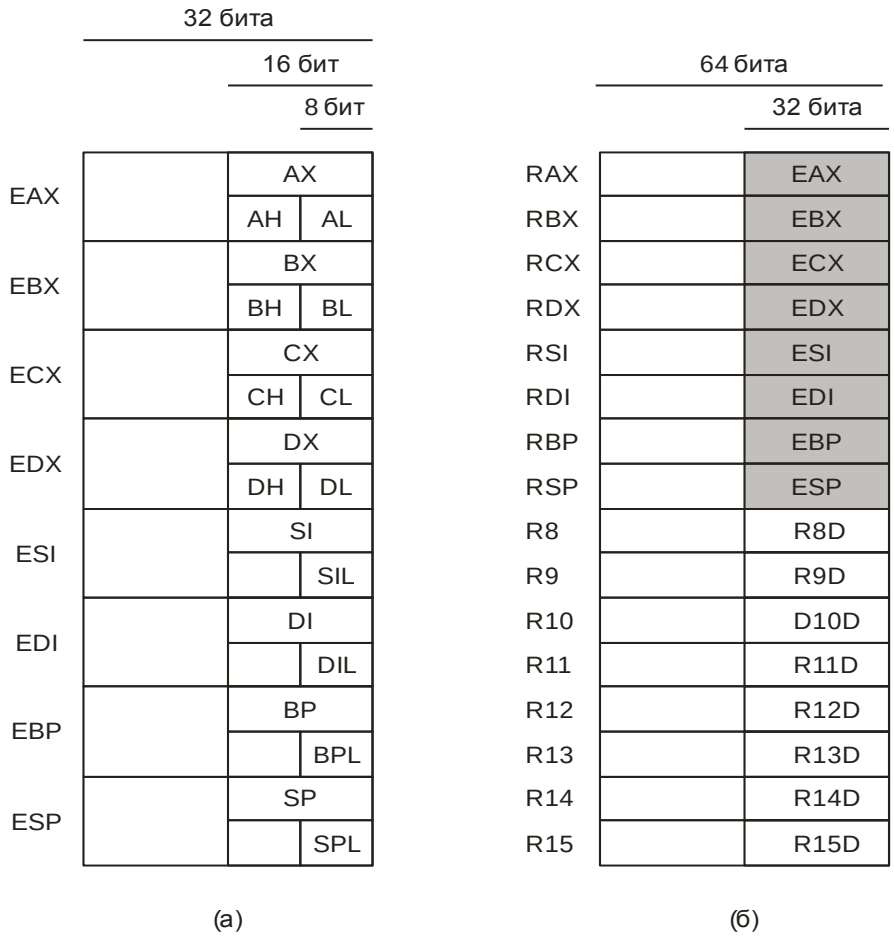

Рис. 46. Регистры общего назначения: а – регистры в архитектуре x86,
б – регистры в архитектуре x86-64

Инструкции сопроцессора адресуют регистры либо явно, либо неявно. Неявная адресация (без указания конкретного регистра) подразумевает использование регистров, находящихся на вершине стека. Явная адресация подразумевает указание смещения регистра относительно вершины стека. Например: st(0) регистр на вершине стека, st(1) – следующий за ним и т.д.

Векторные расширения. Векторные расширения реализуют технологию SIMD-вычислений. На Рис. 47 показаны регистры сопроцессора, MMX и 3DNow! Характеристики основных векторных расширений архитектуры x86/x86-64 представлены в таблице 2.

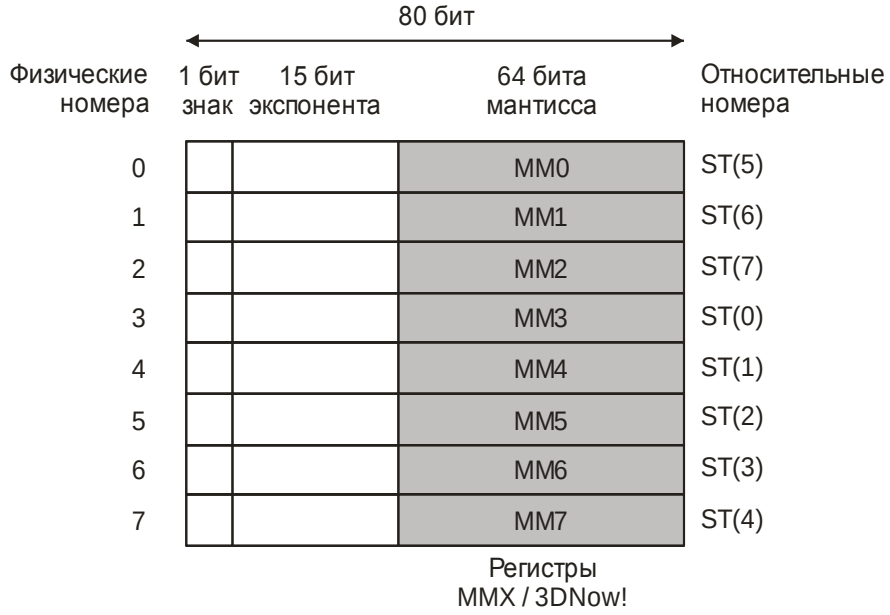

Рис. 47. Регистры сопроцессора, MMX и 3DNow!

Таблица 2

Название	Размер регистра, бит	Число регистров	Имена регистров	Типы элементов данных, размер в битах
MMX	64	8	mm0-mm7	цел.: 8,16,32,64
3DNow!	64	8	mm0-mm7	вещ.: 32

SSE, …	128	8 / 16	xmm0-xmm15	цел.: 8,16,32,64, вещ.: 32, 64
AVX	256	16	ymm0-ymm15	цел.: 8,16,32,64, вещ.: 32, 64

Задание к практической работе

1. Изучить программную архитектуру x86/x86-64:
    - набор регистров,
    - основные арифметико-логические команды,
    - способы адресации памяти,
    - способы передачи управления,
    - работу со стеком,
    - вызов подпрограмм,
    - передачу параметров в подпрограммы и возврат результатов,
    - работу с арифметическим сопроцессором,
    - работу с векторными расширениями.
2. Для программы на языке Си (см. варианты в приложении 1) сгенерировать ассемблерные листинги для архитектуры x86 и архитектуры x86-64, используя следующие уровни комплексной оптимизации компилятора: без оптимизации, минимальный уровень оптимизации, максимальный уровень оптимизации.
3. Проанализировать полученные листинги и сделать следующее.
    - Сопоставить команды языка Си с машинными командами.
    - Определить размещение переменных языка Си в программах на ассемблере (в каких регистрах, в каких ячейках памяти).
    - Описать и объяснить оптимизационные преобразования, выполненные компилятором.
    - Продемонстрировать использование ключевых особенностей архитектур x86 и x86-64 на конкретных участках ассемблерного кода.

- Сравнить различия в программах для архитектуры x86 и архитектуры x86-64.

4. Выполнить полученную программу под управлением отладчика GDB в режиме машинных кодов.
    - Запустить программу в режиме отладки.
    - Установить точку останова на начало вычислительного цикла.
    - Выполнить по шагам несколько итераций вычислительного цикла (внутреннего, если имеются вложенные циклы).
    - Распечатать и включить в отчет распечатку стека вызовов функций для произвольной команды внутри вычислительного цикла.
5. Составить отчет по практической работе. Отчет должен содержать следующее.
    - Титульный лист.
    - Цель практической работы.
    - Полный компилируемый листинг реализованной программы и команды для ее компиляции.
    - Листинг программы на ассемблере, сгенерированный компилятором.
    - Список оптимизационных преобразований, выполненных компилятором, со ссылками на соответствующие места в ассемблерных листингах.
    - Стек вызовов функций для произвольной команды внутри вычислительного цикла.
    - Вывод по результатам практической работы.

Варианты заданий

Варианты заданий взять из практической работы «Определение времени работы прикладных программ».

Контрольные вопросы

1. Назовите отличия архитектур x86 и x86-64.
2. Что означает обратная совместимость архитектуры x86-64 с архитектурой x86?
3. Почему в программах следует избегать лишних обращений в память?
4. Как зависимости между командами влияют на процесс выполнения программы? Как компилятор и процессор борются с ними?

ПРИЛОЖЕНИЕ 4. ВЕКТОРИЗАЦИЯ ВЫЧИСЛЕНИЙ

Цели практической работы:

1. Изучение SIMD–расширений архитектуры x86/x86-64.
2. Изучение способов использования SIMD–расширений в программах на языке Си.
3. Получение навыков использования SIMD–расширений.

SIMD–расширения (векторные расширения) были введены во многие стандартные архитектуры с целью повышения скорости обработки потоковых данных. Основная идея SIMD-вычислений заключается в одновременной обработке нескольких элементов данных (вектора) за одну команду.

Существует несколько способов, позволяющих реализовать возможности имеющихся SIMD–расширений в программах на языках высокого уровня. Условно их можно разделить на ручные, полуавтоматические, автоматические и с помощью готовых библиотек.

Использование вставок на ассемблере

Многие компиляторы языков Си и Си++ дают возможность вставлять в тело функции команды на ассемблере. Программисту, знакомому с ассемблером, это позволяет контролировать производительность программы в наибольшей степени среди всех описанных в этой практической работе способов. Однако использование вставок затрудняет работу компилятора по оптимизации кода. Кроме этого, теряется переносимость кода, так как

- команды во вставке рассчитаны на некоторую конкретную архитектуру,
- отсутствует стандартный синтаксис ассемблерных вставок, и различные компиляторы используют собственный синтаксис.

Рассмотрим пример программы поэлементного сложения двух векторов, состоящих из четырех чисел типа float (листинг 22). Программа может быть скомпилирована с помощью компилятора GCC, но не с помощью компилятора от Microsoft, так как в нем синтаксис ассемблерных вставок отличается.

Листинг 22. Фрагмент программы сложения двух векторов с использованием ассемблерных вставок

```
01  typedef struct{
02          float x, y, z, w;
03  } Vector4;
04  void SSE_Add(Vector4 *res, Vector4 *a, Vector4 *b){
05          asm volatile ("mov %0, %%eax"::"m"(a));
06          asm volatile ("mov %0, %%ebx"::"m"(b));
07          asm volatile ("movups (%eax), %xmm0");
08          asm volatile ("movups (%ebx), %xmm1");
09          asm volatile ("addps %xmm1, %xmm0");
10          asm volatile ("mov %0, %%eax"::"m"(res));
11          asm volatile ("movups %xmm0, (%eax)");
12  }
```

В строках 05–06 адреса двух векторов загружаются в регистры EAX и EBX. В строках 07–08 оба вектора загружаются в SIMD регистры XMM0 и XMM1 (команда MOVUPS позволяет загружать данные по невыровненным адресам). В строке 09 производится поэлементное сложение двух векторов, а в строке 11 – запись результата в память.

Использование встроенных SIMD-функций компилятора

Многие современные компиляторы поддерживают встроенные функции (intrinsics). Реализация этих функций встроена в компилятор. Вместо вызова функции из внешней библиотеки, компилятор подставляет в код программы тело функции, то есть все ее команды. Время выполнения встроенных функций

меньше, чем обычных функций, так как команды вызова подпрограммы и возврата из подпрограммы в код не включаются.

Одной из групп встроенных функций являются встроенные функции векторных расширений (SIMD intrinsics). Они обеспечивают возможность использования команд векторных расширений архитектуры с помощью привычного синтаксиса для вызова функций на языке С вместо использования ассемблерного кода и работы с регистрами процессора. В отличие от варианта с использованием ассемблерных вставок, для встроенных функций компилятор выполняет оптимизацию кода.

Рассмотрим пример программы вычисления скалярного произведения двух векторов с помощью SIMD intrinsics (листинг 23). Предполагается, что число элементов в векторах, передаваемых в функцию inner2 в качестве параметров, кратно четырем.

Листинг 23. Фрагмент программы вычисления скалярного произведения двух векторов с использованием встроенных SIMD–функций компилятора

```
01  #include <xmmintrin.h>
02  float inner2(float* x, float* y, int n){
03    __m128 *xx, *yy;
04    __m128 p, s;
05    xx = (__m128*)x;
06    yy = (__m128*)y;
07    s = _mm_setzero_ps();
08    for(int i=0; i<n/4; ++i){
09      p = _mm_mul_ps(xx[i],yy[i]);
10      s = _mm_add_ps(s,p);
11    }
12    p = _mm_movehl_ps(p,s);
13    s = _mm_add_ps(s,p);
14    p = _mm_shuffle_ps(s,s,1);
15    s = _mm_add_ss(s,p);
16    float sum;
17    _mm_store_ss(&sum,s);
18    return sum;
19  }
```

В строке 01 листинга подключается заголовочный файл с описаниями встроенных функций для векторного расширения SSE. В строке 04 определяются векторные локальные переменные: p – для хранения четырех произведений элементов векторов и s – для накопления четырех частичных сумм произведений. В строке 07 все четыре частичные суммы устанавливаются в нуль. Каждая итерация цикла в строках 08–11 обрабатывает по четыре элемента исходных векторов. На нулевой итерации обрабатываются элементы 0, 1, 2, 3, на первой – 4, 5, 6, 7 и т.д. В строке 09 вычисляется четыре произведения элементов. В строке 10 эти произведения добавляются к суммам, которые накапливаются в векторной переменной s. После выполнения цикла в переменной s хранится четыре частичные суммы (табл. 3). В строках 12–15 эти четыре частичные суммы складываются в одну общую, используя две команды сложения (векторного и скалярного) и две команды перестановки элементов векторов. В строке 17 сумма из векторного регистра записывается в переменную типа float.

Таблица 3

Номер компоненты s	**Частичная сумма, которая хранится в компоненте s**
0	x[0]*y[0] + x[4]*y[4] + x[8]*y[8] +…
1	x[1]*y[1] + x[5]*y[5] + x[9]*y[9] +…
2	x[2]*y[2] + x[6]*y[6] + x[10]*y[10] +…
3	x[3]*y[3] + x[7]*y[7] + x[11]*y[11] +….

Использование встроенных функций векторных расширений компилятора GCC

Компилятор GCC предоставляет средства для описания векторных типов данных и встроенные функции для работы с переменными этих типов. В случае их использования при компиляции исходных текстов программ необходимо указывать ключи, включающие генерацию кода для SSE или SSE2 (`-msse`, -

msse2). Рассмотрим пример функции, вычисляющей поэлементно квадрат разности значений элементов двух исходных векторов (листинг 24).

Листинг 24. Фрагмент программы вычисления квадрат разности значений элементов двух векторов с использованием встроенных функций векторных расширений GCC

```
1  typedef int v4si __attribute__ ((vector_size (16)));
2  v4si sqdif(v4si v1, v4si v2){
3    v4si r;
4    r = a - b;
5    r *= r;
6    return r;
7  }
```

В строке 1 с помощью атрибута определяется векторный тип данных. Переменные этого типа содержат четыре элемента, являющиеся целыми знаковыми числами (типа int). Размер вектора (16 байт) указан в параметре атрибута. В строке 2 указан заголовок функции, принимающей два векторных параметра и возвращающей вектор. В строке 3 объявлена локальная векторная переменная. В строке 4 поэлементно вычисляется разность значений в двух исходных векторах, а результат записывается в локальную переменную. В строке 5 все элементы вектора возводятся в квадрат. В строке 6 в качестве результата функции возвращается получившийся вектор.

Главный недостаток такого способа векторизации – привязка к компилятору GCC. К достоинствам относится отсутствие привязки к конкретной архитектуре. Т. е. возможно написание текста, который будет без модификации выполняться на различных архитектурах, имеющих SIMD-расширения. Кроме этого у текста программы, написанного с использованием данного способа, читаемость выше, чем, например, в варианте с ассемблерными вставками и SIMD intrinsics.

Подытожим сравнение трех способов векторизации. Использование ассемблерных вставок позволяет вручную строить наиболее эффективные программы. Вариант с использованием SIMD intrinsics позволяет достигать почти такой же эффективности полуавтоматически средствами компилятора, без

необходимости использования машинных команд и работы с регистрами. Оба эти варианта привязаны к одной архитектуре процессора: x86/x86-64. Этот недостаток преодолевается с помощью встроенных функций векторных расширений GCC. Данный способ подходит для разных архитектур, имеющих SIMD-команды. Кроме этого, программы, построенные с его использованием, обладают наилучшей читаемостью среди всех рассмотренных способов. Существенный недостаток этого способа заключается в привязке к компилятору GCC.

Использование автоматической векторизации компилятором

Оптимизирующий компилятор, умеющий из обычного кода генерировать код для SIMD-расширения – это наиболее простой и эффективный путь к достижению высокой производительности. Основным недостатком этого подхода является то, что код должен удовлетворять определенным критериям, чтобы компилятор мог его векторизовать. Во многих случаях компилятор не может распознать возможность эффективного применения векторных операций.

Некоторые компиляторы (например, Intel C/C++ Compiler) поддерживают специальные директивы, с помощью которых программист может дать компилятору дополнительную информацию о коде, способствующую его векторизации. Например, директива может указывать компилятору, что итерации некоторого цикла независимы друг от друга, и их можно выполнять параллельно.

Использование высокопроизводительных библиотек

Для многих предметных областей существуют эффективно реализованные библиотеки операций, оптимизированные под различные вычислительные системы. Наиболее ярким примером таких библиотек можно назвать BLAS и LAPACK, которые содержат процедуры, реализующие многие операции линейной алгебры (работа с векторами, матрицами).

Существуют реализации этих библиотеки под многие архитектуры, что обеспечивает переносимость программ, написанных с их использованием. Например, реализацию BLAS содержат библиотеки Intel MKL, AMD ACML,

NVIDIA CuBLAS, свободно распространяемая библиотека ATLAS. Каждая реализация стремится учесть все особенности целевой архитектуры для достижения высокой производительности. В частности, для временного хранения данных эффективно используется кэш-память, а для вычислений используются векторные операции.

Задание к практической работе

1. Написать три варианта программы, реализующей алгоритм из задания:
   - вариант без ручной векторизации;
   - вариант с ручной векторизацией (выбрать любой вариант из возможных трех: ассемблерная вставка, встроенные функции компилятора, расширение GCC);
   - вариант с матричными операциями, выполненными с использованием оптимизированной библиотеки BLAS.

   Для элементов матриц использовать тип данных float. Считать, что значение параметра N кратно количеству элементов типа float в векторном регистре используемого векторного расширения.
2. Проверить правильность работы программ на нескольких небольших тестовых наборах входных данных.
3. Каждый вариант программы оптимизировать по скорости, насколько это возможно.
4. Сравнить время работы трех вариантов программы для следующих значений параметров алгоритма: N = 2048, M = 10.
5. Составить отчет по практической работе. Отчет должен содержать следующее:
   - Титульный лист.
   - Цель практической работы.
   - Результаты измерения времени работы трех программ.
   - Полный компилируемый листинг реализованных программ и команды для их компиляции.

- Вывод по результатам практической работы.

Варианты заданий

1. Алгоритм обращения матрицы A размером $N \times N$ с помощью разложения в ряд:

$$A^{-1} = (I + R + R^2 + \ldots)B,$$

где $R = I - BA$, $B = \dfrac{A^T}{\|A\|_1 \cdot \|A\|_\infty}$, $\|A\|_1 = \max\limits_j \sum\limits_i |A_{ij}|$, $\|A\|_\infty = \max\limits_i \sum\limits_j |A_{ij}|$, I – единичная матрица (на главной диагонали – единицы, остальные – нули). Параметры алгоритма: N – размер матрицы, M – число членов ряда (число итераций цикла в реализации алгоритма).

Контрольные вопросы

1. Что такое SIMD–расширение архитектуры? Зачем вводятся SIMD–расширения обычных архитектур?
2. Какие SIMD–расширения архитектуры x86/x86-64 сейчас существуют? С какими регистрами и типами данных работает каждое SIMD–расширение?
3. Какие существуют способы использования SIMD–расширений в программах на языке Си?
4. Какими свойствами должна обладать программа, чтобы ее можно было векторизовать?

СПИСОК ТЕРМИНОВ

Адрес (address) – уникальный номер ячейки памяти.

Адресное пространство (address space) – множество адресов ячеек оперативной памяти.

Алгоритмическая оптимизация (algorithmic optimization) – процесс выбора оптимального алгоритма относительно заданного критерия.

Аппаратная многопоточность (hardware multithreading) – это аппаратно поддержанный способ выполнения нескольких потоков команд на одном

процессоре. Аппаратная поддержка многопоточности заключается в организации совместного использования ступеней конвейера и других ресурсов процессора (регистров, кэш-памяти) несколькими потоками команд.

Аппаратная предвыборка (hardware prefetching, hardware-controlled prefetching) – загрузка данных из оперативной памяти в кэш-память без наличия соответствующей команды. Кэш-контроллер анализирует порядок, в котором программа запрашивает данные из оперативной памяти и пытается предугадать, какие данные вскоре могут понадобиться программе, и осуществляет их автоматическую предвыборку в кэш-память.

Аппаратный регистр (hardware register) – ячейка памяти, расположенная на кристалле процессора. В отличие от других типов запоминающих устройств регистры являются самыми быстрыми. Множество регистров образуют *регистровый файл*.

Арифметико-логическое устройство (АЛУ) (arithmetic logic unit) – компонент компьютера фон Неймана, выполняющий арифметические и логические операции.

Архитектура компьютера *(computer architecture)* – это логическая организация компьютера с точки зрения программиста. К ней относятся организация памяти, набор команд, а также правила функционирования компьютера.

Ассоциативная схема отображения адресов оперативной памяти в кэш-память – это схема, при которой блок данных с любым адресом может разместиться в любой строке кэш-памяти.

Байт (byte) – единица измерения количества информации, равная 8 битам.

Бит (bit) – единица измерения количества информации, равная одному разряду в двоичной системе счисления.

Буксование кэш-памяти (cache thrashing) – эффект увеличения времени обращения к кэш-памяти, который появляется, когда в программе последовательно происходят обращения к нескольким элементам, отстоящим в

памяти на величину, кратную размеру банка ассоциативности кэш-памяти, а число таких элементов превышает степень ассоциативности этой кэш-памяти.

Буфер адресов перехода (branch target buffer) – специальная кэш-память, которая хранит целевые адреса нескольких последних команд перехода, для которых переход имел место. Буфер используется для предсказания переходов.

Буфер быстрого преобразования адресов (translation lookaside buffer, TLB) – специальная кэш-память, которая хранит физические адреса команд и данных, к которым в последнее время было обращение. Задача буфера – ускорить преобразование виртуального адреса в физический адрес.

Вектор (vector) – набор однотипных элементов данных фиксированного размера.

Векторизация (vectorization) – система преобразований исходной программы, направленная на формирование конструкций, которые допускают векторную реализацию в терминах векторов и векторных команд.

Векторная команда (vector instruction) – команда процессора, выполняющаяся над векторами в памяти или в векторных регистрах для параллельной обработки всех компонентов векторов-операндов.

Векторный процессор (vector processing unit) – процессор, который выполняет векторные команды. Векторный процессор содержит векторные регистры для хранения исходных и результирующих компонентов обрабатываемых векторов, одно конвейерное АЛУ или несколько АЛУ без конвейера или с конвейером.

Векторный регистр (vector register) – регистр, позволяющий хранить вектор значений.

Виртуальная память – это механизм управления иерархической памятью компьютера, который позволяет размещать в памяти и одновременно выполнять несколько процессов. Виртуальная память предполагает, что пользователи имеют дело с кажущейся одноуровневой памятью, объем которой равен всему адресному пространству.

Внешнее запоминающее устройство (ВЗУ) (secondary memory, external memory, auxiliary memory) – энергонезависимая память, предназначенная для длительного хранения команд и данных.

Временной параллелизм (temporal parallelism) – см. *конвейеризация*.

Время доступа к памяти (*латентность*) *(memory access time, memory latency)* – время от запроса данных из памяти до их фактического получения.

Выполнение команд вне порядка (out-of-order instruction execution, OoO) – выполнение команд в произвольном порядке в пределах некоторого окна просмотра.

Выравнивание данных (data alignment) – размещение элемента данных в памяти таким образом, чтобы его адрес был кратен некоторому заданному числу.

Глобальный способ адресации памяти (*global* mechanism of memory addressing) – способ адресации памяти, при котором каждая ячейка доступна программе с помощью адреса. Ячейка имеет один адрес. Нет двух одинаковых ячеек с одинаковым адресом.

Динамическая оптимизация программы (optimization at runtime) – оптимизация программы, осуществляемая процессором во время ее выполнения.

Динамическое предсказание перехода (dynamic branch prediction) – предсказание о наиболее вероятном исходе команды условного перехода, которое принимается исходя из собираемой в процессе выполнения программы информации о предшествующих переходах.

Иерархическая (многоуровневая) память (memory hierarchy) – память, представленная в виде системы уровней запоминающих устройств, которые характеризуются разным объемом, временем доступа и пропускной способностью.

Инструкция процессора *(CPU instruction)* – см. *команда* процессора.

Когерентность данных (data coherence, memory coherence) – согласованность (идентичность) копий одного и того же блока данных в различных запоминающих устройствах.

Команда (*инструкция*) процессора *(CPU instruction)* – элементарная операция, выполняемая процессором.

Компилятор (compiler) – программа, преобразующая код программы на языке программирования в исполняемый машинный код, который может быть выполнен на компьютере.

Конвейеризация (pipelining) – техника, в результате которой выполнение команды разбивается на несколько этапов. Каждый этап выполняется на своем логическом устройстве (ступени). Все ступени соединяются последовательно. В результате выполнение команды сводится к ее продвижению по конвейеру по мере освобождения последующих ступеней.

Конвейерное выполнение программы (*pipelined program execution*) – параллельное выполнение программы, при котором используется техника конвейеризации.

Контроллер (controller) – устройство, управляющее работой некоторого другого устройства. Например, контроллер оперативного запоминающего устройства, контроллер шины USB, контроллер жесткого диска.

Кросс-компиляция (cross-compilation) – генерация кода для архитектуры компьютера, отличной от той, на которой выполняется компилятор.

Крупнозернистая многопоточность (block multithreading) – способ организации аппаратной многопоточности, при котором конвейер в каждый момент времени выполняет команды только одного потока. Если в какой-то момент произошла длительная задержка (в результате промаха при обращении в кэш-память, зависимости по управлению или по другой причине), то процессор автоматически переключается на другой поток команд и начинает выполнять его команды.

Кэш-контроллер (cache controller) – устройство, управляющее работой кэш-памяти. В задачи кэш-контроллера входят обеспечение быстрого доступа к интенсивно используемым данным, накопление данных и удаление тех данных, которые уже не потребуются, а также поддержка когерентности данных.

Кэш-память (cache) – быстрая статическая память небольшого размера, которая содержит копии команд и данных из оперативной памяти.

Кэш-попадание (cache hit) – ситуация, когда копия запрошенной процессором из оперативной памяти ячейки оказалась в кэш-памяти.

Кэш-промах (cache miss) – ситуация, когда копия запрошенной процессором из оперативной памяти ячейки не оказалась в кэш-памяти.

Кэш-строка (cache line) – единица хранения данных в кэш-памяти.

Латентность конвейера (pipeline latency) – время между запуском команды на конвейер и ее завершением при условии отсутствия простоев конвейера.

Латентность памяти (memory latency) – см. *время доступа к памяти*.

Машина потока данных (dataflow computer) – компьютер, работа которого управляется данными, над которыми производятся вычисления.

Машина потока команд (control flow computer) – компьютер, работа которого управляется потоком команд. Например, компьютер фон Неймана.

Мелкозернистая многопоточность (interleaved multithreading) – способ организации аппаратной многопоточности, при котором каждая ступень конвейера на каждом такте переключается между потоками.

Микроархитектура (организация) компьютера *(microarchitecture)* – совокупность инженерных решений для аппаратной реализации той или иной программной архитектуры. В этом значении архитектура описывает, как реализована память, процессор, число функциональных устройств, число аппаратных регистров, наличие сопроцессоров и т.п.

Многопоточный процессор (multi-threaded processor) – процессор, способный выполнять одновременно несколько потоков команд.

Многоядерный процессор (multi-core processor) – процессор с несколькими процессорными *ядрами* на одном или нескольких кристаллах.

Множественно-ассоциативная схема отображения адресов оперативной памяти в кэш-память – схема, при которой блок с заданным адресом может разместиться только в некотором множестве строк кэш-памяти.

Набор команд (*инструкций*) процессора *(CPU instruction set)* – множество команд, которые процессор способен выполнять.

Независимые команды (independent instructions) – группа команд, у которой для каждой пары команд множество входных параметров одной команды не пересекается с множеством результирующих данных другой команды.

Одновременная многопоточность (simultaneous multithreading) – способ организации аппаратной многопоточности, при котором ступени конвейера могут обрабатывать одновременно несколько команд с разных потоков.

Оперативная память (primary memory) – последовательность ячеек памяти одинакового размера, однозначно определяемых своим адресом. Служит для хранения команд и данных во время выполнения программы.

Оперативное запоминающее устройство (ОЗУ) (primary memory, random access memory) – энергозависимая память с произвольным доступом. Служит для реализации оперативной памяти.

Оптимизация программы (program optimization) – проведение таких эквивалентных преобразований программы (т.е. таких, которые не изменяют реализованный ею алгоритм), которые уменьшают время ее выполнения.

Организация компьютера *(computer organization)* – см. *микроархитектура*.

Отладчик (debugger) – инструментальное средство для поиска ошибок в программе путем ее прогона в особом, отладочном режиме. Отладочный режим позволяет выполнять команды и подпрограммы по шагам или непрерывно до достижения заданных точек приостановки выполнения, просматривать значения регистров процессора, ячеек памяти.

Память (memory) – в аппаратной архитектуре общее название различных типов запоминающих устройств (см. *оперативное запоминающее устройство, внешнее запоминающее устройство, кэш-память*), в программной архитектуре общее название для ячеек, способных хранить числовые значения (см. *оперативная память*, *регистр*).

Память с последовательным доступом (sequential access memory) – память, допускающая обращение к своим ячейкам только в последовательном порядке. Например, файл на магнитной ленте.

Память с произвольным доступом (random access memory) – память, допускающая обращение к своим ячейкам в любом порядке. Например, оперативная память.

Параллелизм на уровне данных (data level parallelism) – возможность одновременной однотипной обработки нескольких комплектов данных с помощью одной команды.

Параллелизм на уровне команд (instruction level parallelism, ILP) – возможность одновременного выполнения нескольких независимых команд из одного потока команд.

Параллелизм на уровне потоков (instruction level parallelism) – возможность одновременного выполнения нескольких потоков команд в программе.

Параллельное выполнение программы (*parallel* program execution) – выполнение программы, при котором допускается одновременное выполнение двух или более команд программы.

Переименование регистров (register renaming) – динамическое отображение логических регистров, определяемых программной архитектурой, на физические регистры компьютера. В результате такой процедуры устраняются несущественные зависимости по данным.

Пиковая производительность процессора (компьютера) *(peak performance)* – количество операций, выполняемых в единицу времени всеми функциональными устройствами процессора (компьютера).

Поток (thread) – см. *поток команд* или *программный поток*.

Поток команд (thread, instruction flow) – последовательность команд, поступающих на вход процессору.

Предвыборка (prefetch) – загрузка данных из оперативной памяти в кэш-память до непосредственного их использования. Различают *аппаратную* и *программную предвыборку*.

Предсказание перехода (branch prediction) – техника, позволяющая выполнять выборку и декодирование потока команд по одной из ветвей, не дожидаясь проверки самого условия перехода. Различают *статическое* и *динамическое предсказание перехода*.

Принцип локальности ссылок (principle of locality, locality of reference) – эмпирическое правило, согласно которому процессор с большой вероятностью многократно использует одни и те же команды и данные (локальность во времени), а также команды и данные по адресам, близким к недавно использованным (локальность в пространстве).

Программа (program) – представление алгоритма, выполняемое на реальном компьютере. В компьютере фон Неймана программа представлена последовательностью команд.

Программная оптимизация (software optimization) – оптимизация программы, которая не изменяет ее алгоритм.

Программная предвыборка (software prefetching, software controlled prefetching) – загрузка данных из оперативной памяти в кэш-память в результате выполнения специальной команды предвыборки. Команды предвыборки явно добавляются в текст программы компилятором или программистом.

Программный поток (thread, lightweight process) – термин операционной системы, последовательность команд, выполняемых в составе процесса.

Программный регистр (register, user-accessible register) – ячейка памяти, часть программной архитектуры компьютера. Доступ к программному регистру осуществляется по имени.

Пропускная способность (throughput) – максимальный объем данных, проходящий через устройство в единицу времени. Например, пропускная способность шины данных, кэш-памяти.

Простой конвейера (pipeline stall, bubble) – ситуация, когда стадия конвейера не может сработать по каким-либо причинам.

Пространственный параллелизм (spatial parallelism) – параллельное выполнение независимых команд несколькими функциональными устройствами.

Протокол поддержки когерентности (cache coherency protocol) – система состояний блоков данных (измененный, эксклюзивный, недействительный,…) и правил изменения состояний в результате различных событий (чтение, запись). Протокол обеспечивает *когерентность данных* между всеми кэш-памятями в компьютере.

Процесс (process, job) – термин операционной системы, экземпляр запущенной программы, обладающий собственными ресурсами (например, адресное пространство). В рамках процесса может выполняться один или несколько программных потоков.

Процессор (processor, central processing unit, central processor unit, CPU) в компьютере фон Неймана объединяет устройство управления, арифметико-логическое устройство и регистровый файл.

Процессор с длинным командным словом (very long instruction word, VLIW processor) – процессор с параллелизмом на уровне команд, в котором параллельная обработка указывается явно в специально отведенных полях команды. Выявление команд, которые могут быть выполнены параллельно, обеспечивает компилятор.

Прямая схема отображения адресов оперативной памяти в кэш-память – схема, при которой блок с заданным адресом может разместиться только в одной строке кэш-памяти.

Разрядность (word size) – числовая характеристика процессора, которая равна размеру регистров процессора в битах.

Регистр (register) – см. *программный регистр* как часть программной архитектуры или *аппаратный регистр* как часть аппаратной архитектуры компьютера. Отображение программных регистров на аппаратные выполняется с помощью механизма переименования регистров.

Регистровый файл (register file) – функциональный блок процессора, который содержит аппаратные регистры.

Регистры общего назначения (general purpose register) – регистры, предназначенные для хранения операндов арифметико-логических инструкций, а также адресов или отдельных компонентов адресов ячеек памяти.

Связка команд (bunch of instructions) – набор простых независимых друг от друга команд.

Скалярный процессор (scalar processor) – процессор, в котором одна команда может обрабатывать только один комплект данных.

Слово (word) – единица хранения информации, число разрядов в которой определяется разрядностью архитектуры. Например, для 32-разрядной архитектуры размер слова равен 32 бита или 4 байта.

Спекулятивное выполнение команд (speculative execution) – выполнение команд выбранной ветви программы до завершения проверки условия перехода. Выбор ветви осуществляется с помощью механизма предсказания перехода.

Статическая оптимизация программы (static program optimization) – оптимизация программы, выполняемая программистом или компилятором до ее выполнения.

Статическое предсказание перехода (static branch prediction) – предсказание наиболее вероятного результата команды условного перехода, закодированное в коде команды. Статическое предсказание основывается на некотором априорном знании компилятора о ходе выполнения программы.

Степень ассоциативности кэш-памяти *(cache associativity)* – характеристика кэш-памяти с множественно-ассоциативным отображением адресов, количество кэш-строк в одном множестве.

Суперскалярный процессор (superscalar processor) – процессор с параллелизмом на уровне команд, в котором распознавание и одновременное выполнение независимых команд достигается аппаратными средствами.

Счетчик команд (program counter, instruction pointer) – специальный регистр процессора, который хранит адрес следующей команды для выполнения.

Такт процессора *(clock tick)* – промежуток времени между двумя импульсами тактового генератора.

Тактовая частота (clock rate) – частота генерации импульсов тактовым генератором.

Тактовый генератор (clock generator) – устройство, генерирующее электрические импульсы с заданной частотой для синхронизации процессов в цифровых устройствах, например, в процессорах.

Тэг (tag) – часть адреса, которая идентифицирует блок оперативной памяти, который записан в строку кэш-памяти в данный момент времени.

Устройство управления (УУ) *(control unit)* – компонент компьютера фон Неймана, управляющее работой всех остальных компонентов процессора.

Уплотнение данных (data packing) – вид программной оптимизации, который заключается в изменении размещения данных в оперативной памяти, так чтобы уменьшить расстояние между совместно используемыми элементами данных. Используется, чтобы избежать загрузки неиспользуемых данных в кэш-память.

Упорядоченное выполнение команд (in-order instruction execution) – такое выполнение команд, при котором команды рассматриваются в том порядке, в котором они идут в программе.

Уровень оптимизации (optimization level) компилятора – множество используемых компилятором оптимизирующих преобразований.

Физический адрес (physical address) – адрес ячейки оперативной памяти. В системах с виртуальной памятью физический адрес вычисляется по виртуальному адресу при каждом доступе к памяти. Термин используется только для систем с поддержкой виртуальной памяти. Для иных систем используется термин *адрес*.

Функциональное устройство процессора *(CPU functional unit)* – см. *арифметико-логическое устройство*.

Ширина конвейера (pipeline width) – количество независимых команд, которые могут быть выполнены конвейером параллельно.

Центральный процессор (central processing unit, CPU) – см. *процессор*.

Ядро процессора *(core)* – компонент многоядерного процессора, включающий устройство управления, арифметико-логическое устройство и

регистровый файл и функционирующий как отдельный процессор компьютера фон Неймана.

Ячейка памяти *(memory cell)* – минимально адресуемая единица информации. Компьютер может прочитать или записать значение в ячейку только целиком. Практически во всех современных компьютерах размер ячейки равен одному байту.

SIMD-расширения (Single Instruction Multiple Data extensions) – это функциональный блок процессора, реализующий векторные команды.

VLIW-процессор (VLIW processor, Very Long Instruction Word processor) – процессор с параллелизмом на уровне команд, который получает на вход программу в виде последовательности связок команд (длинных командных слов). Команды в каждой связке являются независимыми и могут выполняться VLIW-процессором параллельно.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Векторизация программ: теория, методы, реализация: Сб. статей: Пер. с англ. и нем. – М.: Мир, 1991. – 275 с.
2. Параллельные вычисления. Воеводин В. В., Воеводин Вл. В. – СПб., 2004. – 599 с.
3. Выполнение математических операций в ЭВМ. Погрешности компьютерной арифметики. Учебное пособие / СПбГЭТУ "ЛЭТИ". Грушин В.В. СПб., 1999. – 56 с.
4. Архитектура микропроцессоров: Учебное пособие. Гуров В.В. – М.: Интернет-Университет Информационных Технологий: БИНОМ. Лаборатория знаний, 2010. – 272 с.
5. Многоядерные процессоры: Учебное пособие. Калачев А.В. – М.: Интернет-Университет Информационных Технологий: БИНОМ. Лаборатория знаний, 2011. – 247 с.
6. Теория оптимизации программ. Эффективное использование памяти. – БХВ-Петербург, Касперски К. СПб.: 2003. – 464 с.
7. Искусство программирования. Том 1. Основные алгоритмы. – Вильямс, Кнут Д,Э. 2010. – 720 с.
8. Искусство программирования. Том 2. Получисленные алгоритмы. – Вильямс, Кнут Д.Э. 2011. – 832 с.
9. Искусство программирования. Том 3. Сортировка и поиск. – Вильямс, Кнут Д.Э. 2012. – 824 с.
10. Искусство программирования. Том 4, А. Комбинаторные алгоритмы. Часть 1. – Вильямс, Кнут Д.Э. 2012. – 960 с.
11. Современные микропроцессоры. – Корнеев В.В., Киселев А.В. 3-е изд., перераб. и доп. – СПб: БХВ-Петербург, 2003. – 448 с.
12. Магда Ю.С. Аппаратное обеспечение и эффективное программирование. – СПб.: Питер, 2007. – 352 с.

13. Архитектура компьютера и проектирование компьютерных систем. – СПб.: Питер, Паттерсон, Дж. Хеннесси. 2012. – 784 с.
14. Архитектура вычислительных систем и компьютерных сетей. – Степанов А.Н. СПб.: Питер, 2007. – 509 с.
15. Структурная организация и архитектура компьютерных систем. Столлингс В. 5-е издание. - М.: Издательский дом "Вильямс", 2002. – 896 с.
16. Архитектура компьютера. Таненбаум Э. С. 4-е изд. – СПб., 2007. – 843 с.
17. Архитектура вычислительных систем. Хорошевский В. Г. – М., 2005. – 510с.
18. Организация ЭВМ и систем. – Цилькер Б. Я., Орлов С. А. – СПб., 2011. 686 с.
19. The essentials of computer organization and architecture. – Jones and Bartlett Publishers. Linda Null and Julia Lobur. – 2003. – 673 P.
20. Advanced computer architecture and parallel processing. Mostafa Abd-el-Barr, Hesham el-Rewini. – 2005. – 272 P.
21. 21st Century computer architecture. – A community white paper. – 2012.
22. Concept and development of modular VLIW processor based on FPGA. – 2nd Intl. Conf. on Computer and Network Technology. – IEEE Computer Society. D. Saptono, V. Brost, F. Yong, E. Wibowo. – 2010. – pp. 561-565.
23. Advanced computer architectures. New York: Addison Wesley Longman Inc., D. Sima, T. Fountain, P. Kacsuk. – 1997. – 766 P.
24. The Working Set Model for Program Behaviors. – Communications of the ACM, P. Denning. – May 1968. – pp. 323-333.
25. Computer organization and architecture designing for performance. – 8th edition – Prentice Hall. William Stallings. – 2010. – 774 P.
26. Computer architecture. A quantitative approach. 5th edition. – Elsevier. John L. Hennessy, David A. Patterson. – 2012. – 831 P.
27. Вычислительные машины и труднорешаемые задачи. – М. – Мир. Гери М., Джонсон Д. – 1982. – 412 с.

В электронном виде

28. Agner Fog. Software optimization resources. – http://www.agner.org/optimize
29. Barney B. POSIX threads programming [Electronic resource] / B. Barney. POSIX threads programming. – Mode of access: https://computing.llnl.gov/tutorials/pthreads/. – Title from screen.
30. Berkeley UPC – Unified Parallel C [Electronic resource]: (a joint project of LBNL and UC Berkeley) / Berkeley Laboratory. – Berkeley, 2011. – Mode of access: http://upc.lbl.gov. – Title from screen.
31. GNU Compiler Collection. – http://gcc.gnu.org/onlinedocs/gcc
32. First Draft of a Report on the EDVAC (http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf)
33. GNU Compiler Collection. – http://gcc.gnu.org/onlinedocs/gcc
34. Intel SSE Tutorial: An Introduction to the SSE Instruction Set. – http://neilkemp.us/src/sse_tutorial/sse_tutorial.html
35. Peter J. Denning. – Thrashing: Its causes and prevention. – Proc. AFIPS Conf. 32 (1968 Fall Joint Computer Conference), pp. 915-922.
36. Tim Jones. – Optimization in GCC. – Linux Journal. Iss. 131. - 2005. – http://www.linuxjournal.com/article/7269
37. TOP500 List [Electronic resource] // Top 500. Supercomputer sites. – 2011. – Mode of access: http://www.top500.org/list/20q03/11/. – Title from screen.
38. Intel SSE Tutorial: An Introduction to the SSE Instruction Set. – http://neilkemp.us/src/sse_tutorial/sse_tutorial.html

ОГЛАВЛЕНИЕ