Оглавление

[Глава 1. Вычислители 3](#_Toc6741999)

[1.1. Электронная вычислительная машина. Классификация ЭВМ 3](#_Toc6742000)

[1.2. Архитектура вычислителя и его расширяющие компоненты 9](#_Toc6742001)

[1.3. Вычислители с множественными потоками команд и одиночным потоком данных (MISD), истинно параллельные процессоры (MIMD), систолические вычислители. 33](#_Toc6742002)

[1.4. Специализированные вычислители. 39](#_Toc6742003)

[1.5. Графический процессор 43](#_Toc6742004)

[1.6. Программная среда специализированного вычислителя 46](#_Toc6742005)

[Глава 2. Процессор (компьютер) 49](#_Toc6742006)

[2.1. Узлы процессора 49](#_Toc6742007)

[2.2. Ядро процессора 60](#_Toc6742008)

[2.3 Прерывания 68](#_Toc6742009)

[2.4. Представление о микросхемах обвязки процессоров 76](#_Toc6742010)

[2.5. Представление о конвейерном выполнении команд в процессорах 79](#_Toc6742011)

[2.6. Классификация и особенности шин в отношении способа передачи, метода синхронизации, топологии, способа управления, адресации устройств 87](#_Toc6742012)

[2.7. Виртуальные машины. Виртуализация и эмуляция. Гипервизор и супервизор. Контейнеры и докеры. 97](#_Toc6742013)

[2.8. Ядро ОС. Планировщик и механизмы переключения задач 105](#_Toc6742014)

[Глава 3. Процессор (программирование) 110](#_Toc6742015)

[3.1. Понятие о параллельных и распределенных вычислениях. Программно-аппаратные переходы и аспекты их применения. 110](#_Toc6742016)

[3.2.Компиляция и интерпретация исходных текстов программ. Объектный код, исполняемый код, динамическое и статическое связывание модулей программы. Байт-код. 114](#_Toc6742017)

[3.3. Специализированные средства разработки программного обеспечения 119](#_Toc6742018)

[3.4. Загрузчики и компоновщики. Форматы объектных модулей. Оверлейные загрузчики и оверлеи. 125](#_Toc6742019)

[Глава 4. Параллельные вычисления 129](#_Toc6742020)

[Глава 5. Оптимизация кода 134](#_Toc6742021)

[5.1. Оптимизация циклов (loop optimization) 134](#_Toc6742022)

[5.2. Индексация циклов к нулю 143](#_Toc6742023)

[5.3. Ветвления 145](#_Toc6742024)

[5.4. Оптимизация массивов 151](#_Toc6742025)

[5.5. Оптимизация switch 155](#_Toc6742026)

[5.6 Устройство кэш памяти 160](#_Toc6742027)

[5.7 Выравнивание данных (data alignment) 168](#_Toc6742028)

[5.8 Все данные размером машинного слова 171](#_Toc6742029)

[5.9 Борьба с глобальными объектами 172](#_Toc6742030)

[5.10 Ускорение вызова функций 174](#_Toc6742031)

[5.11 Передача большого количества параметров 178](#_Toc6742032)

[5.12 Как избегать замедления работы конвейера. 180](#_Toc6742033)

[5.13 Виртуальная машина 182](#_Toc6742034)

[5.14 Хранение или вычисление заново? 185](#_Toc6742035)

[5.15 Утечка памяти 187](#_Toc6742036)

[5.16 Треды 189](#_Toc6742037)

[Приложение 195](#_Toc6742038)

[Список сокращений 199](#_Toc6742039)

# Глава 1. Вычислители

## 1.1. Электронная вычислительная машина. Классификация ЭВМ

*Определение 1.1.1.*

Электронная вычислительная машина (ЭВМ) или компьютер — это автоматическое устройство, которое совершает обработку информации.

Компьютеры классифицируются по следующим критериям:

Классификация компьютеров в отношении круга вычисляемых задач:

Критерием деления вычислительных машин на эти три класса служит форма представления информации, которой они занимаются.

* Цифровые вычислительные машины (ЦВМ) – машины дискретного действия, то есть выполняющие пошаговую работу с информацией, показанной в цифровой форме.
* Аналоговые вычислительные машины (АВМ) – вычислительные машины беспрерывного действия, занимаются с информацией, предложенной в аналоговой (беспрерывной) форме, то есть в виде непрерывного ряда значений какой-либо физической величины (чаще всего электрического напряжения).
* Гибридные вычислительные машины (ГВМ) – вычислительные машины комбинированного действия, занимаются информацией, показанной и в цифровой, и в аналоговой форме; они сочетают в себе преимущества АВМ и ЦВМ. ГВМ полезно эксплуатировать для решения задач управления сложными высокоактивными техническими комплексами.

Классификация компьютеров по условиям использования:

* офисные – созданы для решения значительного спектра задач в стандартных условиях использования.
* специальные – созданы для решения малого круга задач либо одной задачи многократного решения. Используются в автомобилях скорой помощи, вертолетах и самолетах, на ракетах, в зоне действия радиопередатчиков, радаров, под водой на глубине, в условиях грязи, пыли, вибраций, взрывоопасных газов и так далее.

Машинные возможности специальных компьютеров, в большинстве случаях, ограничены. Но вследствие их узкой ориентации формулированный круг задач выполняется особенно продуктивно.

Классификация компьютеров в отношении архитектуры:

*Определение 1.1.2.*

Архитектура компьютера — логическая организация и структура аппаратных возможностей вычислительной системы. Архитектура включает в себя требования к функциональности и принципы координирования важнейших узлов компьютера.

В настоящее время наиболее распространены в компьютерах два типа архитектуры:

* [гарвардская](http://ru.wikipedia.org/wiki/%D0%93%D0%B0%D1%80%D0%B2%D0%B0%D1%80%D0%B4%D1%81%D0%BA%D0%B0%D1%8F_%D0%B0%D1%80%D1%85%D0%B8%D1%82%D0%B5%D0%BA%D1%82%D1%83%D1%80%D0%B0).
* принстонская (фон Неймана);

Для представления каждого из типов добавим еще ряд определений.

*Определение 1.1.3.*

Устройства ввода/вывода (УВВ) — устройства, с помощью которых реализуется взаимодействие компьютера и пользователя. К примеру, ввод информации в память компьютера выполняется с помощью мыши и клавиатуры, а вывод представляется на мониторе или печатается на принтере.  
С помощью интерфейса ввода/вывода (ИВВ) реализуется обмен информацией между данными устройствами.

*Определение 1.1.4.*

Шина — канал связи, который соединяет между собой все электронные компоненты компьютера.

На рисунках ниже присутствуют три типа шин. При помощи адресной шины (см. *Определение 2.6.10*) формируется указатель на требуемые данные, а шина данных передает их компонентам компьютера. По шине управления (см. *Определение 2.6.12*) передаются сигналы, которые определяют характер обмена информацией, например, сигналы чтения, записи, готовности.

*Определение 1.1.5.*

Процессор — это часть компьютера, отвечающая за интерпретирование и выполнение команд, получаемых от аппаратного и программного обеспечения.Наиболее важными частями процессора являются тесно взаимосвязанные между собой устройства: устройство управления и арифметико-логическое устройство. Их определения представлены ниже.

*Определение 1.1.6.*

Устройство управления (УУ) — схема, с помощью которой обрабатывается информация, поступающая процессору. Сначала эту информацию получает УУ, преобразует ее в сигналы управления и направляет в ЦП. Затем он сообщает подключенному оборудованию, какие операции необходимо выполнить. Так, арифметико-логическое устройство, память, а также устройства ввода и вывода получают представление об ожидаемой реакции на команды от программы.

*Определение 1.1.7.*

Арифметико-логическое устройство (АЛУ) — схема, с помощью которой выполняются арифметические и побитовые операции над данными, называемыми операндами. Кроме данных, этому устройству сообщается код с указаниями, что с ними делать. После обработки информация отправляется в память компьютера. За внесение данных и последующее сохранение отвечает УУ. Оба устройства содержат регистры. Одним из важнейших считается сумматор, способный складывать многоразрядные двоичные числа.

*Определение 1.1.8.*

Регистры *—* это специальные ячейки памяти, расположенные непосредственно в процессоре. Работа с регистрами выполняется намного быстрее, чем с ячейками оперативной памяти (см. *Определение 1.2.9*), поэтому регистры активно используются как в программах на языке ассемблера, так и компиляторами языков высокого уровня. В них может содержаться адрес операнда или команды, результат ее выполнения, счетчик команд.

Команда хранится в специальном регистре столько времени, сколько требуется для ее выполнения. Такой регистр называется регистром команд и содержится в УУ. А регистр, в котором хранится адрес очередной команды, называют счетчиком команд. Это понятие важно для архитектуры фон Неймана, описанной ниже.

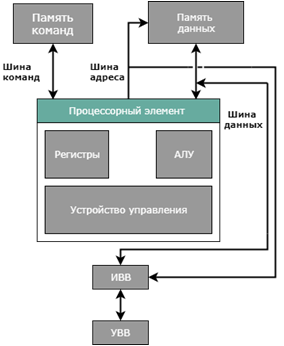
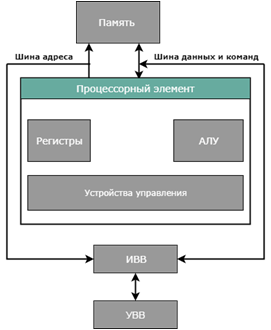
****

Рис.1.1.1. Принстонская архитектура Рис.1.1.2. Гарвардская архитектура

Принстонская архитектура (архитектура фон Неймана) отличается использованием общей оперативной памяти для создания стека, а также для содержания программ и данных, которые поступают в процессор по общей системной шине (см. *Определение 2.2.8.*). Системная шина также используется для обращения к оперативной памяти.

Классическая гарвардская архитектура особенна использованием разделенной памяти.

Отличие принстонской архитектуры и гарвардской в памяти, а точнее в ее структуре. В первой программы и данные хранятся в одном массиве памяти и передаются в процессор по одной шине, а во второй архитектуре предусматриваются отдельные хранилища и потоки передачи для команд и данных.

По работоспособности и характеру использования компьютеры можно классифицировать так:

*Определение 1.1.9.*

Микрокомпьютер — компьютер, особенностью которого является то, что он находится на одной микросхеме.

У продвинутых моделей микрокомпьютеров имеется несколько процессоров. Производительность компьютера определяется еще и емкостью оперативной памяти, типами периферийных устройств (см. *Определение 1.2.13*), качеством конструктивных решений и др.

*Определение 1.1.10.*

Микроконтроллер — специализированный компьютер или его часть, которая обеспечивает работоспособность какого-либо узла или устройства компьютера.

*Определение 1.1.11.*

Персональные компьютеры — компьютеры, рассчитанные на использование небольшим количеством людей.

*Определение 1.1.12.*

Рабочие станции — персональные мощные компьютеры, выполняющие рабочие задачи.

*Определение 1.1.13.*

Серверы — многопользовательские мощные компьютеры в вычислительных сетях, которые обрабатывают запросы от всех станций сети.

Выделим следующие группы компьютеров:

* мэйнфреймы — распределенные компьютеры. Это означает, что разные узлы одного компьютера, из-за своих размеров, оформлены в виде отдельных блоков. Такие компьютеры используются в работе с активными вычислениями и данными больших размеров.
* миникомпьютеры — компьютеры малых размеров, обладающие более низкими параметрами по сравнению с мэйнфреймами. К таким компьютерам могут подключаться сотни устройств и микрокопьютеров.
* суперкомпьютеры — это большое количество серверов, для достижения максимальной производительности они соединены локальным высокоскоростным каналом. Используются для работы с приложениями, требующими наиболее сложных вычислений.  
    
  Критерии выбора решений:
* производительность;
* стоимость;
* надежность;
* масса;
* габариты.

1.2. Архитектура вычислителя и его расширяющие компоненты

*Определение 1.2.1*

Под архитектурой вычислительного средства понимается совокупность общих принципов организации аппаратно-программных средств и их характеристик, определяющая функциональные возможности компьютера при решении соответствующих классов задач.

Обязательные компоненты вычислителя:

* процессоры;
* энергонезависимая память (см. *Определение 1.2.5*);
* системы управления и ввода/вывода информации.

1.2.1. Кэш

Понятие о «горячих» и «холодных» данных.

Разные данные используются пользователями с разной частотой. Например, для ежедневной отчетности и оперативного анализа нужны данные за последние несколько дней, в то время как обращение к данным, которые старше года, требуется не каждый день, и по малому количеству показателям. Характеристику данных через частоту обращения к ним легко представляется в виде температуры, как в школьной физике, только вместо молекул здесь «движутся» блоки данных. Таким образом, данные, лежащие без малейшего движения на текущем временном интервале, называют холодными, а наиболее востребованные – горячими.

*Определение 1.2.2.*

Кэш-память — высокоскоростная память произвольного доступа, не имеющая адресов и используемая процессором компьютера для временного хранения данных.

Кэш-память служит своеобразным буфером между различными устройствами для временного хранения и обработки данных. Время доступа к информации, хранящейся в кэш-памяти, меньше, чем время доступа к этой же информации, хранящейся в других видах памяти компьютера.

В зависимости от назначения кэш-память представляется в следующей иерархии:

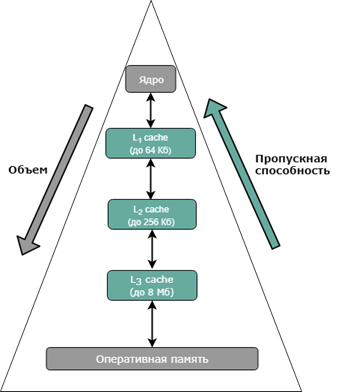
****

Рис. 1.2.1. Уровни кэш-памяти

* Кэш первого уровня (L1) — наиболее быстрый и наименьший по объему, имеет время доступа порядка нескольких тактов и размер в десятки Кбайт. Содержит данные, чаще всего используемые процессором, работает на частотах близких ему и напрямую с его ядрами, причем каждое из ядер имеет доступ только к своей микросхеме L1. Также является буфером между процессором и кэш-памятью второго уровня.
* Кэш второго уровня (L2) – от 2 до 10 раз медленнее L1, но имеет более существенный объем (несколько сотен Кбайт). Предназначена для временного хранения важной информации, у которой вероятность запроса ниже, чем у информации из L1. Служит буфером между уровнями L1 и L3.
* Кэш третьего уровня (L3)  – в массовых процессорах используется недавно. Он медленнее (время доступа около сотни тактов), чем два предыдущих, но гораздо быстрее оперативной памяти, объемом до нескольких десятков Мбайт. Данный уровень является общим для всех ядер процессора, в отличие от L2 и L3. L3 обеспечивает взаимодействие ядер процессора между собой и хранит важные данные с относительно низкой вероятностью запроса. При отсутствии данного уровня кэш-памяти L2 совмещает в себе функции L2 и L3.

Тэги доступа

Каждый структурный компонент кэш-памяти кроме байт данных оперативной памяти должен сохранять и его четырехбайтовый адрес в оперативной памяти. Такая своего рода строка называется кэш-строкой(*cache-line*).

В кэше всякой из кэш-строк дополнительно назначается в соответствие тег (*tag*), являющийся адресом повторяющихся в этой кэш-строке данных в основной памяти. Помимо 24 бит тег имеет бит значимости, проверяющий действительность строки (когда бит значимости 0, данная строка принимается недействительной и обращения к данным не кэшируются).

При доступе процессора в память прежде всего проверяется, содержит ли кэш требуемые из памяти данные. Для этого совершается сопоставление адреса запроса значениям всех тегов кэша, в которых запрашиваемые данные могут храниться. Если есть совпадение с тегом некоторой кэш-строки — это называется попаданием в кэш (*cache hit*), тогда осуществляется чтение или запись данных в этой кэш-строке, иная ситуация именуется кэш-промахом (*cache miss*).

Отношение числа попаданий в кэш к общему числу запросов к памяти называют рейтингом попаданий (*hit rate*), определяет меру производительности кэша для выбранного алгоритма или программы.

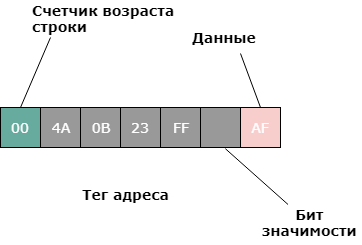
****

Рис. 1.2.2. Структура гипотетического кэша

Загрузка и выгрузка

Устройство, которое управляет содержанием кэша, получает требуемые данные из оперативной памяти, передает их процессору и возвращает итоги операций в оперативную память, называется кэш-контроллером.

Таким образом, главное назначение кэш-контроллера – заполнить кэш актуальными данными, причём с минимальным количеством кэш-промахов.

Загрузка кэша данными строится на принципах стратегий кэширования, а выгрузка данных — на базе политики замещения.

Варианты стратегий помещения данных в кэш-память:

* кэширование по требованию (*on demand*): обращение к оперативной памяти (с последующим помещением копии данных в кэш) происходит при возникновении кэш-промаха;
* на основе прогноза о данных, которые потребуются в ближайшем будущем, кэш-контроллер заранее осуществляет их загрузку в кэш-память;
* упреждающая спекулятивная загрузка данных:
  + загрузка из оперативной памяти в кэш не только затребованных данных, но и соседних данных в порядке возрастания адресов (эффективно для последовательной обработки данных);
  + адрес следующей запрашиваемой ячейки памяти предсказывается на основе анализа предыдущих обращений (интеллектуальные стратегии *Look Through* и *Look Aside*);

Первая стратегия имеет достаточно высокую частоту кэш-промахов, вследствие чего не используется. Последняя стратегия имеет высокую эффективность и сводит частоту возникновения кэш-промахов к ничтожно малому значению.

Существует несколько вариантов политики замещения:

* замещаемые данные выбираются произвольным образом (*Random*);
* замещаются данные с самая низкой частота обращений (*Least Frequently Used, LFU*);
* замещаются данные, к которым дольше всего не обращались (*Least Recently Used, LRU*);
* замещаются данные, загруженные раньше всех (*First Input First Output, FIFO* (см. *Определение 2.1.12.*) или *Least Recently Replaced, LRR*);

Первый вариант самый простой и не интеллектуальный, при этом не применяется т.к. неэффективен. Алгоритмы же LRU и FIFO (см. *Определение 2.1.12.*) используются во всех современных процессорах.

Поддержание когерентности кэш-памяти

Когерентность кэш-памяти — алгоритм, который обеспечивает согласованность данных, содержащихся в оперативной и кэш-памяти, выполняется кэш-контроллером.

Если бы оперативная память (см. *Определение 1.2.9*) разрешала лишь чтение информации, то ее копия в кэш-памяти была бы идентична оригиналу в оперативной памяти. Но поскольку в оперативной памяти также можно записывать данные, обновление их в оперативной памяти очень важно для корректной работы с данными*.*

Действительно, к оперативной памяти могут обращаться не только процессор, но и периферийные устройства (см. *Определение 1.2.13.*), а в многопроцессорных и многоядерных процессорах (см. *Определение 2.2.2.*) разные процессоры или ядра. Предположим, ячейка памяти, уже преобразованная в кэше, но пока не выгруженная в основную память, получает запрос от периферийного устройства или другого процессора. Тогда кэш-контроллеру нужно прежде всего обновить данные соответствующей ячейки оперативной памяти, в противном случае считаются неактуальные данные. Точно также же, в случае изменения содержимого ячейки оперативной памяти периферийным устройством или другим процессором, кэш-контроллер обязан проверить, загружены ли обновленные данные ячейки в его кэш-память, и если нет, то это обязательно сделать.

Значит, необходимо вовремя поддерживать когерентность между кэшем и оперативной памятью.

Пути реализации когерентности:

1. Сквозная запись (*Write True*) — простейшее, но самое неэффективное решение поддержки когерентности. Оперативная память кэшируется исключительно на чтение, а запись производится, пропуская кэш, прямо в оперативную память. Исходя из своей неэффективности алгоритм не применяется в современных процессорах.

2. Алгоритм обратной записи (*Write Back*) — сложнее предыдущего. Для мониторинга изменения данных в кэш-памяти все кэш-строки содержат индивидуальный флаг (см. *Определение 2.3.5*) состояния. Если данные кэш-строки изменились, то флаг становится в состояние «модифицированное» (*Dirty*). При обращении к памяти периферийного устройства кэш-контроллер делает проверку на наличие запрашиваемых данных в кэше, и если такие есть, то по флагу состояния устанавливается актуальность данных в оперативной памяти. Если значение флага «Dirty», то в оперативной памяти данные не актуальны, и содержимое этой кэш-строки выгружается в оперативную память, а флаг ставится в состояние «не модифицировано» (*Clear*). При замещении кэш-строк кэш-контроллер таким же образом вначале анализирует состояние флага, и если строка была изменена, то она предварительно выгружается в оперативную память.

3. Протокол MESI применяется в реализации когерентности в современных процессорах. Он является модификацией алгоритма Write Back с четырьмя допустимыми значениями флага изменения кэш-строки:

* Модифицированная (*M, Modified*). Текущая строка была преобразована, но до памяти эти преобразования пока не дошли. Возможны чтение и запись в нее без опроса остальных.
* Эксклюзивная (*E, Exclusive*). Данные этой строки уникальны и не хранятся в остальных кэшах (для кэша более, чем одного уровня), кроме того они полностью совпадают с данными в оперативной памяти. Запись и чтение осуществляется без внешних запросов, т.к. она содержится только в одном кэше. После записи такая строка меняет значение флага на Modified.
* Разделяемая (*S, Shared*). Строка может находится в кэшах сразу нескольких устройств и применяться совместно. Запросы на запись в такую строку направляются на общую шину, вследствие чего все строки с таковым адресом отмечаются как старые в остальных кэшах. В это время данные основной памяти тоже обновляются. А чтение из такой строки, не запрашивает никаких внешних запросов.
* Недействительная (*I, Invalid*). Строка является недопустимой, и попытка прочитать ее содержимое повлечет кэш-промах. Данный флаг (см. *Определение 2.3.5*) устанавливается, когда строка пуста или имеет неактуальную информацию.

1.2.2. Прямой доступ к памяти

Прямой доступ к памяти (ПДП) — это главный способ разгрузить процессор от операций обмена с памятью.

Определение процессора см. в главе 1.1.

*Определение 1.2.3.*

Сопроцессор — это специализированный процессор, расширяющий возможности центрального процессора компьютерной системы, но оформленный как отдельный функциональный модуль. Физически сопроцессор может быть обособленной микросхемой или может быть встроен в центральный процессор.

Информация, хранимая во внешних устройствах памяти большой емкости, таких как накопители на магнитных дисках и лентах, организована в виде блоков размером единицы и более килобайт. Для обмена данными между указанными устройствами памяти и основной (оперативной) памятью микропроцессора не подходят ни программный способ обмена, ни прерывания (см. *Определение 2.3.1*). Это обусловлено тем, что обмен производится блоками фиксированного размера в строгой последовательности, соответствующей расположению информации на магнитном носителе. Время на обмен одного байта данных строго фиксировано, ограничено скоростью передвижения носителя относительно магнитных головок и составляет весьма малое значение (единицы микросекунд и меньше). При программно-управляемом обмене и обмене с использованием прерываний на передачу байта данных затрачивается большее время. Для обмена данными в указанных условиях организуется прямой доступ к памяти (ПДП), или Direct Memory Access (DMA).

*Определение 1.2.4.*

Канал прямого доступа к памяти — средства, позволяющие осуществить быстрый обмен данными непосредственно между основной памятью и внешним устройством (ВУ) без участия процессора. При этом способе обмена процедура ввода/вывода полностью осуществляется аппаратными средствами и возлагается на микроконтроллер ПДП.

Обычный обмен между внешним устройством и памятью реализуется за два командных цикла: вначале данные поступают от источника в центральный процессор, а затем — из [процессора](http://pue8.ru/protsessory/665-funktsii-i-sostav-protsessora-vosmi-shestnadtsatirazryadnyj-protsessor-soprotsessor.html) в приемник. При ПДП данные не проходят через [процессор](http://pue8.ru/protsessory/665-funktsii-i-sostav-protsessora-vosmi-shestnadtsatirazryadnyj-protsessor-soprotsessor.html), и передача слова производится за один цикл. Поэтому основное достоинство обмена по каналу ПДП — высокая скорость обмена, ограниченная только временем доступа к памяти.

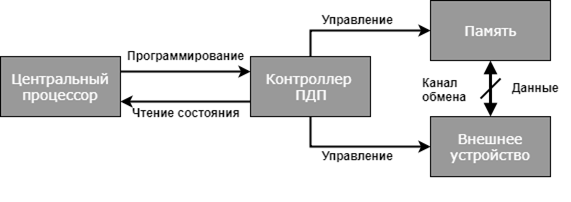
****

Рис. 1.2.2.1. Общий принцип организации канала прямого доступа к памяти

Принципы организации ПДП.

Структура координации канала ПДП показана на Рис. 1.2.2.1. Прямой доступ к памяти открывается по истечении работы данного машинного цикла [процессора](http://pue8.ru/protsessory/665-funktsii-i-sostav-protsessora-vosmi-shestnadtsatirazryadnyj-protsessor-soprotsessor.html). Отличаясь от прерывания (см. *Определение 2.3.1*) обмен по каналу ПДП проводится без участия программы, благодаря чему данные в работающих регистрах процессора сохраняются, и вхождение в режим ПДП не нуждается в расходе времени (нет потребности хранить в стеке данные работающих регистров процессора).

Центральный [процессор](http://pue8.ru/protsessory/665-funktsii-i-sostav-protsessora-vosmi-shestnadtsatirazryadnyj-protsessor-soprotsessor.html) осуществляет программирование микроконтроллера, налаживая его на определенный режим работы, и контролируя состояние. В процессе обмена данными по каналу ПДП процессор не активен, а микроконтроллер формирует сигналы управления обменом для памяти и внешнего устройства. Контакт внешнего устройства с памятью осуществляется по шинам адреса и данных системного интерфейса.

Сложность общего использования шин центрального процессора и внешнего устройства устраняется построением режима обмена с «захватом цикла» или (пакетного, или непрерывного) режима с блокировкой центрального процессора.

Для первого варианта решения обмен производится единичными передачами, тогда как для прямого доступа к памяти применяются отдельные циклы (такты), другими словами, передача данных (слов) чередуется с выполнением программы.

Одна из разновидностей обмена с «захватом цикла» заключается в применении таких тактов, в которых центральный процессор не меняется данными с памятью. При этом микроконтроллер ПДП должен знать эти такты. Некоторые [процессоры](http://pue8.ru/protsessory/665-funktsii-i-sostav-protsessora-vosmi-shestnadtsatirazryadnyj-protsessor-soprotsessor.html) формируют дополнительный сигнал, определяющий обращается ли процессор в текущем цикле к памяти. Использование такого варианта построения обмена не сокращает продуктивность [процессоров](http://pue8.ru/protsessory/665-funktsii-i-sostav-protsessora-vosmi-shestnadtsatirazryadnyj-protsessor-soprotsessor.html), но тем не менее:

● вызывает вспомогательные аппаратные затраты и допускает реализацию исключительно произвольных, нерегулярных передач;

● скорость обмена будет не быстрой, а темп нерегулярен, поскольку продолжительности циклов неодинаковых команд отличаются, к тому же, прямой доступ, возможно, отсрочит время завершения программы, если цикл ПДП не превосходит интервал, соответствующий такту процессора.

Следующая разновидность способа с «захватом цикла» встречается чаще. Здесь центральный процессор принудительно отключается от системных шин адреса и данных. Его осуществление связано с включением двух линий для передачи сигналов запроса на захват шин (ЗЗхв) и подтверждения захвата (ПЗхв). Сигнал ЗЗхв вырабатывается микроконтроллером ПДП.

После приема сигнала ЗЗхв процессор:

● прекращает на некоторое время выполнение текущей команды, не дожидаясь ее завершения;

● передает в системный интерфейс сигнал подтверждения захвата;

● отсоединяется от шин адреса и данных, переводя в высокоомное состояние шинные формирователи.

После приема сигнала ПЗхв микроконтроллер ПДП обращается к шинам системного интерфейса для обмена байтом или словом между ВУ и памятью. После этого определяет сигнал запроса на захват и передает управление шинами центральному процессору. Подготовив следующий байт или слово данных, микроконтроллер ПДП снова передает сигнал ЗЗхв [процессору](http://pue8.ru/protsessory/665-funktsii-i-sostav-protsessora-vosmi-shestnadtsatirazryadnyj-protsessor-soprotsessor.html) и т. д.

Как говорилось ранее, режим ПДП не нуждается в удержании информации о состоянии регистров процессора в стеке. Из-за этого передача данных с «захватом цикла» выполняется с более высокой скоростью, чем при обмене в режиме прерываний (см. *Определение 2.3.1*).

Для обмена данными по каналу ПДП необходима предварительная подготовка микроконтроллера, заключающаяся в установлении программы загрузки требуемых параметров для передачи:

● число передаваемых байтов (слов) данных;

● исходный адрес передаваемых данных (адрес первого байта или слова);

● направление передачи (запись/чтение).

Для работы с перечисленными параметрами в микроконтроллере рассчитаны регистр адреса и счетчик байтов (слов).

1.2.3. Энергонезависимая память

*Определение 1.2.5.*

Энергонезависимая память — это компьютерная память, которая может хранить информацию при отсутствии питания (флеш-память, жесткий диск и т.д.).

*Определение 1.2.6.*

Постоянное запоминающее устройство (ПЗУ) — энергонезависимая память, используемая для хранения массива неизменяемых данных.

Она включает в себя все виды памяти на ПЗУ (*ROM, Read-Only Memory*) и ППЗУ (*EPROM, Erasable Programmable Read-Only Memory*). Это является главным преимуществом энергонезависимой памяти над оперативной, которая носит временный характер, т.е. при выключении компьютера все данные, с которыми работал пользователь, исчезают. Кроме того, максимальная емкость оперативной памяти составляет всего несколько Гбайт, тогда как объем энергонезависимой памяти может достигать более одного Тбайта (пример - жесткий диск), но при этом, как правило, значительно уступает оперативной памяти в быстродействии.

*Определение 1.2.7.*

Флэш-память — тип долговременной памяти, информацию которой можно удалить или перепрограммировать электрическим методом.

*Определение 1.2.8.*

Разрядность процессора — это число битов, обрабатываемых процессором одновременно, т.е. скорость. Процессор может быть разных разрядов: 8-, 16-, 32- и 64-разрядным и т.д. (разрядность с 128 по 512 — это возможные разрядности шины памяти видеокарт). Вместе с быстродействием, разрядность характеризует объем информации, обрабатываемой процессором за определенный промежуток времени. Разрядность вычислителя зависит от разрядности шин. Современные технологии пытаются прийти к концепции: шина быстрее, чем процессор.

Чем больше разрядность, тем медленнее, но точнее процессор будет считать. Эффективность зависит от поставленной задачи.

[Понятие о масштабируемой разрядности](#1302m92).

Есть регистры α, β, γ, δ каждый — 32 бита, если обычные типы long и unsigned, то мы этими регистрами оперируем независимо, но оказывается возможным оперировать парами регистров, когда они выстроены друг за другом. Регистры полноценно работают с 64 разрядом, а есть такие, которые работают по четверкам — α, β, γ, δ (соотв. разрядность 128, Рис.1.2.3). Как итог: масштабируемая разрядность это — использование «раздельных» регистров вместе — над парами или четверками осуществляется одна и та же операция, понимая, что справа — младшие разряды, а слева — старшие.

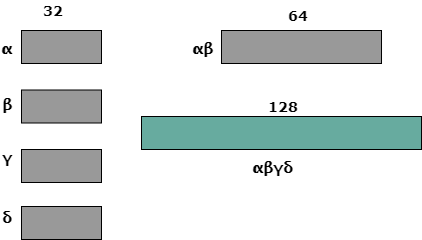
****

Рис. 1.2.3.1. Масштабируемая разрядность

Использование флэш-памяти в качестве мобильного носителя информации или в качестве хранилища программного обеспечения цифровых устройств — это два определяющих способа применения флэш-памяти.

Элементарная ячейка хранения данных флэш-памяти представляет собой транзистор с плавающим затвором, особенность которого состоит в том, что он умеет удерживать электроны (заряд). Именно на различном методе соединения таких ячеек разработаны основные типы флэш-памяти — NOR и NAND.

Архитектура NOR получила название благодаря логической операции ИЛИ–НЕ (англ. NOT OR), а NAND И–НЕ (англ. NOT AND).  
 Если под операндами понимать значения ячеек памяти, то в архитектуре NOR единичное значение на битовой линии будет наблюдаться только в том случае, когда значение всех ячеек, подключенных к данной битовой линии, равно нулю т.е. все транзисторы закрыты (Рис.1.2.3.2). Тогда архитектура NAND подразумевает, что битовая линия имеет нулевое значение в случае, когда все подсоединенные к ней транзисторы открыты, и единичное значение – когда хотя бы один из транзисторов закрыт (Рис.1.2.3.3).

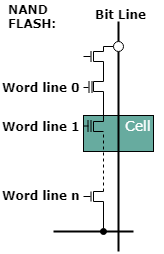
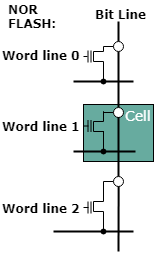
**** ****

Рис. 1.2.3.2. NAND флэш-память Рис.1.2.3.3. NOR флэш-память

Требования хранения больших объемов и компактности однозначно определяют технологию используемой флэш-памяти. Кроме того, возможность выполнять в памяти записанный программный код — XIP Capability (*XIP, eXecute In Place* – мгновенная запись) является не менее важным критерием выбора.

Такая возможность существует у NOR–технологии и отсутствует у NAND. А плотность записи данных для технологии NAND порядком превосходят достигнутые в NOR.

Таким образом, основным назначением памяти, произведенной по технологии NAND, является хранение данных, а по технологии NOR — хранение исполнимого программного кода и, в меньшей степени, данных.

Более подробно о строении этих архитектур написано в источнике <https://ru.bmstu.wiki/%D0%A4%D0%BB%D0%B5%D1%88-%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C>

Особенности флеш-памяти

* Количество попыток на запись в такую память ограничено (предел - миллион раз), в то время как чтение флеш-памяти бесконечно.
* Карты флеш-памяти компактны и гарантируют сохранность данных при их использовании в мобильных устройствах.
* Карты флеш-памяти обладают высокой плотностью упаковки и низким энергопотреблением.
* Флэш-память имеет особую технологию организации запоминающих ячеек. Ячейки сформированы в матрицу. Любая ячейка состоит из одного полевого транзистора. Разрядность данных внешнего интерфейса — 8 (16) бит. Стертые ячейки содержат единицу во всех битах. В процессе записи необходимые биты зануляются. Стирание реализуется для всей матрицы ячеек, иными словами, стирание одиночной ячейки невозможно, что влечет увеличение производительности в режиме записи.
* Алгоритм чтения флэш-памяти: подается адрес ячейки, затем через определенное время доступа на выходе получаем данные. Алгоритм записи данных: для программирования каждого байта приходится выполнять операции записи и считывания, которые адресованы к микросхеме флеш-памяти. Таким образом, в устройстве с флэш-памятью возможно перепрограммирование (перезапись) без извлечения микросхем из устройства.
* Время доступа флеш-памяти при чтении данных составляет 35-200 нс. Стирание информации у микросхем занимает меньше 1-2 секунд. На случай ошибочной записи (стирания) существуют методы аппаратной и программной защиты. Под программной защитой понимается последовательность команд, в случае ее нарушения операции стирания и записи не начнутся. Аппаратная защита не позволяет стирание и запись, если на входы не поданы необходимые уровни напряжения.

1.2.4. Оперативная память

*Определение 1.2.9.*

Оперативная память (оперативное запоминающее устройство – ОЗУ или RAM, *Random Access Memory*) — память с произвольным доступом, область временного хранения данных, обеспечивающая функционирование программного обеспечения, т.е. это совокупность ячеек для хранения данных – байтов, каждый из которых имеет свой адрес, позволяющий обратиться к любому байту памяти.

Функции ОЗУ:

* Хранение, обработка данных и команд.
* Хранение модифицированных обновленных данных.
* Программирование и считывание ячеек.

Ключевые свойства ОЗУ:

* Оперативная память является энергозависимой, иначе говоря при отключении питания все данные на ней удаляются.
* Оперативная память напрямую считывает/записывает данные из любой ячейки ОЗУ, не затрагивая при этом остальные.

Существует 2 типа памяти, возможных к применению в качестве ОЗУ в компьютере:

* SRAM (Static random access memory– память статического типа) — хранит записанные данные до тех пор, пока не появятся новые или не выключат питание;
* DRAM (Dynamiс random access memory – память динамического типа) — хранит данные лишь ограниченное время, после чего данные необходимо регенерировать, в противном случае они будут потеряны.

1.2.5. SRAM

Основное преимущество памяти статического типа – высокая скорость чтения и записи. Это достигается ввиду минимального времени доступа микросхемы SRAM (не больше 2 нс) и возможностью синхронно работать с процессорами. К плюсам статической памяти также следует отнести низкое энергопотребление, простое устройство интерфейса и надежность в использовании.  
Область применения — кэширование информации в процессоре и других устройствах (например, в регистрах процессора).

SRAM содержит совокупность триггеров (переключателей, которые могут находиться в одном из двух состояний: логический ноль или единица), каждый из которых содержит 1 бит данных. Как ячейки DRAM, триггеры соединяются в матрицу, которая состоит из строк (*row*), и столбцов (*column*), которые, в свою очередь, называются битами (*bit*).

Главный недостаток статической памяти – низкая плотность хранения данных (6-8 элементов на один бит, тогда как у DRAM – 2). Таким образом, микросхемы статической памяти имеют бо́льшие габариты, из-за чего значительно возрастает стоимость за килобайт памяти.

Отмечают следующие типы статической памяти:

* Синхронная (одновременно производятся все операции только по тактовому сигналу, в связи с этим скорость обработки ограничивается тактовой частотой).

Именно на данном типе SRAM в современных процессорах строится кэш первого уровня.

* Асинхронная (чтение и запись выполняются по фронту сигнала данных, другими словами, контроль совершается поступающими данными).

Отличается быстрым доступом к информации.

1.2.6. DRAM

Оперативная память ПК, которая предоставляется программе в процессе ее работы, исключая сегмент данных, стек и тело программы. Динамическое размещение данных определяет применение динамической памяти в процессе работы программы. При таком размещении предварительно не известны ни тип, ни число данных, также к ним невозможно обратиться по именам, будто к статическим переменным.

DRAM обладает большой емкостью и стоимостью порядком ниже SRAM, но при этом она намного медленнее. Так как динамическая память производится на базе конденсаторов малой емкости, быстро теряющих заряд, из-за чего, чтобы не потерять хранимые данные, конденсаторы нужно подзаряжать через установленные промежутки времени. Такой процесс называется регенерацией памяти и значительно замедляет работу системы, т.к. во время него невозможно обмениваться данными с памятью. Регенерация осуществляется специальным микроконтроллером, установленным либо на кристалле центрального процессора, либо на материнской плате.

*Определение 1.2.10.*

Виртуальная память компьютера – метод управления памятью, отображающий используемые программами виртуальные адреса в физические адреса в памяти ПК.

Основная память является набором смежных сегментов.

Операционная система управляет виртуальными адресными пространствами и соотносит оперативную память с виртуальной. Программное обеспечение в операционной системе может расширить эти возможности, с целью обеспечения виртуального адресного пространства, которое может превысить объем оперативной памяти и, следовательно, иметь больше памяти, чем есть в компьютере.

Виртуальная память позволяет изменять ресурсы памяти, делать объем оперативной памяти намного больше, для того чтобы пользователь, поместив туда как можно больше программ, сэкономил время и увеличил эффективность своего труда.

*Определение 1.2.11.*

Ассоциативная память или ассоциативное запоминающее устройство — память, адресация в которой определяется содержанием объекта (автоматический поиск производится не по расположению данных), т.е. результатом поиска будут все адреса, где встречаются такие же данные.   
 Этот особый вид машинной памяти используется в приложениях очень быстрого поиска.

*Определение 1.2.12.*

Порт — это какое-либо средство ввода/вывода, отображенное в адресное пространство памяти.

По адресу порта вывода записываются данные, которые отправились через порт с определенного устройства вывода. По адресу порта ввода считываются данные, принятые с устройства ввода. Так осуществляется работа с некоторыми устройствами ввода/вывода (например, с клавиатурой, сетевой картой). Они вписаны в адресное пространство памяти, но при этом сами по себе не являются памятью, а только устройствами обмена информации.

Порты ввода/вывода соединяют два устройства. Однако часто целесообразно подключить к одному порту передачи данных несколько устройств, причем необязательно однотипных.

Системная шина (см. *Определение 2.2.8.)* состоит из трех частей:

* Шина адреса (см. *Определение 2.6.10*)
* Шина данных, по которой, собственно, и будет передана необходимая информация;
* Шина управления (см. *Определение 2.6.12*)

Частный случай шины – шина с двумя сторонами, которая называется мостом. Она удобна тем, что на ней нет адресации.

*Определение 1.2.13.*

Периферийные устройства — аппаратура, которая позволяет вводить [информацию](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%86%D0%B8%D1%8F) в [компьютер](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80) или выводить ее из него[.](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B8%D1%84%D0%B5%D1%80%D0%B8%D0%B9%D0%BD%D0%BE%D0%B5_%D1%83%D1%81%D1%82%D1%80%D0%BE%D0%B9%D1%81%D1%82%D0%B2%D0%BE#cite_note-1)

Взаимодействие периферийных устройств с адаптером происходит через порты ввода/вывода.

По способу передачи информации порты ввода/вывода делятся на:

* Последовательные – информация передается по одному биту, последовательно бит за битом; для передачи информации используется один провод. Подключаются внешние устройства, находящиеся на расстоянии от компьютера.
* Параллельные – несколько битов информации передается одновременно; для передачи информации используется несколько проводов. Подключаются устройства, находящиеся рядом с компьютером.

1.2.7. Системы электропитания и энергосбережения

Наиболее важную часть средств вычислительной техники (СВТ) составляет источник питания, основное назначение которого — снабжать напряжением с заданными параметрами и качеством все узлы и составные блоки устройства вычислительной техники. Качество его функционирования в значительной степени определяет работу СВТ.

Потребление энергии в разных режимах работы ЭВМ

Выделяют следующие основные состояния «системы в целом»:

* Обычный – S0: Все компоненты компьютера потребляют столько энергии, сколько им нужно.
* Сон – S3: Возобновление работы с компьютером с прежнего места, происходит значительно быстрее, чем из режима гибернации (см. *Определение 1.2.13*).

В состоянии сна питание подаётся на ОЗУ и материнскую плату. Выключаются периферийные устройства, а на процессор, оперативную память, беспроводные модули и прочие компоненты в таком режиме подается минимальное напряжение. Выход компьютера из режима сна в рабочий режим практически мгновенный, где рабочий стол со всеми запущенными программами возвращается в том состоянии, в котором они пребывали перед переходом в сон. Содержимое ОЗУ полностью сохраняется, включая кэшированные данные (*Prefetch* и *Superfetch*).

Режим сна весьма удобен для повседневного использования на десктопе. SSD или HDD диск – значения не имеет, компьютер включается мгновенно. Так как в ОЗУ сохраняются кэшированные данные, то повторный запуск приложений очень быстрый относительно холодного старта компьютера, что опять же нивелирует разницу в скорости между SSD и HDD.

Такое потребление энергии очень настолько мало, что даже Wi-Fi роутер, VoIP адаптер и телевизор в режиме ожидания потребляют больше, по 4-5 Вт каждый.

* Гибернация – S4

*Определение 1.2.13.*

Гибернация — это технология, которая позволяет «выключить» компьютер, при этом не закрывая открытые приложения и не теряя несохраненные результаты работы. После восстановления работы компьютера из гибернации, вы можете продолжить работу с того места, где вы закончили. При этом, находясь в режиме гибернации, компьютер не потребляет электроэнергию.

Компьютер обесточивается полностью (остается только питание на кварцевом генераторе для поддержания даты/времени), таким образом, это самый энергосберегающий режим. В нём всё содержимое оперативной памяти сохраняется в энергонезависимой памяти, такой, как жёсткий диск: состояние операционной системы, всех приложений, открытых документов и т. д. Это означает, что на жесткий диск записывается точная копия оперативной памяти компьютера в момент выключения и при включении компьютера этот образ восстанавливается с жесткого диска обратно в оперативную память, аналогично режиму S3 (сон). Иными словами, компьютер запоминает состояние системы перед уходом в режим гибернации, а затем восстанавливает все вкладки и настройки, какие были.

Различие между S4 и S3, кроме дополнительного времени на перемещение содержимого оперативной памяти на диск и назад, в том, что перебои с питанием компьютера в S3 приведут к потере всех данных в оперативной памяти, включая все несохраненные документы, в то время как компьютер в S4 этому не подвержен.

В следующей таблице кратко описаны вышеперечисленные состояния энергопотребления.

|  |  |  |  |
| --- | --- | --- | --- |
| Состояние | Энергопотребление | Возобновление работы ПО | Аппаратная задержка |
| S0 | Максимальное | - | - |
| S1 | S1<S1≤S0 | Система возобновляет работу с прерванной точки (в S0) | <2sec |
| S2 | S3<S2≤S1 | Система возобновляет работу с прерванной точки (в S0) | ≥2sec |
| S3 | S3≤S2 | Система возобновляет работу с прерванной точки (в S0) | ≥2sec |
| S4 | Ток подается только на кнопку включения электропитания, происходит активация электроники | Система перезапускается из сохраненного файла гибернации и возобновляет работу с прерванной точки (в S0) | Длительная и неопределенная |
| S5 | Ток подается только на кнопку включения электропитания | Система загружается «с нуля» | Длительная и неопределенная |

Таблица 1.2.7.1. Состояния энергопотребления

Охлаждение

* Система воздушного охлаждения
* Пассивная

Составляющие системы: радиатор.

Принцип работы: тепло от нагревающегося элемента передается на радиатор за счет теплопроводности материала или при помощи тепловых трубок, далее радиатор рассеивает тепло в окружающее пространство. Эффективность работы радиатора зависит от площади поверхности и материала изготовления.

Достоинства: экономия, надежная работа, безопасность, отсутствие шума.

Недостатки: низкая эффективность для современного оборудования.

* Активная

Составляющие системы: вентилятор, радиатор, который очень часто для эффективности работы добавляется.

Принцип работы:горячий воздух отводится за пределы системного блока. Обычно устанавливают один или несколько вентиляторов, которые производят циркуляцию воздушного потока от передней стенки корпуса к задней.

Достоинства:низкая стоимость, простота в установке и обслуживании.

Недостатки:основной источник шума в компьютере, низкие, по сравнению с другими системами, показатели эффективности.

* Система жидкостного охлаждения

Составляющие системы: помпа, теплосъемник, радиатор, резервуар с рабочей жидкостью, тепловые трубки, датчик потока жидкости

Принцип работы: тепло передается от нагревающегося компонента к радиатору при помощи рабочей жидкости, которая циркулирует в системе. Обычно в качестве рабочей жидкости применяется дистиллированная вода с добавками, обладающими бактерицидный или антигальванический эффект.

Достоинства:почти бесшумная работа (слышен шум журчания воды), высокая эффективность охлаждения.

Недостатки:высокая стоимость, сложность установки, большой размер системы, высокая вероятность повреждения ряда ключевых компьютерных компонентов при выходе из строя.

1.3. Вычислители с множественными потоками команд и одиночным потоком данных (MISD), истинно параллельные процессоры (MIMD), систолические вычислители.

*Определение 1.3.1.*

MISD (англ. *Multiple Instruction stream, Single Data stream*) — это тип архитектуры параллельных вычислений (см. *Определение 4.1.1*), в котором несколько функциональных модулей (два или более) совершают разные операции над одними данными.

*Определение 1.3.2.*

MIMD (англ. *Multiple Instruction stream, Multiple Data stream*) — это многовариантный поток команд и данных, который включает некоторое число независимых процессоров, асинхронно выполняющих различные команды над различными данными. Таких машин очень мало. В эту категорию попадают большинство параллельных процессоров. К MIMD–машинам относятся и мультипроцессоры (машины с общей памятью), и мультикомпьютеры (машины с обменом сообщениями).

*Определение 1.3.3.*

SIMD (*Single Instruction Stream & Multiple Data Stream*) или ОКМД (Одиночный поток Команд и Множественный поток Данных) — архитектура, в которой есть возможность выполнять одну арифметическую операцию сразу над многими данными – элементами вектора.

*Определение 1.3.4.*

AVX (*Advanced Vector Extensions*) *—* это набор SIMD инструкций, функций и схем кодирования для x86-процессоров Intel, который повышает производительность рабочих нагрузок.

*Определение 1.3.5.*

SSE (*Streaming SIMD Extensions*) *—* ранняя версия AVX, которая полностью ее заменяет, поэтому в новых разработках SSE не используется. MMX *(Multimedia Extensions)* это ранняя (20-тилетней давности) попытка сделать расширенный набор команд, в том числе и для векторов. SSE – ее развитие, а AVX это современная версия (AVX2 или AVX512).

AVX расширяет 128-битные регистры SSE до 256 бит (ymm0 - ymm15), при этом 128-битные SSE регистры ссылаются на младшие 128 бит соответствующего AVX регистра.

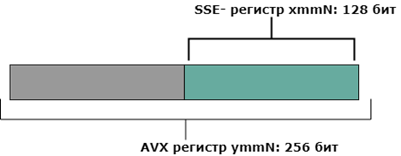
****

Рис. 1.3.1. AVX и SSE регистры

Используется для интенсивных вычислений, увеличивает производительность с вещественными числами и там, где высокая степень параллелизма.

Улучшения AVX:

* Ширина векторных регистров SIMD (обеспечивает параллелизм на уровне данных) увеличивается с 128 до 256 бит. Существующие 128-битные SSE-инструкции будут использовать младшую половину новых YMM регистров, не изменяя старшую часть. Для работы с YMM-регистрами добавлены новые 256-битные AVX-инструкции. Возможно расширение векторных регистров SIMD до 1024 бит, на данный момент реализовано 512-битные версии.
* Неразрушающие операции. Набор AVX-инструкций использует трехоперандный синтаксис. К примеру, вместо m = n + m можно использовать s = n + m, при этом регистр m – неизмененный. Когда значение m используется дальше в вычислениях, это избавляет от необходимости сохранять перед вычислением и восстанавливать после вычисления регистр, содержавший m, из другого регистра или памяти.

AVX – это очень длинный регистр, в который можно положить одно 512-разрядное число или два независимо работающих числа, каждое из которых 256-разрядное. Помножив два таких регистра, получится заготовка для скалярного произведения. Потом можно разделить пополам и получить четыре 128-разрядных числа; и так далее вплоть до char (8 бит). Тогда результатом будет 64 числа — столько элементов размером 1 char можно хранить в одном регистре, который будет работать как вектор. Операция, совершаемая над этим регистром, выполняется независимо над каждым кусочком в один машинный такт. Это значительно повышает производительность для интенсивных вычислений с плавающей точкой.  
AVX специализируется на работе с числами с плавающей точкой типа float и double.

*Определение 1.3.6.*

Систолические вычислительные системы  — это системы класса SIMD, основным принципом которых является то, что все данные, регулярно и ритмически проходящие через массив, используются многократно. Это позволяет значительно повысить эффективность и достичь высокой вычислительной производительности за счет распараллеливания вычислений и сокращения обмена систолической системы с внешними устройствами.

В фон-неймановских машинах данные, считанные из памяти, однократно обрабатываются в процессорном элементе (ПЭ), после снова возвращаются в память. Идея систолической матрицы: организация вычисления производится таким образом, что данные на своем пути от считывания из памяти до возвращения обратно пропускались через наибольшее число ПЭ.

Проведём аналогию между структурами вычислительной системы и живого организма: памяти можно отвести роль сердца, множеству ПЭ - роль тканей, а поток данных рассматриваем как циркулирующую кровь. Отсюда и название систолическая матрица (систола – сокращение предсердий и желудочков сердца при котором кровь нагнетается в артерии). Систолические структуры эффективны при выполнении матричных вычислений, обработке сигналов, сортировке данных и т.д.

В качестве примера возьмем линейный массив для алгоритма матричного умножения. В основе схемы лежит прохождение двух потоков данных навстречу друг другу. Последовательные элементы каждого потока разделены одним тактовым периодом, для того чтобы каждый из них мог пересечься с любым элементом встречного потока. Вычисления выполняются параллельно в процессорных элементах, каждый из которых реализует один шаг в операции вычисления скалярного произведения (*IPS, Inner Product Step*) и носит название IPS-элемента.

Таким образом, систолическая структура - это однородная вычислительная среда из процессорных элементов, которая совмещает свойства конвейерной и матричной обработки и обладает свойствами:

* вычислительный процесс в систолических структурах есть непрерывная и регулярная передача данных от одного ПЭ к другому без запоминания промежуточных результатов вычисления;
* каждый элемент входных данных выбирается из памяти однократно и используется необходимое количество раз по алгоритму, ввод данных осуществляется в крайние ПЭ матрицы;
* образующие систолическую структуру ПЭ однотипны и каждый из них может быть менее универсальным, чем процессоры обычных многопроцессорных систем. Тип ПЭ выбирается в соответствии с назначением систолической матрицы и структурой пространственных связей (наиболее распространены процессорные элементы, ориентированные на умножение с накоплением);
* потоки данных и управляющих сигналов обладают регулярностью, что позволяет объединять ПЭ локальными связями минимальной длины;
* алгоритмы функционирования совмещают параллелизм с конвейерной обработкой данных;
* улучшение производительности матрицы осуществляется за счет добавления в нее определенного числа ПЭ, при этом коэффициент повышения производительности линеен.

Гибкие архитектуры и дилемма «отказоустойчивость-эффективность»

Рассмотрим несколько примеров гибких архитектур.

Машины сгруппированы в самостоятельный блок, на котором стоит процессор, память и все, что нужно для общения с шиной. Из внешних микроконтроллеров есть только контроллер шины. Существует большая вероятность того, что за час что-то выйдет из строя. Но пользователь, по возможности, этого заметить не должен. Есть встроенные аппаратные средства диагностики, которые позволяют выполнять сканирование системы, исследуя аппаратные компоненты и программные конфигурации на наличие и устранение проблем. В это же время от аппаратных средств в ОС идет сигнал о выходе из строя одного из вычислителей. ОС анализирует какие задания этот вычислитель не успел посчитать (хотя получил). Эти задания до их исполнения хранятся в этой иерархии серверов, распределяющих задачи. Затем, чтобы примерно выровнять нагрузку на все компьютеры, планировщик распределяет эти задачи. Таким образом пользователь особо ничего не заподозрит, несмотря на то, что задержка будет более чем двойная.

Другой пример, облачные системы – это системы распределенного способа обработки и хранения данных, автоматически решают вопрос, какие данные использовать. Google позволяет через интернет обучать нейросети, используя их удаленные процессоры.

Архитектура систем сотовых связей тоже есть гибкая архитектура. Система сотовой связи допускает большое разнообразие вариантов конфигураций по набору выполняемых функций. Сложность и гибкость системы можно понять из того факта, что она может обеспечивать передачу не только речи, но и других видов информации, например, текстовых сообщений и компьютерных данных. В части передачи речи может быть реализована обычная двусторонняя телефонная связь, многосторонняя телефонная связь, голосовая почта. При организации обычного двустороннего телефонного разговора, который начинается с вызова, возможны режимы автодозвона, ожидания вызова и переадресации вызова.

Также рассмотрим дилемму «отказоустойчивость-эффективность».

Динамическое балансирование используется там, где ответственность за ошибку ниже: когда имеется возможность при выходе какой-либо взаимозаменяемой части ресурса, заменить другими с общим уменьшением производительности. Т.е. есть статическое в этом смысле распараллеливание, есть динамическая балансировка нагрузки.

Статическое распараллеливание используется при жесткой системе, т.е. вероятность ошибки должна стремиться к нулю. Кроме того, при таком решении задачи центральный компьютер может считать вероятность того, что найденное решение правильное.

1.4. Специализированные вычислители.

*Определение 1.4.1.*

Специализированные вычислители — компьютеры, которые предназначены для узкого класса задач или для конкретной задачи, в соответствии с архитектурой и набором команд.

При проектировании приходится пренебрегать некоторыми характеристиками, например: размерами компьютера, производительностью, функциональностью и надежностью, ради выгоды в иных параметрах, таких как: надежность компьютера, стоимость устройства, энергопотребление и тепловыделение. Это необходимо, чтобы компьютер соответствовал поставленным требованиям (техническим, экономическим — общая стоимость, суммируемая из стоимости всех составляющих компьютера).

Пример: Графический процессор в видеокарте или специальные наиболее надежные компьютеры, применяемые вооруженными силами.

К простейшим способам классификации вычислительных устройств относится выявление их способностей, где каждый вычислитель может быть соотнесен с одним из трех следующих типов:

* специализированные устройства, имеющие возможность выполнять единственную функцию;
* устройства специального назначения, выполняющие некоторый ограниченный набор функций;
* устройства общего назначения, используемые сегодня, к которым, как правило, применяется название «компьютер».

*Определение 1.4.2.*

Специализированный процессор — процессор, у которого особенности архитектуры, набора структурных блоков, системы команд или конструктивно-технологического исполнения дают возможность в значительной мере повысить эффективность решения довольно узкого круга специальных задач в сравнении с другими его применениями.

Например, разработка специализированных процессоров для нейросетей. Также активно внедряются процессоры, называемые криптопроцессорами, которые автоматически защищают информацию, т.е. её запаковывают и распаковывают из защищенного вида.

Рассмотрим виды сопроцессоров:

* математические сопроцессоры общего назначения (применяются для ускорения вычислений с плавающей запятой);
* сопроцессоры ввода/вывода, которые разгружают центральный процессор, снимая с него необходимость контроля операций ввода/вывода или увеличивая пространство адресов процессора;
* сопроцессоры, осуществляющие узкоспециализированные вычисления.

Микропроцессор, оперативная память, флеш-память, порты ввода/вывода, таймеры (см. *Определение 2.4.6*), интерфейсы связи — все это заключено в микроконтроллере.

Наличие таких устройств уменьшает нагрузку, приходящуюся на центральный процессор. Благодаря чему значительно увеличивается производительность системы. Суть такой помощи заключается в освобождении микроконтроллером центральных процессоров компьютера от наиболее медленных операций ввода/вывода информации.

*Определение 1.4.3.*

Математический сопроцессор (англ. *floating point unit, FPU*, сопроцессор с плавающей точкой) — это сопроцессор, который расширяет набор команд центрального процессора и обеспечивает его функциональным модулем для работы с числами с плавающей точкой.

Принципиальные отличия математического сопроцессора от обычного процессора заключаются в том, что он выполняет не математические операции, а операции нужные для представления вещественных чисел с плавающей точкой и устроен через стек.

*Определение 1.4.4*

Модуль операций с плавающей точкой — это часть процессора, используемая для выполнения широкого спектра математических операций над вещественными числами.

Модуль операций с плавающей точкой поддерживает работу на уровне примитивов — загрузка, выгрузка вещественного числа или математические операции над ними, которые выполняются одной командой, за счет чего достигается значительное ускорение таких операций.

*Определение 1.4.5.*

Цифровой сигнальный процессор (ЦСП) — процессор, оптимизированный для решения задач обработки электрических (радио, видео, аудио и других) сигналов в цифровой форме, как правило, в режиме реального времени.

Примеры использования ЦСП:

* Музыкальные и речевые синтезаторы;
* Кодирование аудио (видео) потоков;
* Распознавание речи, изображений;
* Управление технологическими процессами.

Математически эти задачи сводятся к поэлементному перемножению элементов многомерных векторов действительных чисел и к последующему суммированию этих произведений. Сигнальные процессоры оптимизированы по быстродействию для выполнения именно таких операций. ЦСП используют специальную архитектуру памяти, позволяющую получать данные и инструкции одновременно (гарвардская архитектура)*.*

*Определение 1.4.6.*

Потоковые процессоры — процессоры, особенностью которых является возможность обработки нескольких наборов данных одной командой. Так, одно и то же действие, например, вычитание и сложение, может быть произведено над многими числами одновременно.

Потоковые процессоры разделяют на отдельные потоковые (*Single-streaming processor – SSP*) и многопотоковые процессоры (*Multi-Streaming Processor – MSP*).

Примером потоковых процессоров можно считать процессоры Intel, начиная с Pentium. Эта технология оптимальна для таких задач как обработка речи, кодирование и декодирование видео- и аудиоданных, создание трехмерной графики и работа с изображениями.

*Определение 1.4.7.*

Графический процессор *(GPU – Graphics Processing Unit*) — процессор, основной задачей которого является обработка графики и вычислений с плавающей точкой.

GPU облегчает работу главного процессора. Он создает графику, текстуры, цвета (т.е. создает изображение и выдает его пользователю на экран), а также занимается обработкой графики в формате 2D и 3D. Благодаря GPU компьютеру быстрее и легче удается выполнять важные задачи. Особенность графического процессора состоит в том, что он увеличивает скорость расчета графической информации на максимальном уровне. Его архитектура позволяет обрабатывать визуальную информацию эффективнее, чем центральный CPU компьютера.

*Определение 1.4.8.*

Аналого-цифровой преобразователь*(*АЦП, *англ. Analog-to-digital converter, ADC*) — электронное устройство, преобразующее входной аналоговый сигнал в дискретный код (цифровой номер).

*Определение 1.4.9.*

Эффективная разрядность (*ENOB — Effective Number of Bits*) — параметр АЦП, характеризующий меру его динамического диапазона (точности). Однако существует несколько способов его расчета, которые приводят к разным результатам. Если неизвестно по какой формуле она была вычислена, то этот параметр лучше не использовать, чтобы избежать ошибок.

1.5. Графический процессор

Использование графических процессоров для вычислений (взаимодействие графического и центрального процессоров).

У GPU нет инструментов прямого взаимодействия с оперативной памятью компьютера и с устройствами ввода/вывода (помимо монитора). По этой причине руководство графическим процессором проводится исключительно посредством центрального процессора. Модель взаимодействия центрального и графического процессоров представлено на рис. 1.5.1.

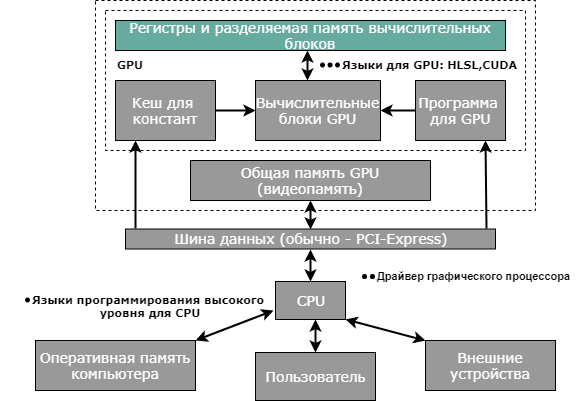
****

Рис. 1.5.1. Модель взаимодействия центрального и графического процессоров

Графические ускорители присоединяются к системной плате ПК посредством высокоскоростной шины данных (в настоящее время PCI-Express). С помощью неё CPU получает доступ к видеопамяти, а также к определенным разделам кэш-памяти, находящейся на графическом процессоре. Посредством шины центральный процессор загружает в графический процессор программу и запускает её.

Прежде чем запустить программу, исполняемую на GPU, центральный процессор посылает графическому следующие данные:

* значения констант, применяемых в программе;
* один, либо несколько крупных массивов данных для потоковой обработки.

Так как к константам нужен регулярный и быстрый доступ, то они вносятся в кэш-память (или регистры), размещенную на кристалле GPU. Массивы данных нередко могут быть столь огромны, что полностью в кэш-память не помещаются, но при элементарной потоковой обработке все элементы массивов данных применяются единственный раз. Таким образом, для хранения этих массивов предназначена видеопамять (общая память), которая представляет собой отдельные микросхемы на плате графического ускорителя, функционирует медленнее кэш-памяти и регистров, однако обладает бόльшим объемом .

Результаты своей работы GPU тут же заносит в область видеопамяти, которая именуется буфером кадра, посылающим их на монитор. Однако имеется допустимость не показывать результаты на экране, а дублировать их из видео-памяти в оперативную память компьютера, откуда они оказываются доступными для последующей обработки центральным процессором. Это основополагающее применение графических процессоров для вычислений общего направления, не связанных с обработкой графики.

Устройство GPU

У графического процессора много ядер, функционирующих преимущественно на низких скоростях. Они занимаются вычислениями пикселей и вершин. Обработка последних в основном происходит в системе координат. GPU обрабатывает различные задачи, создавая на экране трехмерное пространство, это значит, что объекты в нем перемещаются. Каждый из типов графического процессора фильтрует треугольники, входящие в него. Он определяет, какие на виду, удаляет те, которые скрываются за другими объектами, прорисовывает источники света и определяет, каким образом эти источники влияют на цвет.

Существуют два вида графических процессоров:

* Встроенный. Он есть практически во всех компьютерах. Его устанавливают в CPU, чтобы сделать потребление энергии в несколько раз ниже.
* Дискретный. Его устанавливают на отдельные модули, так как он отличается своей мощью, но ему необходимо отличное охлаждение.

1.6. Программная среда специализированного вычислителя

*Определение 1.6.1.*

BIOS (частный случай) (*Basic Input Output System*) *—* базовая система ввода-вывода; набор микропрограмм, реализующих [API](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F_%D0%BF%D1%80%D0%B8%D0%BB%D0%BE%D0%B6%D0%B5%D0%BD%D0%B8%D0%B9) для работы с аппаратурой компьютера и подключенными к нему устройствами. В эту систему входят разные программы ввода-вывода, обеспечивающие связь между ОС, прикладными программами с одной стороны и устройствами, входящими в состав компьютера (внутренними и внешними), с другой.

Существует BIOS для материнских плат и для периферийных устройств.

На сегодняшний день BIOS — это сложная система, которая состоит из значительного числа утилит, предназначенных для автоматического распознавания, установленного на компьютер оборудования, его настройки и контроля функционирования. Вызов программ BIOS обычно осуществляется через программные или аппаратные прерывания (см. *Определение 2.3.3*). При каждом включении питания компьютера типа IBM PC (или совместимого с ним) и до начала загрузки ОС процессор компьютера осуществляет процедуру BIOS под названием «Самотест по включению питания» – POST (*Power On Self Test*). Главной целью процедуры POST является проверка базовых функций и подсистем компьютера (таких как память, процессор, материнская плата, видеоконтроллер, клавиатура, гибкий и жесткий диски и т. д.) перед загрузкой ОС. Это в некоторой степени застраховывает пользователя от попытки работать на поврежденной системе, что могло бы послужить причиной, например, разрушения пользовательских данных на HDD.

BIOS представляется как микросхема, которая установлена на материнской плате компьютера. Обратим внимание, что название ROM BIOS сейчас не совсем справедливо, так как «ROM» подразумевает применение постоянных запоминающих устройств (*Read Only Memory*), а для хранения кодов BIOS в настоящее время используют в основном перепрограммируемые запоминающие устройства. Более перспективной для хранения системы BIOS считается флэш-память. Она дает возможность изменять функции для поддержки новых устройств, подключаемых к компьютеру.

*Определение 1.6.2.*

Firmware (рус. прошивка) *—* специализированное программное обеспечение внутри техники с микроконтроллером. Практически все электронные устройства помимо простейших включают в себя Firmware: компьютерные мыши, клавиатуры, жесткие диски, маршрутизаторы, современные модели автомобилей, телевизор, стиральная машина и т.д.

*Определение 1.6.3.*

Операционные системы реального времени ([*ОСРВ*](http://embedded.prosoft.ru/tags/osrv/) *(real-time operating system – RTOS)*) — операционные системы, которые способны предоставить предсказуемое время обработки неожиданно появляющихся внешних событий. Разделяют ОС «жесткого» и «мягкого» реального времени: для первых временные характеристики гарантированы, и выход за их границы расценивается как отказ, для вторых временные ограничения обычно соблюдаются, и выход за их пределы является снижением производительности. Основная масса современных ОСРВ представляют собой встраиваемые ОС.

Окружение времени выполнения (*run-time environment*) программных модулей - RTE. Существуют обычные ОС, если их упрощать и убирать многие функции, то получается ОСРВ. При убирании почти всех функций получается RTE. RTE - небольшое количество функций, которые уже не являются операционной системой и часто стоят на различных специфичных ЭВС. Могут включать в себя до пары десятков функций. ОСРВ, в отличие от RTE, может иметь какую-то вариативность в действиях, другими словами, универсальность.

*Определение 1.6.4.*

Встраиваемые операционные системы — ОС, которые предназначены для управления специализированными устройствами и, как следствие, умеющие работать в условиях ограниченных ресурсов (небольшие объемы памяти, нехватка вычислительных мощностей и т.п.) и в необслуживаемом режиме. Отличительными свойствами встраиваемых ОС считаются модульная структура, компактность, производительность, масштабируемость и повышенная отказоустойчивость.

ОСРВ и RTE отличаются от обычных ОС областью применения, функциональностью, скоростью работы и отказоустойчивостью. ОСРВ, к примеру, могут не обладать возможностью воспроизводить звук (и кучу других функций), однако могут работать быстрее и надежней при обстоятельствах, в которых ОС общего назначения затормозит. Подобный переход от ОСРВ к RTE представлен в таблице 1.6.1.

|  |  |
| --- | --- |
| ОС (OS – operating system) | Обычная система.  Может использоваться на компьютерах, ноутбуках, некоторых планшетах (Windows, Ubuntu, Android). |
| ОСРВ (RTOS – real time OS) | «Обкусанная» ОС, из которой выброшены все ненужные функции, но при этом сохраняется определенная универсальность (свобода действий, возможность решения широкого круга задач)  Пример использования: мосты, шлюзы, маршрутизаторы (RTOS Linux). |
| ОВВ (RTE – run-time environment) | Это уже нельзя назвать операционной системой, так как она «обкусана» до такой степени, что может решать только строго определенный узкий круг задач. Нет динамического выделения памяти, есть только стек, нет кэша. Нет периферических устройств, кроме портов: с одного мы всегда берем, на другой отдаем. Там нет никаких функций отладки.  Используется, например, в GPS-навигации, сотовых станциях. |

Таблица 1.6.1. Переход от ОСРВ к RTE системе.

Глава 2. Процессор (компьютер)

2.1. Узлы процессора

*Определение 2.1.1.*

Процессор общего назначения — процессор, который способен эффективно решать обширный класс вычислительных задач, а также задач управления и прочих. Именно эти процессоры используются в качестве центральных в настольных рабочих станциях.

Центральный процессор (ЦП) выполняет программы, находящиеся в основной памяти.

*\*определение регистра см. определение 1.1.8.*

Доступ к значениям, хранящимся в регистрах обычно намного быстрее, чем доступ к ячейкам ОЗУ, но объем ОЗУ превосходит суммарный объем регистров

Свойства регистров:

* Регистры нужны для вычислений и связи ЦП с внешним миром.
* Большая часть кода программ состоит из команд копирования значений из оперативной памяти в регистры и обратно.
* Во всех операциях программы без регистров не обойтись.
* Если данные не хранятся в регистрах, то в них обязательно будут указатели на эти данные (адреса данных в памяти).
* C помощью команд машинного языка пользователь может обращаться к доступным регистрам, к которым обычно имеют доступ все программы — и приложения, и системные.

Регистры классифицируются на аккумуляторы, регистры данных, адресные регистры, индексные, регистры общего назначения, регистры команд и флаговые регистры.

*Определение 2.1.2.*

Аккумулятор *—* главный регистр процессора над которым осуществляется подавляющее большинство арифметических и логических побитовых операций.

*Определение 2.1.3.*

Регистры данных *—* регистры, которые используются для записи и хранения промежуточных вычислений.

*Определение 2.1.4.*

Адресные регистры нужны для хранения адреса (или его части) операнда выполняемой программы, они формируют адреса данных и команд.Регистр адреса памяти содержит двоичное число – адрес области памяти. Выход этого регистра называется адресной шиной (см. *Определение 2.6.10*).

*Определение 2.1.5.*

Индексный регистр используется в обычном режиме адресации, когда адресом является результат сложения содержимого индексного и базового регистра.

При сегментной адресации память делится на сегменты, которые состоят из разного числа машинных слов. Из адреса сегмента и смещения относительно начала сегмента складывается адрес ячейки памяти. При таком режиме адресации базовый адрес сегмента (его начало) хранится в одном из регистров. Таких регистров может быть несколько: один – для операционной системы, остальные — для приложений, которые исполняются в данный момент.

Регистры общего назначения используются для запоминания данных и/или операндов при исполнении команд.

В качестве примера рассмотрим регистры для процессоров Intel.

Регистры EAX, EBX, ECX, EDX – регистры общего назначения. Они имеют определенное назначение, но в них можно хранить какую угодно информацию.

Регистры EBP, ESP, ESI, EDI – это тоже регистры общего назначения, они уже используются для конкретных задач. В них также можно хранить пользовательские данные, но делать это нужно осторожнее, чтобы не получить «неожиданный» результат.

*Определение 2.1.6.*

Счетчик команд *—* специальный регистр, в котором хранится адрес команды, которая должна быть выполнена после исполнения текущей команды.

*Определение 2.1.7.*

Регистр указателя стека *—* это 16-разрядныйрегистр, который указывает на положение данных в стеке.

При стековой адресации выделяется специальный регистр, размещающий указатель на вершину стека. Такой режим адресации позволяет использовать некоторые команды, в которых отсутствует поле адреса.

В состав всех процессоров входит регистр или набор регистров, известный как регистр слова состояния программы (*program status word, PSW*), содержащий коды условий (см. *Определение 2.1.8.*) и другую информацию о состоянии:

|  |  |  |
| --- | --- | --- |
| Символ | Позиция | Имя и назначение |
| P | PSW.0 | Флаг паритета. Устанавливается и сбрасывается аппаратуры в каждом цикле команды и фиксирует нечетное/четное число единичных бит в аккумуляторе, т.е. выполняет контроль по четности |
| - | PSW.1 | Не используется |
| OV | PSW.2 | Флаг переполнения. Устанавливается и сбрасывается аппаратно при выполнении арифметических операций |
| RS0 | PSW.3 | Выбор банка регистров. Устанавливается и сбрасываетсяпрограммой для выбора рабочего банка регистров |
| RS1 | PSW.4 |
| F0 | PSW.5 | Флаг 0. Может быть установлен, сброшен или проверен программой как флаг, специфицируемый пользователем |
| АС | PSW.6 | Флаг вспомогательного переноса. Устанавливается и сбрасывается только аппаратурными средствами при выполнении команд сложения и вычитания, сигнализирует о переносе или заеме в бите 3 |
| С | PSW.7 | Флаг переноса. Устанавливается и сбрасывается аппаратурными средствами или программой при выполнении арифметических и логических операций |

Таблица 2.1.2. Формат слова состояния программы (PSW)

Флаг (см. *Определение 2.3.5*) переноса (С) преимущественно используемый флаг PSW. Он участвует и преобразуется в ходе выполнения множества операций, в число которых входят сложение, вычитание и сдвиги. Помимо этого, флаг переноса осуществляет функции «булевого аккумулятора» в командах, управляющих битами. Флаг переполнения (OV) фиксирует арифметическое переполнение при выполнении операций над целыми числами со знаком и допускает применение арифметики в дополнительных кодах. АЛУ не манипулирует флагами селекции банка регистров (RS0, RS1), их значения целиком формируются прикладной программой и используются с целью выбора одного из четырех регистровых банков.

*Определение 2.1.8.*

Коды условий *—* некоторый порядок битов, которые определяются или сбрасываются процессором, зависящих от результата выполненных операций.

Большая часть компьютеров содержит и внутренние (см. *Определение 1.1.9*), и внешние шины (см. *Определение 1.1.10*).

*Определение 2.1.9.*

Внутренняя шина (локальная) — такая шина, через которую все внутренние составляющие компьютера подключаются к материнской плате.

К ключевым внутренним шинам принадлежат:

* системная шина (см. *Определение 2.2.8.*) с разъемом процессора;
* шина памяти с разъемами модулей памяти;
* шина и слот видеокарты;
* шины и слоты плат расширения;
* шины и порты накопителей;
* шина и разъемы электропитания;
* линии и порты интерфейса управления питанием;
* порты и панели индикации;
* шины и порты управления системой.

*Определение 2.1.10.*

Внешняя шина — шина, с помощью которой внешние устройства (см.ниже) подключаются к материнской плате.

Из числа внешних интерфейсов следует особо отметить группу, которая обслуживает не обязательные компоненты компьютерной системы, такие как монитор, клавиатура, мышь, а дополнительные внешние устройства, объединяемые понятием «периферия».

Стандартными внешними интерфейсами являются:

* порты видеокарты для подключения мониторов и телевизора;
* порты PS/2 для клавиатуры и мыши;
* последовательные порты СОМ;
* параллельные порты LPT;
* последовательный порт FireWire;
* сетевой порт RJ45;
* порт модема RJ11 для телефонной линии;
* порты аудиоинтерфейса.

Однако, не всегда можно определить тип шины: например, шина SATA — внешняя или внутренняя? Диск можно подключить как снаружи, так и внутри. Аналогично шина USB. Также стоит отметить, что на материнских платах довольно много внутренних USB-разъемов, к которым подключаются внутренние устройства, например, картридеры. Устройство внутри, но шина внешняя. Таким образом, шины SATA и USB нельзя назвать определенно внутренними или определенно внешними.

Как работает шина.

Некоторые устройства, которые соединены с шиной, являются активными и способны инициировать передачу информации по шине, другие — пассивные и ожидают запросов. Активное устройство — задающее, пассивное — подчиненное.

Внешние формирователи. Большая часть задающих устройств связана с шиной через микросхему, называемую драйвером шины и являющуюся цифровым усилителем. Большая часть подчиненных устройств связана с шиной через приемник шины.

Для задающих и подчиненных устройств применяется приемопередатчик шины (трансивер).

Регистры вместе с АЛУ формируют тракт данных с поступающими данными. Базовая операция тракта данных выполняется так: выбирается один или два регистра, АЛУ производит над ними операцию, после результат вновь помещается в какой-либо регистр.

АЛУ может выполнять следующие операции:

* логические операции;
* арифметические операции;
* операции циклического сдвига, инвертирования, сброса

Функциональное устройство АЛУ можно разделить на две части (Рис 2.1.1.):

* микропрограммное устройство, создающее последовательность микрокоманд;
* операционное устройство, которое реализует заданную последовательность микрокоманд.

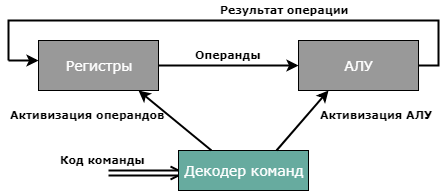
****

Рис 2.1.1. Структура АЛУ

Особенность АЛУ заключается в том, что оно способно оперировать не только байтами, но и битами. Программно-доступные биты могут быть установлены, инвертированы, переданы, сброшены, проверены и использованы в логических операциях. Это важно, потому что при управлении объектами часто применяются алгоритмы, которые содержат операции над выходными булевыми переменными.

Процедура выполнения команд.

Программа, загружаемая в оперативную память компьютера, состоит из машинных команд. Процессор выполняет их в той последовательности, в какой они записаны в программе, чтобы ее реализовать.

Для того чтобы процессор мог знать, какую команду необходимо выполнить в определенный момент, существует счетчик команд.

Счетчик команд работает с памятью, которая находится внутри процессора и называется очередью команд (см. *Определение 2.1.11.*). В счетчике хранятся адреса команд в этой памяти, куда их помещают перед выполнением.

Ряд операций, совершающихся процессором при выполнении одной машинной команды, называется циклом выполнения команды. Параллельно процессор должен произвести выборку, раскодирование и выполнение*.* Если команда содержит операнд, находящийся в оперативной памяти, то процессор вынужден провести две операции: выборку операнда из памяти и запись вычислений в память. Каждое из перечисленных действий охарактеризовано ниже.

* Выборка команды. Блок управления берет команду из памяти, записывает ее во внутреннюю память процессора и повышает значение счетчика команд на длину этой команды.
* Раскодирование команды*.* Блок управления распознает тип выполняемой команды, направляет заданные в ней операнды в АЛУ и вырабатывает электрические сигналы управления АЛУ, соответствующие типу проводимой операции.
* Выборка операндов*.* Если в команде используется операнд, находящийся в оперативной памяти, то блок управления приступает к его выборке из памяти.
* Запись результата в память. Когда результат выполнения команды должен быть сохранен в памяти, блок управления начинает операцию сохранения данных.
* Выполнение команды*.* АЛУ применяет заданную в команде операцию, сохраняет итоги вычислений и модифицирует состояние флагов (см. *Определение 2.3.5*), относительно которых программа делает выводы об итоге выполнения команды.

Обобщим сказанное и составим цикл выполнения команды.

1. Выбрать из очереди команд ту, на которую указывает счетчик команд.
2. Определить адрес следующей команды в очереди и записать ее адрес в счетчик команд.
3. Декодировать команду.
4. Если в команде есть операнды, находящиеся в памяти, то выбрать операнды.
5. Выполнить команду и установить флаги (см. *Определение 2.3.5*).
6. Записать результат в память (по необходимости).
7. Начать осуществление следующей команды с п.1.

*Определение 2.1.11.*

Очередь — это структура, которая разделяет поток данных, где чтение слов происходит в порядке их записи.

*Определение 2.1.12.*

FIFO — сокращенное обозначение метода *first-in, first-out:* первым пришел, первым вышел. Отличается от очереди тем, что не может иметь приоритетов.

Блок предсказания переходов

Эффективность алгоритмов предсказания переходов критична для архитектур, в которых применяется высокий уровень параллелизма инструкций. Такие переходы разрывают параллелизм, потому что результат предыдущей инструкции следует ожидать перед тем, как продолжит выполняться поток инструкций. Предсказание переходов дает возможность узнать, какая будет взята инструкция, и в случае если предсказание верно, вычисляется дальнейший адрес выполнения.

Branch Target Buffer требуется, чтобы сохранять результат ветвлений по мере продолжения выполнения кода. Так, к массиву прилагается алгоритм определений результатов следующего ветвления.

Декодеры преобразуют команды в макрокоманды, стоящие из команд тех программ, под управлением которых в процессоре выполняются простые операции.

Простая инструкция преобразуется в 1–2 микрокоманды — совокупности кодов микроопераций, которые управляются схемами в течение одного машинного такта.

Сложная инструкция из памяти микрокода подбирает последовательность из более чем двух микрокоманд. Применяя технологию macrofusion, четыре декодера готовы обработать одновременно пять команд, преобразуя их в четыре микрокоманды. Данная методика заключается в том, что ряд пар связанных между собой последовательных инструкций представляются декодером одной микрокомандой. Таким образом, технология *macrofusion* дает возможность закодировать одной микрооперацией две команды, чтобы уменьшить нагрузку на конвейер для некоторых сложных случаев.

Помимо основных узлов процессора, необходимы и прочие вспомогательные узлы, которые нужны для обработки данных. Далее рассмотрим некоторые дополнительные возможности процессора.

*Определение 2.1.13.*

Упреждающая выборка команд (*англ. instruction prefetch*)  *—* это метод, используемый компьютерными процессорами для повышения производительности. Инструкции или данные из их исходного хранилища более медленной памяти перемещаются в более быструю локальную память.

*Определение 2.1.14.*

Контроллер прямого доступа к памяти (*англ. Direct Memory Access, DMA*) *—* блок, который позволяет некоторым аппаратным подсистемам получать доступ к основной системной памяти (оперативной памяти) независимо от ЦП.

2.2. Ядро процессора

*Определение 2.2.1.*

Ядро — важнейшая часть процессора, которая содержит все функциональные блоки, а также выполняет все логические и арифметические операции.

Функциональные блоки, содержащиеся в ядре процессора:

* блок выборки инструкций;
* блоки декодирования инструкций;
* блоки выборки данных;
* управляющий блок;
* блоки выполнения инструкций;
* блоки сохранения результатов;
* блок работы с прерываниями (см. *Определение 2.3.1*);
* ПЗУ, содержащее микрокод;
* набор регистров;
* счетчик команд.

Блок выборки инструкций за такт считывает инструкции по адресу, указанному в счетчике команд. Количество считываемых инструкций зависит от количества блоков декодирования, так как на каждом такте работы необходимо максимально загрузить блоки декодирования. Для оптимальной работы блока выборки в ядре процессора существует предсказатель переходов.

Предсказатель переходов старается определить, какая последовательность команд будет выполняться после совершения перехода. Это необходимо для максимальной нагрузки конвейера ядра процессора после условного перехода.

Блоки декодирования выполняют расшифровку инструкций, определяют, какое действие нужно совершить процессору, какие дополнительные данные требуются для выполнения инструкции. Длина инструкций и количество операндов — нефиксированные, поэтому для множества современных коммерческих процессоров, построенных на базе концепции **CISC** (см. *Определение 2.5.2*)**,** процесс декодирования — задача сложная.

Блоки выборки данных выполняют отбор из кэш-памяти или ОЗУ данных, обязательных для реализации текущих инструкций. Каждое ядро процессора имеет несколько блоков выборки данных. К примеру, в процессорах Intel Core используется по два блока.

Управляющий блок, полагаясь на декодированные инструкции, осуществляет управление блоком выполнения инструкций, распределяет нагрузку между ними, а также следит за своевременностью и верностью выполнения инструкций. Это один из важнейших блоков ядра процессора.

Блоки выполнения инструкций состоят из нескольких разнотипных блоков:

* Арифметико-логического устройства (АЛУ);
* FPU;
* Блока сохранения результатов;
* Блока работы с прерываниями (см. *Определение 2.3.1*).

Блок сохранения результатов записывает результат выполнения инструкции в ОЗУ по адресу, который указан в обрабатываемой инструкции.

Этапы цикла работы ядра процессора выглядят следующим образом:

1. Блок выборки инструкций проверяет наличие прерываний. Если таковые есть, то данные регистров и счетчика команд будут записаны в стек, а в счетчик команд будет записан адрес команды обработчика прерываний (см. *Определение 2.3.2.*). После работы функции обработки прерываний данные из стека будут восстановлены;
2. Блок выборки инструкций считывает из счетчика команд адрес команды для выполнения. После чего по найденному адресу считывается команда из кэш-памяти или ОЗУ. Далее извлеченные данные передаются в блок декодирования;
3. Блок декодирования команд осуществляет расшифровывание команд, используя, когда это необходимо, микрокод, который записан в ПЗУ. Если расшифрованная команда является командой перехода, то в счетчик команд записывается адрес перехода и управление передается в блок выборки инструкций, в ином случае счетчик увеличивается на размер команды и осуществляет передачу управления в блок выборки данных;
4. Блок выборки данных считывает из кэш-памяти или ОЗУ данные, нужные для выполнения команды, и передает управление планировщику;
5. Управляющий блок определяет блок выполнения инструкций, который будет обрабатывать текущую задачу и передает управление этому блоку;
6. Блоки выполнения инструкций выполняют действия, которые описаны в программе, и передают управление блоку сохранения результатов;
7. Если необходимо сохранить результаты в ОЗУ, то блок сохранения результатов выполняет нужные для этого действия, после чего передает управление блоку выборки инструкций.

Этот цикл называется процессом, а последовательность исполняемых команд называется программой.

Устройство ядра процессора определяет время, затраченное на выполнение одной инструкции, и время работы каждого этапа цикла. А от тактовой частоты зависит скорость перехода между этапами цикла.

*Определение 2.2.2.*

Многоядерный процессор — центральный процессор, который содержит более одного вычислительного ядра на одном процессорном кристалле или корпусе.

В многоядерных процессорах тактовая частота обычно специально снижена. Это позволяет уменьшить потребление энергии процессором без ухудшения производительности. В некоторых процессорах тактовая частота зависит от его индивидуальной нагрузки каждого из ядер и может изменяться. Ядро это полноценный микропроцессор, который использует все достижения микропроцессорной техники, такие как конвейеры, внеочередное исполнение кода, многоуровневый кэш, поддержка векторных команд.

Каждое ядро также может использовать технологию SMT для поочередного исполнения нескольких потоков, создавая иллюзию нескольких «логических процессоров» на основе каждого ядра.

*Определение 2.2.3.*

Многопоточность(англ. *Multithreading*) — это способность программы или операционной системы позволять текущим потокам сосуществовать независимо друг от друга в контексте процесса. Они выполняются в его адресном пространстве. При этом поток поддерживает список информации, относящейся к его выполнению, например, расписание приоритетов, обработку исключений, набор регистров ЦП и состояние стека.

*Определение 2.2.4.*

Одновременная многопоточность (англ. *Simultaneous Multithreading (SMT*), *hyper-threading*) — технология, позволяющая разбить одно физическое ядро процессора на виртуальные ядра, для того чтобы позволить каждому из них запускать несколько потоков команд в одно и то же время. Как следствие, может повыситься производительность.

*Определение 2.2.5.*

Временная многопоточность (англ. *Temporal multithreading, super-threading*) — ещё один тип многопоточности, при котором переключение между потоками происходит по фиксированному расписанию. При чём в конкретный момент времени обрабатывается только один из потоков команд. Обычно приоритет и время их выполнения определяет ОС. Помимо фактора времени, привести к окончанию исполнения потока может аппаратное прерывание (см. Определение 2.3.1) или системный вызов. После чего процессор сохраняет контекст потока — его состояние — и переключается на состояние следующего. Если поддержка потоков реализована аппаратно, ЦП делает это самостоятельно; иначе ему требуется помощь операционной системы для переключения контекстов.

Временная многопоточность во многом похожа на одновременную многопоточность: аппаратное обеспечение должно хранить полный набор состояний для каждого реализованного параллельного потока, а также поддерживать иллюзию, что текущий поток располагает необходимыми ресурсами процессора. Кроме того, при обоих типах многопоточности должна применяться технология справедливой очереди, чтобы один поток не преобладал над временем или ресурсами процессора. При одновременной многопоточности ЦП выделяет больше тепла, однако временная позволяет выполнять всего лишь один поток сразу.

Гиперпоточность (англ. *hyper-threading technology*, HT или HTT) — технология, созданная компанией Intel, позволяющая ядру процессора осуществлять больше потоков данных, чем один (обычно два). Так как было обнаружено, что обычный процессор в большинстве задач применяет не более 70% всей вычислительной мощности, было решено использовать технологию, допускающую при простоте некоторых вычислительных блоков — загрузить их работой с другим потоком. Это дает возможность повысить производительность ядра от 10 до 80% в зависимости от задачи.

Кэш-память: во всех имеющихся на сегодня многоядерных процессорах кэш-памятью 1-го уровня располагает каждое ядро в отдельности, а кэш-память 2-го уровня имеется в двух вариантах:

* разделяемая — размещенная на одном кристалле с ядрами и открыта каждому из них в полном объеме.
* индивидуальная — отдельные кэши равного объема, интегрированные в каждое из ядер.

Многоядерные процессоры также имеют гомогенную или гетерогенную архитектуру:

* гомогенная архитектура — все ядра процессора одинаковы и реализуют одни и те же задачи
* гетерогенная архитектура — ядра процессора совершают разные задачи.

Значительное большинство современных процессоров владеют двумя и более ядрами. Верхние модели могут вмещать и 8, и даже 12 ядер. Мы, по сути, получаем несколько процессоров, способных самостоятельно решать каждый свои задачи, при этом, естественно, повышается производительность. Однако, прирост производительности далеко не всегда оправдывает ожидания.

Во-первых, далеко не все программы поддерживают распределение вычислений на несколько ядер. Конечно, программы можно разбивать между ядрами, чтобы на каждом ядре работал свой подбор самостоятельных программ. К примеру, на одном ядре работает операционная система с набором служебных программ, на другом пользовательские программы и так далее.

Но это приносит пользу в производительности до тех пор, пока не появляется программа, запрашивающая больше ресурсов, чем может предоставить одно ядро. Хорошо, если она содействует распределению нагрузки между несколькими ядрами.

Во-вторых, осложняется работа с памятью, так как ядер – много, и всем им необходим доступ к ОЗУ. Требуется сложный механизм, распознающий очередность доступа ядер процессора к памяти и к другим ресурсам компьютера.

В-третьих, повышается энергопотребление, а, стало быть, возрастает тепловыделение и необходима мощная система охлаждения.

**Синхронная и асинхронная работа ядер**

Синхронная загрузка отмечает, что полностью одинаковые операции реализуются независимо над различными кусочками данных. Эту обработку можно выполнять последовательно на одном ядре. Иногда полезно отдать различные кусочки всем ядрам для обработки. Все ядра, т.к. у каждого, как например, свое собственное независимое АЛУ, если приступили к обработке различных кусочков по одинаковой операции в один момент времени, то одновременно завершат так же в один момент времени.

Если нужно осуществить множество различных операций, то одновременной работы не получится, т.к. исполняя разные операции с разными данными, ядра оканчивают в разный момент, т.к. для каждой команды надо разное время.

Синхронный режим — когда процессор и память работают на одной частоте шины.

Асинхронный режим — это когда шины процессора и памяти состоят в соотношении 2:3, 4:5 и т.д.

В многоядерных процессорах возникли два новых элемента: синхронизатор и арбитр.

*Определение 2.2.7.*

Арбитр — особое устройство, обеспечивающее доступ с низкой латентностью к интерфейсу FSB.

*Определение 2.2.8.*

Front Side Bus (FSB, системная шина) — шина, позволяющая соединить внутренние устройства машины с x86/x86-64-совместимым центральным процессором.

Арбитр связывается с ядром посредством синхронизатора, специальным служебным элементом, существующем на каждом ядре. Работает это устройство на частоте, которая кратна частоте FSB.

По необходимости можно менять частоту работы ядра и кэша с помощью асинхронного интерфейса между любым ядром и арбитром. Из-за наличия арбитра появляется соответствующая латентность при взаимодействии ядра и системного интерфейса. При иных равных условиях запросы на чтение получают приоритет, как наиболее важные.

*Определение 2.2.9.*

Application-specific instruction-set processor (ASIP) (рус. «проблемно-ориентированный процессор», «заказная микросхема», «заказной микропроцессор») — компонент (обычно это ядро процессора), который используется при проектировании [систем на кристалле](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BD%D0%B0_%D0%BA%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BB%D0%BB%D0%B5).

*Определение 2.2.10.*

Однокристальная система или система на кристалле ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *System-on-a-Chip*) — в [микроэлектронике](https://ru.wikipedia.org/wiki/%D0%9C%D0%B8%D0%BA%D1%80%D0%BE%D1%8D%D0%BB%D0%B5%D0%BA%D1%82%D1%80%D0%BE%D0%BD%D0%B8%D0%BA%D0%B0) — электронная схема, размещенная на одной [интегральной схеме](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%82%D0%B5%D0%B3%D1%80%D0%B0%D0%BB%D1%8C%D0%BD%D0%B0%D1%8F_%D1%81%D1%85%D0%B5%D0%BC%D0%B0), способная выполнять функции полноценного устройства (например, [компьютера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80)).

[Система команд](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0_%D0%BA%D0%BE%D0%BC%D0%B0%D0%BD%D0%B4) ASIP спроектирована специально для выполнения специфичных программ. Такая специализация ядра дает возможность достичь компромисса между универсальностью [процессора](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80) общего назначения (CPU) и производительностью ASIC (см. Список сокращений).

*Определение 2.2.11.*

«Жесткий» процессор — это процессор, в котором набор инструкций задается жестко, то есть каждая машинная инструкция: сложение, сдвиг, копирование – воплощена непосредственно в схеме. От увеличения числа инструкций схема становится сложнее.

В процессорах, использующих микрокод, на схеме реализованы только основные команды, все остальные инструкции реализованы в микропрограммах. Благодаря этому не нужно тратиться на создание сложных схем с реализацией большого множества инструкций процессора. Если при подходе используются сложные инструкции микрокода, его называют CISC (см. *Определение 2.5.2*).

2.3 Прерывания

Работу с прерываниями можно назвать одной из самых важных задач процессора, она позволяет в нужный момент реагировать на события, на время приостанавливать ход работы программы и выполнять требуемые от него действия. С помощью прерываний процессор способен перейти к псевдопараллельной работе, т.е. к так называемой, многозадачности.

*Определение 2.3.1*

Прерывание — сигнал от программного или аппаратного обеспечения, объявляющий процессору, что наступило событие, которое запрашивает срочного внимания. Прерывание передает сообщение процессору о высокоприоритетном событии, требующего приостановки кода, который в настоящий момент выполняется процессором. Процессор в свою очередь временно прекращает текущую деятельность, сохраняя при этом состояние, и обрабатывает прерывание (см. *Определение 2.3.2.*), которое отвечает на событие и обслуживает его, а после этого возвращает управление в прерванный код.

Обработка прерываний происходит по определенному алгоритму. В начале каждого цикла работы процессор проводит проверку на наличие запроса о прерывании. Если имеется прерывание, требующее обработки, то процессор сохраняет в стек адрес инструкции, которую он должен был выполнить, а также данные, полученные после выполнения последней инструкции. Затем процессор переходит непосредственно к выполнению функции обработки прерывания.

В результате после выполнения функции обработки прерывания, сохраненные в стек данные считываются из него, и процессор возвращается к выполнению приостановленной задачи.

По характеру источника, который порождает сигнал, прерывания делятся на:

* асинхронные (внешние или аппаратные) — события, исходящие от внешних аппаратных устройств, они могут произойти в любой момент времени (к примеру, сигнал от таймера (см. *Определение 2.4.6*), нажатие клавиши клавиатуры, движение мыши). Наличие вышеупомянутых событий понимается как запрос на прерывание, т.е. устройства сообщают, что они немедленно требуют внимания от операционной системы;
  + синхронные (внутренние) — события, которые происходят при каких-либо нарушениях одновременно с исполняемым машинным кодом (в частности, обращение к недоступным адресам памяти или некорректный код операции, деление на ноль или переполнение стека);
  + программные (частный случай внутреннего прерывания) прерывания вызываются в тех случаях, когда выполняется особая инструкция в коде программы. Используются такие виды прерываний для обращения к функциям встроенного программного обеспечения, драйверов и операционной системы.

Каждому прерыванию присвоен свой номер. Система же определяет, какое именно прерывание произошло, а также его номер, после чего запускает соответствующую ему программу обработки.

В качестве примера рассмотрим прерывание под номером 9, которое поступает от клавиатуры при нажатии или отпускании клавиши. Происходит чтение данных с устройства ввода и обозначается в ОС как IRQ1, где IRQ — обозначение прерывания, а 1 — приоритет прерывания.

*Определение 2.3.2.*

Обработчик прерываний — это процедура, которая обрабатывает прерывания, является частью ОС, предназначена для ответа на конкретное условие, вызвавшее прерывание.

Предположим, что программа А находится в процессе решения, но в какой-то момент времени поступает сигнал прерывания от произвольного источника. В таком случае, управление автоматически передается обработчику прерываний. Завершив обработку, управление вернётся обратно в точку программы А, в которой было прервано ее выполнение.

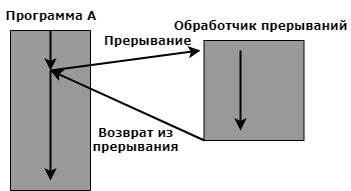
****

Рис. 2.3.1 Обработчик прерываний

Вектором прерывания называют адреса обработчиков прерываний.

Векторы прерываний собираются в таблицу векторов прерываний, которые включают в себя адреса обработчиков прерываний. Расположение таблицы зависит от типа и режима работы процессора.

Таблица векторов прерываний занимает первый килобайт оперативной памяти, то есть адреса от 0000:0000 до 0000:03FF. В составе таблицы всего 256 элементов (адресов обработчиков прерываний). В первом слове каждого элемента таблицы хранится смещение, а во втором – адрес сегмента обработчика прерывания. Полные адреса памяти программы являются векторами. Данная программа должна быть активирована в случае возникновения прерывания.

Прерыванию под номером 0 ставится в соответствие адрес 0000:0000, прерыванию под номером 1 соответствует 0000:0004 и т.д. Такой адрес состоит из пары 2-байтовых слов, в связи с этим каждый вектор занимает четыре байта.

Просмотр таблицы векторов прерываний в компьютере возможен при использовании программы DEBUG. Эта команда отразит первые 128 байтов или 32 вектора. Команда D предназначена для вывода содержимого начала памяти: D 0:0. Пример использования программы DEBUG:

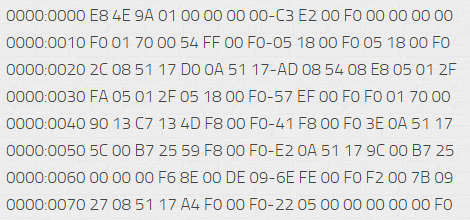


Рис. 2.3.2. Таблица векторов прерываний в компьютере

Механизм обработки прерываний:

Во время обработки любого прерывания выполняются следующие действия:

Во время обработки любого прерывания выполняются следующие действия:

* Прием сигнала и распознавание прерывания
* Фиксирование состояния процесса, который был прерван (определяется адресом следующей команды и тем, что содержат регистры процессора)
* Перенос руководства прерывающей программе (начальный адрес подпрограммы обработки прерываний записывается в счетчик команд, а информация из слова состояния процессора заносится в соответствующие регистры).
* Обрабатывание прерывания.
* Возрождение процесса, который был прерван, и возвращение в прерванную программу.

Основные функции механизма прерывания:

* Идентификация и систематизация прерываний.
* Перенос руководства обработчику прерываний.
* Осуществление возврата к программе, которая была прервана.

Типы прерываний

Возникающие при работе вычислительной системы прерывания дифференцируют на 4 группы:



Рис. 2.3.3. Типы прерываний

Определение 2.3.3

Аппаратные прерывания — прерывания, которые вызваны процессами в аппаратуре вычислительной системы. Аппаратные прерывания не согласуются с работой программного обеспечения. Во время вызова прерывания процессор останавливает свою работу, осуществляет прерывание, и только затем возвращается на исходное место.

Определение 2.3.4

Внешние прерывания появляются по сигналу внешних устройств и классифицируются на два вида: немаскируемые и маскируемые.

Процессор может иметь защиту от прерываний: отключение всей системы прерываний или маскирование отдельных сигналов.

Маскируемые прерывания могут быть замаскированы через программные средства компьютера;

Немаскируемые прерывания таким образом замаскировать не получится.

Внутренние прерывания появляются в результате событий, связанных с работой процессора:

при нарушении адресации (указан неразрешенный или несуществующий адрес);

наличие в поле кода, двоичная комбинация которого не задействована;

при делении на 0;

при переполнении или потере порядка;

при выявлении ошибок четности, ошибок в работе разных устройств аппаратуры средствами контроля;

Программные прерывания

Программы имеют возможность сами вызывать прерывания с определенным номером. Для этого применяется команда INT, при которой процессор выполняет почти те же действия, что и при обыкновенных прерываниях, но только это осуществляется в конкретной точке программы — там, где программист расположил данную команду.

Программные прерывания в прямом значении прерываниями не являются, потому что предполагают собой только своеобразный метод вызова процедур не по адресу, а по номеру в таблице.

Определение 2.3.5

Флаг — это один бит или некоторое количество битов памяти, принимающие бинарные значения (или комбинации значений) и характеризующие состояние какого-либо объекта. Флаг имеет возможность принимать 0 (сброшен) или 1 (установлен).

Регистр флагов называется FLAGS. Это тридцатидвухразрядный регистр. Но во время работы в защищенном режиме применяются только старшие шестнадцать разрядов. К младшим шестнадцати разрядам этого регистра можно обращаться как к отдельному регистру с именем FLAGS.

Регистр применяется побитно, то есть каждый бит имеет конкретное функциональное предназначение, которое называется флагом. Они управляются разными командами для индикации состояния операции. При любом случае флаг сохраняет свое значение, пока иная команда не изменит его.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| CF | 1 | PF | 0 | AF | 0 | ZF | SF | TF | IF | DF | OF | IOPL | | NT | 0 |

Таблица 2.3.4. Флаговый регистр FLAGS.

CF (Carry Flag) — флаг переноса. Содержит значение «переносов» (0 или 1). В случае если результат предшествующей операции не уместился в приемнике и случился перенос из старшего бита или если требуется заём (при вычитании), устанавливается в 1, в ином случае – в 0.

PF (Parity Flag) — флаг четности. Проверяет младший байт результатов предшествующей команды. Нечетное количество бит приводит к установке этого флага в 0, а четное – в 1.

AF (Auxiliary Carry Flag) — дополнительный флаг переноса. Устанавливается в 1, в случае если арифметическая операция приводит к переносу 4-ого справа бита (бит номер 3) в регистровой однобайтовой команде. Этот флаг имеет отношение к арифметическим операциям над символами кода ASCII и к десятичным упакованным полям.

ZF (Zero Flag) — флаг нуля. Устанавливается как результат арифметических команд и команд сравнения. В случае если результат предшествующей команды равен 0, устанавливается 1.

SF (Sign Flag) — знаковый флаг. Устанавливается в соответствии со знаком результата (старшего бита) после арифметических операций: положительный результат устанавливает 0, а отрицательный – 1.

TF (Trap Flag) — флаг трассировки (пошагового выполнения). В случае если данный флаг установлен в 1, то процессор переходит в режим пошагового выполнения команд или же управление временно передается отладчику (см. Определение 3.3.4).

IF (Interrupt Flag) — флаг прерывания. При установке состояния 0 прерывания запрещены, при 1 – разрешены.

DF (Direction Flag) — флаг направления. Применяется в строковых операциях для определения направления передачи данных. При состоянии 0 строки станут обрабатываться в сторону увеличения адресов, при 1 – в сторону уменьшения адресов.

OF (Overflow Flag) — флаг переполнения. Фиксирует арифметическое переполнение, то есть перенос вниз старшего (знакового) бита при знаковых арифметических операциях. К примеру, в случае если при сложении 2-ух положительных чисел получается число со старшим битом, равным единице, то оно отрицательное. И наоборот.

IOPL (Input/Output Privilege Level) — флаг уровня приоритета ввода/вывода. Применяется в защищенном режиме работы процессора для контроля доступа к командам ввода/вывода в зависимости от приоритетной задачи.

NT (Nested Task) — флаг вложенности задач. Применяется в защищенном режиме работы процессора, фиксируя, что одна задача вложена в другую.

2.4. Представление о микросхемах обвязки процессоров

*Определение 2.4.1*

Чипсет (англ. *chipset*) — набор микросхем, которые обычно применяются при создании системных плат для «обвязки» процессора, спроектированных для совместной работы с целью выполнения набора каких-либо функций.

Чипсет в компьютере, размещаемый на материнской плате, выполняет роль связующего компонента, обеспечивающего совместное функционирование подсистем памяти, центрального процессора, устройств ввода/вывода и других. Чипсеты встречаются и в других устройствах, например, в сотовых телефонах.

*Определение 2.4.2*

Обвязка — вспомогательные части системы, обеспечивающие работу основного элемента, и как правило расположенные вокруг него.

Микросхемы обвязки процессоров — вспомогательные микросхемы, расположенные вокруг процессора.

В современных процессорах внутри корпуса уже расположены несколько шин, видеокарта и несколько микроконтроллеров шин. Всё чаще для современных процессоров требуется все меньше и меньше обвязки. Но при этом, процессору по-прежнему нужны шины, внешние устройства, которые в большинстве случаев удается уместить в корпус процессора. Конечно же, с точки зрения архитектуры, все это отдельные части, которые просто находятся под одной крышкой.

*Определение 2.4.3.*

Аппаратный порт — разъем специального назначения в компьютере, предназначенный для предоставления подключения оборудования какого-либо установленного типа.

Как правило, портами называют разъемы, необходимые для работы периферийного оборудования, значительно обособленного от архитектуры компьютера.

Северный и южный мосты компьютера (вернее материнской платы) — два основополагающих функциональных микроконтроллера, отвечающих за деятельность всех элементов системной платы и называются чипсетом.

*Определение 2.4.4.*

Северный мост (англ. *Northbridge*) — системный микроконтроллер, считающийся одним из компонентов чипсета материнской платы, отвечающий за взаимодействие с оперативной памятью (RAM), видеоадаптером и процессором (CPU).

Ответственен за частоту системной шины, тип оперативной памяти и ее предельно допустимый объем. Одна из центральных функций северного моста *—* обеспечение работы системной платы и процессора, и вместе с этим установление скорости работы. Во многие современные материнские платы по умолчанию встраивается видеоадаптер. Также функциональная черта северного моста заключается в управлении шиной видеоадаптера и ее быстродействием. К тому же северный мост отвечает за связь всех вышеперечисленных устройств с южным мостом.

*Определение 2.4.5*

Южный мост (англ. *Southbridge*) — функциональный микроконтроллер, чаще называемый контроллером ввода-вывода или ICH (*In/Out Controller Hub*). Контролирует »медленные» операции, к числу которых принадлежит обработка взаимодействия между интерфейсами IDE, SATA, USB, LAN, Embeded Audio и северным мостом системы, который напрямую соединен с процессором и другими значимыми составляющими (оперативная память, видеоподсистема и т.п.). Также южный мост ответственен за обработку данных на шинах PCI, PCIe.

*Определение 2.4.6*

Таймер — многофункциональное устройство, задающее временный интервал. Представляет собой ступень между аналоговыми и цифровыми устройствами. С помощью него осуществляется конвертирование аналоговых сигналов в последовательность импульсов по заданному закону.

*Определение 2.4.7*

Контроллер прерываний (англ. *Programmable Interrupt Controller, PIC*) — микросхема или встроенный узел процессора, определяющий возможность ступенчатой обработки запросов на прерывание от других устройств.

*Определение 2.4.8*

Цифро-аналоговый преобразователь (ЦАП) — устройство, выполняющее преобразование цифрового (обычно двоичного) кода в аналоговый сигнал (ток, напряжение или заряд), служит для взаимодействия дискретного цифрового мира и аналоговых сигналов.

Аналого-цифровой преобразователь (АЦП) осуществляет обратную операцию.

## 

2.5. Представление о конвейерном выполнении команд в процессорах

Выполнение каждой команды складывается из ряда последовательных этапов, суть которых сохраняется от команды к команде. Считается, что конвейерный принцип обработки информации, наиболее распространенный в настоящее время, позволяет достичь оптимальной производительности процессора. Особенность этого принципа состоит в том, что в каждый момент времени для работы над разными стадиями выполнения нескольких команд выделяются аппаратные ресурсы. С каждым тактовым импульсом команды в конвейере продвигаются на последующую стадию обработки, при этом выполненная команда покидает конвейер, а новая поступает в него.

На рисунке 2.5.1 изображен конвейер из пяти блоков (ступеней). Первая ступень (блок С1) вызывает очередную команду из памяти, помещая ее в буфер, где она и хранится, пока не потребуется. Вторая ступень (блок С2) декодирует эту команду, определяя типы самой команды и операндов. Третья ступень (блок С3) вызывает операнды из регистров или из памяти. Четвертая ступень (блок С4) выполняет команду, при этом операнды, как правило, проходят через тракт данных. Пятая ступень (блок С5) заносит результат обратно в нужный регистр.

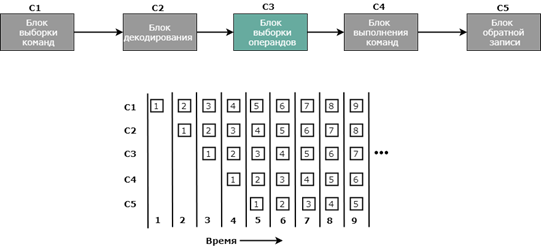
****

Рис. 2.5.1 – Пятиступенчатый конвейер. Состояние каждой ступени в зависимости от кол-ва пройденных циклов

1 цикл: блок С1 вызывает из памяти команду 1 и обрабатывает ее.

2 цикл: блок С2 декодирует команду 1, а блок С1 вызывает из памяти следующую команду 2.

3 цикл: блок С3 вызывает операнды для команды 1, блок С2 декодирует команду 2, а блок С1 вызывает команду 3.

4 цикл: блок С4 выполняет команду 1, С3 вызывает операнды для команды 2, С2 декодирует команду 3, а С1 вызывает команду 4.

5 цикл: блок С5 заносит результат выполнения команды 1 обратно в регистр, а другие ступени обрабатывают последующие команды.

Наборы команд и взаимодействие с операндами с позиций длины и операндов.

1. Процессор с сокращенным набором команд

*Определение 2.5.1*

RISC (англ. *restricted (reduced) instruction set computer* — «компьютер с сокращенным набором команд») — архитектура процессора, при которой скорость работы повышается путем упрощения команд, что позволяет сделать их декодирование проще, а время выполнения — меньше.

Недостатки RISC вытекают из преимуществ этой архитектуры. Один из ключевых недостатков — меньшее число команд по сравнению с CISC (см. *Определение 2.5.2*), из-за чего на выполнение ряда функций приходится тратить не одну, а несколько команд. Это приводит к удлинению кода программы, увеличению загрузки памяти и трафика команд между памятью и ЦП. Установлено, что RISC-программа в среднем на 30% длиннее аналогичной CISC-программы.

Несмотря на то, что большое количество регистров дает существенные преимущества, они в то же время усложняют схему декодирования номера каждого регистра, это ведет к увеличению времени доступа к регистрам.

Устройство управления с аппаратной логикой, реализованное в большинстве RISC-системах, менее гибко и надежно, затрудняет поиск и исправление ошибок, проигрывает при выполнении сложных команд.

1. Процессор с расширенным набором команд

*Определение 2.5.2*

CISC (англ. *Complex Instruction Set Computing* — «компьютер с полным набором команд») — это архитектура процессора, позволяющая одиночным командам выполнять несколько операций низкого уровня (например, загрузка из памяти, арифметические операции, хранилище памяти), а также производить многоступенчатые операции или режимы адресации в рамках одной инструкции.

Основная идея CISC-архитектуры состоит в том, что в данной архитектуре стремятся иметь отдельную машинную команду для каждого возможного (типового) действия по обработке данных.  
CISC-архитектура возникла одной из первых. Совершенствование процессоров шло по пути создания ВМ, способных выполнять как можно больше различных команд. Это упрощало работу программистов, которым приходилось писать программы на языке ассемблера. Использование более сложных команд помогало сократить время разработки программы.  
  
Для CISC-процессоров характерны следующие черты организации:

* большое количество различных машинных команд (порядка сотен), выполняющихся за несколько тактов процессора;
* устройство управления с программируемой логикой;
* малое количество регистров общего назначения;
* различные форматы команд с разной длиной;
* преобладание двухадресной адресации;
* продвинутый механизм адресации операндов с использованием методов косвенной адресации.

В результате использования CISC-подхода больше не было возможности исполнять разные команды только с помощью аппаратных средств. Как результат, процессоры обзавелись блоками, которые заменяют некоторые команды, в частности сложные, на последовательности из упрощенных команд. Кроме того, опытным путём было выяснено, что при разработке программ нет нужды во многих сложных командах. В итоге, из-за повышенной сложности команд и их немалого числа устройство управления ВМ вынуждены были строить только на базе программируемой логики, а точнее, используя «медленную» управляющую память. Последнее весьма снижает возможность повышения тактовой частоты процессора. Все эти факторы способствовали повороту в сторону RISC-архитектуры. В это же время CISC-архитектура имеет множество абсолютных преимуществ, из-за этого она всё ещё актуальна (особенно удобна разработчикам программных приложений). Учитывая всё это, неудивительно, что лидирующие фирмы-производители ВМ (Intel, AMD, IBM и др.) в своих новейших разработках могут задействовать CISC-подход.

1. Процессор, работающий со сверхдлинными командами

*Определение 2.5.3*

VLIW (англ. *very long instruction word* — «особо длинное командное слово») — это архитектура процессоров, которая характеризуется возможностью соединения некоторого числа команд в так называемую связку. Необходимо, чтобы команды, входящие в нее, были независимы друг от друга и выполнялись параллельно. В итоге некоторые независимые машинные команды под действием транслятора (см. *Определение 3.3.2*) формируются в единое «особо длинное командное слово».

Сама идея VLIW заключается в том, что задача по эффективному планированию параллельного исполнения команд лежит на «разумном» компиляторе (см. *Определение 3.2.1*). При помощи этого компилятора сначала происходит анализ исходной программы. Задача анализа: найти исполняемые одновременно команды, причем так, чтобы это не сопровождалось конфликтами между ними. По ходу анализа с помощью компилятора можно частично повторять выполнение интересующей нас программы. Затем перед компилятором лежит задача по объединению этих команд в пакеты (связки), они все рассматриваются как очень длинные команды. Простые команды соединяются в одну сверхдлинную по определённым правилам:

* простых команд, соединяемых в единую сверхбольшой длины, столько же, сколько хранящихся в процессоре функциональных (исполнительных) блоков (ФБ);
* в особо длинную команду вошли лишь исполняемые разными ФБ простые команды, иначе говоря, обеспечивается параллельное выполнение всех составных фрагментов сверхдлинной команды.

Сверхдлинная команда обычно составляет в длину от 256 до 1024 битов. В такого рода мегакоманде содержится некоторое число полей (по количеству простых команд, образующих ее), все они описывают операцию для соответствующего функционального блока.

Как простые команды, образующие сверхдлинную, чаще всего используются команды RISC-типа.

Факт того, что выполняемая сверхдлинная команда не сопровождается конфликтами, дает возможность максимально упростить аппаратуру VLIW-процессора и, соответственно, добиться более высокого быстродействия. Практически все цифровые сигнальные процессоры и мультимедийные процессоры, имеющие производительность более 1 млрд операций, построены на архитектуре типа VLIW.

Стоит указать и недостатки VLIW. У кода для VLIW пониженная плотность. Большое количество пустых инструкций к незадействованным устройствам приводит к тому, что программы VLIW-процессоров могут выходить значительно длиннее подобных программ традиционных архитектур.

Ввиду сложных зависимостей внутри кода, фактически нет возможности вручную программировать на уровне машинных кодов для VLIW-архитектуры. Программисты вынуждены обратиться к оптимизации компилятора, а он может быть с ошибками.

Укажем проблемы VLIW-архитектуры:

1) регистровый файл усложнен, в том числе и связи этого файла с вычислительными устройствами;

2) трудно создавать компиляторы, которые ищут в программах независимые команды, объединяют эти команды в продолжительные строки и обеспечивают их одновременное исполнение.

Наборы команд и взаимодействие с операндами с позиций числа независимых потоков данных

*Определение 2.5.4*

SISD (Single Instruction stream / Single Data stream) — вычислительная система с одиночным потоком команд и одиночным потоком данных.

Архитектура SISD — это компьютер архитектуры фон-Неймана с одним процессором, который, работая с одним потоком данных, выполняет последовательные инструкции. SISD–машина не является параллельной, т.к. в данном классе не используется параллелизм данных и инструкций. К этому классу также принято относить суперскалярные (см. Определение 2.5.5), конвейерные и VLIW–процессоры.

SIMD

В SIMD–компьютеры входит один командный процессор (управляющий модуль), который называется микроконтроллером, и процессорные элементы (модули обработки данных). Поступающие команды принимаются, анализируются и выполняются управляющим модулем. Встретив в команде данные, микроконтроллер подает на все процессорные элементы команду, которая выполняется на некотором числе процессорных элементов, каждый из которых имеет свою собственную память для хранения данных. Преимущество данной архитектуры в более эффективной реализации логики вычислений. Почти половина логических инструкций обычного процессора связана с управлением выполнения машинных команд, а остальная часть относится к выполнению арифметических операций и работе с внутренней памятью процессора.

Архитектура ILP–процессоров

Увеличение производительности достигается благодаря модернизации полупроводниковой технологии и повышению плотности микросхем. Использование реализации установленного типа параллелизма тоже помогает добиться роста скорости выполнения программ. Параллелизм на уровне команд (instruction-level parallelism, ILP) оказался возможен за счет построения методик компиляции (см. Определение 3.2.3) и процессоров, которые благодаря параллельному выполнению одиночных RISC-операций ускоряют работу. Системы на основе ILP применяют программы, составленные на традиционных языках высокого уровня для последовательных процессоров, а выявление «скрытого параллелизма» автоматически проводится с использованием соответствующей компиляторной технологии и аппаратного обеспечения.

Суперскалярные процессоры

*Определение 2.5.5*

Суперскалярный процессор — это реализация ILP-процессора для последовательных архитектур, программа для которых не может предоставлять точных сведений о параллелизме. Так как сама программа не обладает точными сведениями о наличии ILP, задача выявления параллелизма отводится специальной аппаратуре, формирующей алгоритм выявления «скрытого параллелизма». Код для суперскалярных процессоров включает цепочку инструкций, дающую корректный результат при выполнении в указанном порядке.

Программа задает последовательный алгоритм, не зная точную природу аппаратного обеспечения, на котором будет выполнять работу или конкретный временной порядок, в котором будут осуществляться команды. Данный подход повышает сложность аппаратного обеспечения. Кроме того, суперскалярный процессор формирует план выполнения, применяя преимущества факторов, определяющихся непосредственно во время выполнения.

Разрешенные для изменения части динамической структуры программы указывают имеющиеся на множестве инструкций взаимосвязи: зависимость по данным и по управлению. Описывая архитектуры суперскалярных процессоров, обычно используют модель окна исполнения. При выполнении инструкции микропроцессор передвигает по статической структуре программы окно исполнения. Команды без зависимостей при этом могут выполняться в окне одновременно.

*Определение 2.5.6*

Тип архитектуры EPIC (EPIC, Explicitly Parallel Instruction Computing — «вычисление с явным параллелизмом машинных команд») — развитие архитектуры VLIW, собравшее в себе многие понятия суперскалярной архитектуры, но адаптированные к архитектуре EPIC.

В сущности, EPIC является «идеологией», которая определяет, как инициировать ILP-процессоры, и, к тому же совокупность характеристик архитектуры, поддерживающей данную идеологию множества разных архитектур наборов команд (ISA, см. Список сокращений).

С архитектурой EPIC связывают ряд различных ISA. Кроме включения или исключения каких-либо параметров архитектуры, проектировщики процессоров должны учитывать требования по набору кодов операций, пределу поддерживаемых типов и числу применяемых регистров. Концепция архитектуры EPIC - архитектура общего назначения, способная обеспечивать высокую степень параллелизма на командном уровне и в числовых, и в скалярных приложениях.

Одна из задач архитектуры EPIC — сохранение построенного в VLIW принципа статического создания плана выполнения, расширенного при этом возможностями суперскалярного процессора, которые помогут новой архитектуре принимать во внимание динамические факторы, обычно ограничивающие параллелизм. Исходя из этого, принципы EPIC реализовывалась на таких идеях, как:

* компилятор — основополагающий элемент в создании плана выполнения, а архитектура предусматривает поддержку компонентов;
* архитектура предоставляет функции, которые позволяют компилятору применять статические ILP;
* архитектура должна позволять передавать план выполнения компилятора на аппаратное обеспечение.

2.6. Классификация и особенности шин в отношении способа передачи, метода синхронизации, топологии, способа управления, адресации устройств

*Определение 2.6.1*

Цифровой последовательной передачей называется последовательная отправка [битов](https://ru.wikipedia.org/wiki/%D0%91%D0%B8%D1%82) по одному проводу, оптическому пути или частоте. Вследствие того, что это требует меньшей [обработки сигнала](https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D1%81%D0%B8%D0%B3%D0%BD%D0%B0%D0%BB%D0%BE%D0%B2), а также минимальной вероятности ошибки, чем при параллельной передаче, то скорость передачи данных по каждому отдельному пути возможна быстрее. Этот механизм может использоваться на более дальних расстояниях.

*Определение 2.6.2*

Параллельная передача — это передача соответствующих компонентов сигнала одновременно по нескольким путям. Применяя большое количество электрических проводов допускается транслировать сразу несколько бит, что позволяет увеличить скорость передачи, в отличие от последовательной передачи. Данная технология используется внутри компьютера (например, во внутренних шинах данных), в некоторых случаях и во внешних устройствах (например, в принтерах). Возможно искажение изображения, так как провода при параллельной передаче содержат не полностью одинаковые свойства, поэтому некоторые биты могут быть переданы раньше остальных. Бит четности может позволить снизить число ошибок. Несмотря на это, электрический провод при параллельной передаче данных не так надежен на больших расстояниях, так как передача нарушается с очень большой вероятностью.

Главным отличием последовательных шин от параллельных является способ передачи данных.

Последовательные шины используют один сигнальный канал (для разделения потоков приема-передачи можно использовать два отдельных каналов). А значит, что информационные биты здесь передаются последовательно. Данные для передачи через последовательную шину облекаются в пакеты (единица информации, передаваемая как целое между двумя устройствами), которые, сверх собственных полезных данных, включают еще некоторое количество служебной информации.

В параллельных шинах рассматривается совокупность сигнальных каналов, объединенных по их назначению (управление, адреса, данные), которые имеют определенные электрические характеристики и протоколы передачи информации. Группы этих сигнальных каналов также называются шинами:

* Канал для управления данными (шина управления);
* Канал для адресации данных (шина адреса);
* Канал для обмена данными (шина данных).

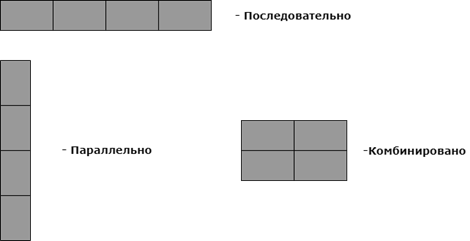
Комбинированная шина:

В каких-то случаях выгодно огромный кусок данных разбить так, чтобы некоторые его части передавались последовательно, а какие-то — параллельно.

Например, нужно передать четыре элемента данных.

Вероятные варианты передачи:

1. Комбинированно: элементы данных объединяют по два и используют два такта для передачи.
2. Передать за четыре такта времени по последовательной шине.
3. Передать за один такт времени по параллельной шине.

**** Рис. 2.6.1. Варианты передачи четырех элементов данных

Классификация и особенности шин в отношении метода синхронизации:

(Синхронизируются 2 объекта: передатчик и приемник. Синхронизация — передача данных)

С целью последовательной передачи данных достаточно одной линии, согласно которой могут поочередно передаваться биты данных. Приемник должен обладать способностью различать, где начинается и где заканчивается сигнал, который отвечает каждому биту данных, то есть передатчику и приемнику необходимо уметь синхронизироваться. В случае, если качество синхронизации невысокое (за время передачи одного бита несогласованность достигает нескольких процентов), используется асинхронный режим передачи данных: производится согласование синхрогенераторов (генерирует импульс, именуемый импульсом синхронизации) в начале передачи любого байта.

Обычно передача байта начинается с особого старт-бита, затем следуют биты данные, а за ними бит четности (Рис. 2.6.2). После битов данных передается стоп-бит. Старт-бит и стоп-бит всегда имеют конкретное значение: старт-бит кодируется логическим нулем, а стоп-бит — логической единицей. Между передачей стоп-бита одного байта и старт-бита следующего байта может проходить произвольное время.

Асинхронный режим очень зависит от погрешностей синхрогенераторов, что задает моменты приема битов. Чем выше скорость передачи, тем выше эта погрешность. В следствие абсолютно всех ограничений темп передачи в асинхронном режиме ограничен сотнями килобит в одну секунду (стандартные скорости: 50, 75, 110, 150, 300, 600, 1200, 2400, 4800, 9600, 19200, 38400, 57600, 115200 бит/с).

*Определение 2.6.3*

Синхронный метод последовательной передачи данных позволяет транслировать сведения блоками. Шаблонная модель передачи данных при синхронном методе: в ходе сопоставления работы приемника и передатчика в начале блока отсылаются биты синхронизации; далее передаются данные, символ, означающий завершение передачи, и код обнаружения ошибки. При синхронной передаче данные могут посылаться как в виде символов, так и потоком битов. Под кодом обнаружения ошибки подразумевается цикличный избыточный код обнаружения ошибок, определяющийся по содержимому поля, благодаря которому устанавливается достоверность данных.

*Определение 2.6.4*

Пакетный режим передачи данных ускоряет процессы чтения данных и их запись в память. Если необходимо чтение из памяти (записи в память) одного машинного слова, процессор считывает вместе с ним еще несколько подряд расположенных слов. Длина любого машинного слова (размер отдельного блока данных, передаваемого между микропроцессором и памятью), равна разрядности внешней шины данных микросхемы памяти.

При пакетной передаче сведений нет потребности указывать целый адрес (номер строки и столбца) каждой ячейки памяти. Взамен подается только адрес начальной ячейки пакета, а следующие ячейки считываются последовательно из текущей строки такое количество раз, сколько слов в пакете. Преимущество такого рода схемы в экономии времени на передачу адреса: для чтения нескольких слов данных необходимо указать лишь один адрес.

В большинстве архитектур SDRAM (см. Список сокращений) подсистема памяти отправляет критическое слово в первую очередь, а уже после идут остальные данные. Таким образом функционирует память SDRAM в пакетном режиме *(burst mode*). Память отправляет на шину четыре блока по 8 байт (или один пакет). Там же присутствует задержка в три тактовых импульса между отправкой запроса и поступлением на шину первого из четырех слов, в случае если задержка чтения в системе составляет 3 такта шины памяти. По этой причине первое критическое слово имеет задержку в 3 такта, и каждый последующий 8-байтный блок — задержку в один такт.

Классификация и особенности шин в отношении топологии

Топология определяет способ взаимодействия компьютеров в сети. Разным видам топологий соответствуют разные методы взаимодействия, которые сильно влияют на сеть. Существуют три базовые топологии, на основе которых строятся все сети: шина *(bus),* звезда *(star)* икольцо *(ring)*. В жизни нередко встречаются достаточно сложные комбинации, которые объединяют свойства нескольких топологий, однако сами по себе базовые топологии легки.

*Определение 2.6.5*

Самую простую и широко распространенную топологиютипа **шина** часто называют линейной шиной (linear bus). В этой топологии все компьютеры параллельно подключаются к одному кабелю, называющемуся магистралью или сегментом. По нему всем участникам сети передаются данные — электрические сигналы, в которых, в том числе зашифрован адрес получателя. Компьютер именно с таким адресом способен их принять. Посылать информацию может тоже только один компьютер единовременно. Из-за этого производительность сети зависит от количества подключений к шине: чем больше компьютеров, готовых передать данные, тем медленнее сеть. Также на быстродействие влияет частота передачи информации, тип кабеля и сетевых приложений, характеристики компьютеров, расстояние между ними в сети.

Шину относят к пассивным топологиям. В этом случае компьютеры в сети только принимают сигналы, не перемещая данные. Так что выход из строя одной машины не повлияет на работу остальных. При активной топологии компьютеры непосредственно отвечают за отправку данных по сети, а сигнал усиливается на каждом шаге при переходе от одного компьютера к другому.

Информация распространяется по всей сети (от одной границы кабеля к другой). Если только одна сторона кабеля к чему-то подключена, возникнет проблема: достигнув неподключенного конца, сигнал отразится и не позволит другим компьютерам осуществлять передачу. Поэтому ко всем свободным окончаниям кабельных линий присоединяют терминатор, способный предотвратить отражение электрических сигналов.

*Определение 2.6.6*

При топологии «звезда» все компьютеры подключаются к центральному компоненту – концентратору (hub) при помощи сегментов кабеля. Сигналы через концентратор распространяются от передающего компьютера к остальным. В сетях с этой топологией подключение кабеля и управление конфигурацией сети централизованы. Однако, для больших сетей значительно увеличивается расход кабеля, из-за того, что все компьютеры подключены к центральной точке.

Если центральный компонент выйдет из строя, нарушится работа всей сети. А если выйдет из строя только один компьютер (или кабель, соединяющий его с концентратором), тогда только этот компьютер не сможет передавать или принимать данные по сети, а на остальные компьютеры в сети это не повлияет.

*Определение 2.6.7*

При топологии «кольцо» компьютеры подключаются к кабелю, который замкнут в кольцо.

У этого кабеля нет свободного конца, к которому нужно подключать терминатор. Сигналы передаются по кольцу в одном направлении и проходят через каждый компьютер. В отличие от топологии «шина», здесь каждый компьютер выступает в роли репитера, который усиливает сигналы и передает их следующему компьютеру. Следовательно, если выйдет из строя один компьютер, перестанет работать вся сеть. Один из принципов передачи данных в рассматриваемой топологии называется передачей маркера: маркер последовательно передается до тех пор, пока его не получит тот компьютер, который «хочет» передать данные. Передающий компьютер изменяет маркер, помещает электронный адрес в данные и посылает их по кольцу. И так далее, пока данные не окажутся у того, чей адрес совпадает с адресом получателя. Затем принимающий компьютер посылает передающему компьютеру сообщение, где подтверждает факт приема данных. Передающий компьютер, получив подтверждение, создает новый маркер и возвращает его в сеть.

*Определение 2.6.8*

Комбинированные топологии– это такие топологии, которые для сочетания компоновки сети применяют принцип шины, звезды и кольца. Звезда-шина *(star-bus)* — это комбинация методов «звезда» и «шина». В случае если ряд сетей с топологией «звезда» соединяются с использованием магистральной линейной шины, то отказ 1-ого ПК никак не влияет на сеть - другие ПК всё так же взаимодействуют между собой. А отказ концентратора в работе приведет к остановке подключенных к нему всех ПК и концентраторов.

*Определение 2.6.9*

Комбинация звезда-кольцо *(star-ring)* отчасти схожа со звездой-шиной. В данных двух топологиях компьютеры подсоединены к концентратору, что формирует шину, либо кольцо. Отличие в том, что концентраторы в звезде-шине объединены магистральной линейной шиной, а в звезде-кольце на основе главного концентратора они формируют звезду.

|  |  |  |
| --- | --- | --- |
| Топология | Преимущества | Недостатки |
| Шина | Экономный расход кабеля. Относительно недорогая и простая в использовании среда передачи. Простота, безопасность. Свободно расширяется. | При достаточно больших объемах трафика снижается пропускная способность среды. Сложно локализовать проблемы. Вывод из строя кабеля останавливает работу многих пользователей. |
| Кольцо | Все ПК обладают равным доступом. Число пользователей незначительно влияет на производительность. | Отказ в работе одного ПК способен вывести из строя всю сеть. Сложно локализовать проблемы. Изменение конфигурации сети потребует приостановки работы всей сети. |
| Звезда | Несложно изменить сеть, добавляя новые компьютеры. Централизованный контроль и управление. Отказ в работе одного компьютера не влияет на работоспособность сети. | Отказ в работе центрального узла выводит из строя всю сеть. |

Таблица. 2.6.2

Классификация и особенности шин в отношении адресации устройств (индивидуальная, групповая, широковещательная).

Распределение шин в отношении адресации устройств зависит от назначения шины. Персональная адресация используется для простой связи двух устройств типа «точка-точка». Однако нередко необходимо связать ряд устройств (объединены шиной и у любого устройства имеется адрес), для этого и необходимы групповая и широковещательная адресации.

Широковещательная адресация, предполагает, что имеется сетевая станция, которая в одно время контактирует со всеми устройствами, которые входят в ту же самую область (домен) широковещания или подсеть. В данном случае все устройства в такой области или подсети могут «слушать» сетевую станцию и получать те же самые данные, которые сетевая станция передает единожды для всех устройств. Недостаток этого режима заключается в том, что, например, с видеосервером возможно в реальном времени реализовывать трансляцию телепередач, к примеру, новостей CNN, в сеть масштаба университетского городка, и при этом они будут доступны каждому пользователю. В данном случае широковещательный трафик обязан пересекать границы подсетей и под него нужно, чтобы была отведена существенная доля пропускной способности. Для широковещания необходимо, чтобы каждые машины и межсетевое оборудование сети, то есть маршрутизаторы и коммутаторы, обрабатывали пакеты широковещательного трафика, в том числе и в случае, если приём широковещательных сообщений требуется исключительно для незначительной части хостов.

Такая проблема решается с помощью групповой IP-адресации, при которой данные с одной станции способны переходить в некоторый ряд станций-приемников одновременно, только в отличие от одноадресного и широковещательного режимов, компьютер-передатчик выбирает определенную группу машин с целью извлечения его данных. Такое допустимо вследствие использования группового IP-адреса, который можно представить как некоторый телевизионный канал. Машины группы просто «настраиваются» на конкретный групповой адрес, посредством которого получают поток представляющей интерес информации. В этом методе потоки данных общего доступа передаются по сети лишь один раз и только таким пользователям, какие хотят их получать. Если, предположим, у нас есть 40 подсетей, а трансляция CNN принимается всего на двух машинах, тогда полоса пропускания 38 оставшихся подсетей не применяется.

Классификация и особенности шин в отношении характера передаваемой информации.

*Определение 2.6.10*

Шина адреса — компьютерная шина (серия, линия, соединяющая два или более устройств), используемая с целью указания адреса ячейки памяти, либо устройства, с которым будет осуществляться обмен. В случае, если процессору нужно осуществить чтение (запись) в ячейку памяти, тогда он указывает ее на адресной шине (значение отправляется по шине данных). Система с 32-битной адресной шиной может адресовать ячеек памяти, аналогично для 64-битной системы – . Пространство адресуемой памяти составляет 4 ГБ, если каждая ячейка памяти содержит 1 байт.

*Определение 2.6.11*

Командная шина представляет собой удобный метод инкапсуляции задач, которые приложение должно интерпретировать в простые и понятные команды. По ней из оперативной памяти поступают различные команды, представленные в виде байтов, которые выполняет процессор. Простые команды вкладываются в один байт, но есть и такие команды, для которых требуется 2, 3 и больше байт.

*Определение 2.6.12*

Компьютерная шина, по которой передаются сигналы, характеризующие вид обмена данными по магистрали, называется шиной управления. По поступающим сигналам устанавливается какую операцию необходимо произвести (считывание или запись), синхронизируют обмен между устройствами и информацией и т.д. В шине управления есть линии, транслирующие следующие сигналы:

* WR — сигнал записи;
* RD — сигнал чтения;
* MREQ — сигнал инициализации устройств памяти (ПЗУ или ОЗУ);
* IORQ — сигнал инициализации портов ввода-вывода.

*Определение 2.6.13*

Мультиплексная шина — это тип структуры шины, в которой количество сигнальных линий, составляющих шину, меньше количества бит данных, адреса или управляющей информации, передаваемых между элементами системы. К примеру, мультиплексированная адресная шина может использовать 8 сигнальных линий для передачи 16 битов адресной информации. Информация передается последовательно, т.е. мультиплицируется во временной области, с дополнительными линиями управления, которые используются для последовательной передачи.

# 2.7. Виртуальные машины. Виртуализация и эмуляция. Гипервизор и супервизор. Контейнеры и докеры.

*Определение 2.7.1*

Виртуализация — абстракция вычислительных ресурсов и передача пользователю системы, которая инкапсулирует собственную реализацию. Иначе говоря, пользователь работает с удобным для себя представлением объекта, и для него не играет роли, как объект устроен в действительности.

Одним из видов виртуализации является виртуализация платформ.

Продуктом этого вида виртуализации являются виртуальные машины (*virtual machine*) — контейнер ПО, который содержит операционную систему и приложения.

Виртуализация платформ — формирование программных систем на основе существующих аппаратно-программных комплексов.

Система, наделяющая аппаратным ресурсом и программным обеспечением, называется хостовой (*host*), а симулируемые ей системы – гостевыми (*guest*). Гостевая система не зависит от архитектуры [хостовой](https://ru.wikipedia.org/wiki/%D0%A5%D0%BE%D1%81%D1%82) платформы и реализации платформы виртуализации.

*Определение 2.7.2*

Гостевая система — это виртуальная машина, которая использует общие средства, предоставленные хостом. Она охватывает приложение и все, что необходимо для его запуска: библиотеки и системные исполняемые файлы. Такая система несет в себе целый аппаратный стек, в том числе виртуальные сетевые адаптеры, файловое хранилище и ЦП, а также свою полноценную гостевую ОС.

Процессор с поддержанием виртуализации работает в режимах root operation и non-root operation.

В режиме root operation работает отдельное ПО, являющееся прослойкой между гостевыми операционными системами и оборудованием — монитор виртуальных машин (*Virtual Machine Monitor*) или гипервизор (*hypervisor*).

Для перевода процессора в режим виртуализации, платформе виртуализации нужно вызвать инструкцию VMXON и передать управление гипервизору, который запускает виртуальную гостевую систему инструкцией VMLAUNCH и VMRESUME (точки входа в виртуальную машину), см. рис 2.7.1.

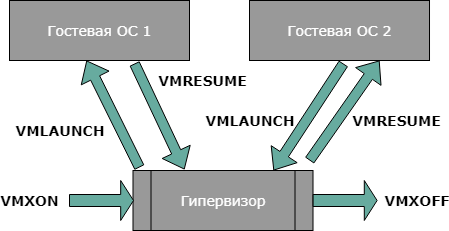
****

Рис.2.7.1. Перевод в режим виртуализации

В случае использования вызова инструкции VMXOFF, Virtual Machine Monitor сможет выйти из режима виртуализации процессора.

Любая из гостевых ОС запускается и работает вне зависимости от других .

Аппаратная виртуализация

В аппаратной виртуализации базовым слоем является гипервизор (см. *Определение 2.7.4*). Этот слой загружается на сервере и обеспечивает взаимодействие между аппаратным обеспечением сервера и виртуальными машинами. Для предоставления ресурсов виртуальным машинам, обеспечивается их виртуализация на сервере. Виртуальные машины запускают свою собственную копию операционной системы и приложений на виртуализированном оборудовании.

Преимущества аппаратной виртуализации

* Создание множества виртуальных машин с различными операционными системами.
* Виртуальные машины содержат собственное виртуальное оборудование и ПО, запускаемое в виртуальных машинах без модификации.
* Виртуальные машины изолированы друг от друга и от ОС сервера, где происходит запуск виртуальных машин.

Программная виртуализация на базе ядра ОС

Программная виртуализация разделяет ресурсы сервера на уровне ядра операционной системы. С помощью средств ОС создаются контейнеры. Виртуальный сервер мало чем отличается от выделенного.

В связи с тем, что все виртуальные машины собраны на одном ядре, пользователь может применять операционную систему, которая совместима с ним. На базе Linux можно устанавливать только дистрибутивы Linux (CentOS, Debian, Ubuntu), что касается Windows, можно использовать только версию, как и у ядра.

Достоинства программной виртуализации:

* программная виртуализация фактически не насчитывает ограничений в плане архитектуры процессора;
* допустимость увеличения или уменьшения системных ресурсов: оперативной памяти, дискового пространства, вычислительных ресурсов процессора;
* так как на физическом сервере привлекается только одно ядро, экономятся ресурсы процессора, дисковое пространство.

Недостатки:

* отсутствие разделения ресурсов;
* использование единой файловой системы, что негативно влияет на общую безопасность;
* запросы ПО на базе гостевой ОС, обрабатываются сначала общим ядром сервера, затем адресуются к аппаратным ресурсам. В результате процесс исполнения команд замедляется;
* неосуществимость размещения ресурсоемких проектов.

Виды виртуализации платформ:

* Полная эмуляция (симуляция)

Проекты, выполняемые по технологии полной эмуляции, работают как интерпретаторы (см. *Определение 3.2.2*): последовательно выбирают код гостевой ОС и эмулируют разные аппаратные архитектуры. Минус подхода — высокая потеря производительности гостевой ОС.

* Частичная эмуляция (нативная виртуализация)

Для изолированного запуска виртуальная машина виртуализирует аппаратное обеспечение, что разрешает запускать гостевые ОС, разработанные только для той же архитектуры, что и у хоста. Из этого следует что, гостевые системы могут быть запущены параллельно. Данный способ виртуализации увеличивает быстродействие гостевых систем по сравнению с полной эмуляцией. Для ускорения работы, в платформах виртуализации, используется «прослойка» между гостевой ОС и оборудованием — гипервизор, который помогает гостевой системе напрямую обращаться к возможностям аппаратного обеспечения. Минус этого способа — то, что виртуальные машины зависят от архитектуры аппаратной платформы.

* Частичная виртуализация (виртуализация адресного пространства)

Виртуальная машина симулирует некоторое количество экземпляров аппаратного окружения, в частности, пространства адресов. В связи с этим можно одновременно использовать ресурсы и изолировать процессы, не разделяя экземпляры гостевых ОС. То есть, пользователем не создаются виртуальные машины, а происходит изоляция некоторых процессов на уровне ОС.

Гипервизор и супервизор.

*Определение 2.7.3*

Супервизор — программа, которая управляет процессами, значительно упрощая работу с памятью и оборудованием ОС.

*Определение 2.7.4*

Гипервизор — программа, с помощью которой можно одновременно управлять несколькими операционными системами на одном и том же хост-компьютере.

Обязанности гипервизора:

* изолирование ОС друг от друга;
* разделение ресурсов между ОС;
* максимально эффективное управление ресурсами;
* обеспечение защиты и безопасности ОС.

Контейнер, докер

Контейнеры и виртуальные машины близки относительно своих задач: изолирование приложения и его зависимостей в самостоятельный блок, который можно запускать с разных хост-компьютеров, т.к. в процессе перемещения между ними содержимое не меняется.

Главное отличие между контейнерами и виртуальными машинами — в их архитектурном подходе.

На рис.2.7.2 представлена архитектура виртуальной машины, а на рис.2.7.3 архитектура контейнера.

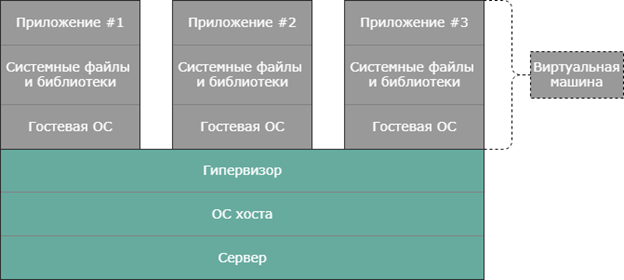
****

Рис. 2.7.2. Архитектура виртуальной машины

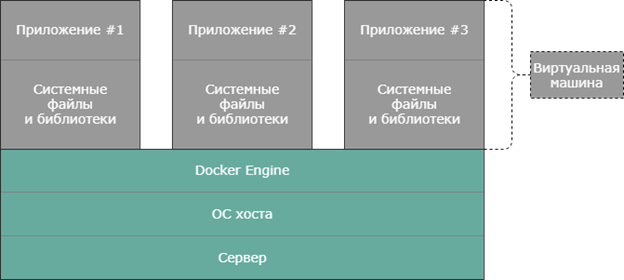
****

Рис. 2.7.3. Архитектура контейнера

Контейнеры, в отличие от виртуальной машины, которая обеспечивает аппаратную виртуализацию, обеспечивают виртуализацию ОС.

Если запуск выполняется в виде контейнера, то приложение попадает в ту же ОС, где происходит взаимодействие с остальными приложениями. Это происходит потому, что контейнеры разделяют ядро хоста с другими контейнерами. А виртуальная машина может выбрать любую доступную ей ОС. Таким образом, при запуске запросы со стороны контактирующих приложений совершают длинный путь из одной операционной среды в другую и обратно.

Еще одно различие состоит в порядке создания каждого из типов.   
Контейнеры не упаковывают ядро или виртуальную аппаратуру, а включают лишь пользовательское пространство, в которое потом можно добавить то, что еще потребуется. При этом каждый контейнер получает персональное изолированное пользовательское пространство для обеспечения возможности запуска нескольких контейнеров на одном хосте. Контейнеры разделяют между собой ресурсы хоста-контейнера, именно поэтому контейнеры настолько легковесны.  
Для виртуальных машин, наоборот, сначала берется полная операционная система, а затем, исключается то, что не понадобится для работы данного приложения.

*Определение 2.7.5*

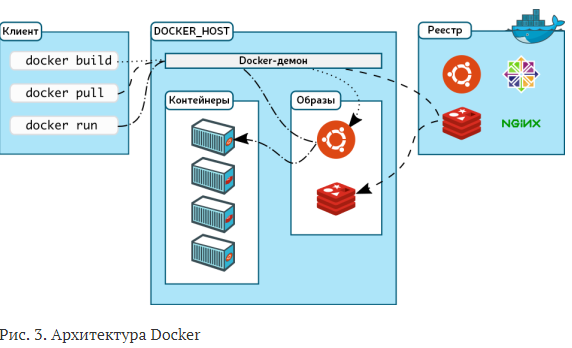
Docker — инструмент для развертывания (осуществление действий, подготавливающих программную систему к использованию) и управления приложениями в среде виртуализации на уровне операционной системы.

Главными преимуществами Docker являются:

* Легкость использования. Созданная на контейнерах docker платформа дает возможность портировать полезную нагрузку. К тому же docker-контейнеры могут работать не только на реальной локальной машине в центре обработки данных, но и на виртуальной машине в облаке.
* Скорость работы. Docker-контейнеры легковесны и быстры. Можно создать и запустить Docker-контейнер за секунды, в отличие от виртуальных машин, которые каждый раз запускают полноценную виртуальную ОС.
* Модульность и масштабируемость. Docker предоставляет возможность разделять функциональность приложения в отдельные контейнеры. Вследствие чего обновление компонентов приложения может происходить независимо друг от друга.

На рисунке 2.7.4 представлена архитектура Docker.

* Docker Daemon (демон Докера) — фоновый сервис, запускается на хост-машине, с которой взаимодействует клиент, отвечает за создание, запуск и уничтожение docker-контейнеров.
* Docker Client (клиент Докера) — программа, которая получает инструкции от пользователя и взаимодействует с docker-демоном.

**** Рис. 2.7.4 Архитектура Docker

# 2.8. Ядро ОС. Планировщик и механизмы переключения задач

Ядро (англ. Kernel) операционной системы координирует всю её работу, упорядочивая путь, начиная от системного оборудования до программного и аппаратного обеспечения, которое посредством графического интерфейса управляется пользователем.

Функциональное назначение ядра:

* Обработка прерываний;
* Формирование и уничтожение процессов;
* Перевод процесса из состояния в состояние;
* Остановка и активация процессов;
* Организация взаимодействий между процессами;
* Синхронизация процессов;
* Манипуляция блоками управления процессами;
* Управление устройствами ввода-вывода;
* Поддержка выделения и перераспределения памяти;
* Поддержка функционирования файловой системы;
* Поддержка механизма вызова-возврата при вызове процедур ОС;
* Поддержка функций по ведению учета работы машины.

Несколько вариантов реализации:

* Монолитное ядро.

Широкий набор служебных процедур работает в привилегированном режиме, задействуя общие структуры данных, причем каждая из процедур может вызвать каждую. Так, все компоненты ОС не являются независимыми модулями, а формируют одну большую программу.

* Модульное ядро.

Предоставляет некоторый механизм как динамической, так и статической подгрузки модулей ядра, поддерживающий определенное аппаратное обеспечение. Таким образом, модульное ядро не требует многократной полной перекомпиляции при каком-либо изменении состава аппаратного обеспечения компьютера.

* Микроядро.

Обеспечивает работу с оборудованием посредством функций управления процессами и элементарного набора примитивов, реализующих базовые сервисы ОС, в которых размещены все или почти все драйверы и модули.

* ОС с экзоядром (экзо- — нечто внешнее, находящееся снаружи).

Предоставляет функции для межпроцессорного взаимодействия с ограничением: экзоядро должно иметь возможность гарантировать безопасное выделение и освобождение ресурсов оборудования.

* Наноядро.

Выполняет только задачу обработки аппаратных прерываний, производимых устройствами компьютера и, используя тот же механизм, отправляет отчет о результатах.

* Гибридные ядра (преобразованные микроядра)

Предоставляют возможность ускорить работу, запуская второстепенные части в пространстве ядра.

*Определение 2.8.1.*

Планировщик задач — это часть ОС, которая отвечает за построение очереди для выполнения задач, потоков и процессов.

*Определение 2.8.2.*

Невытесняющая планировка — это система планирования процессов, при которой активный процесс продолжает работать, пока по собственной инициативе не передает управление планировщику ОС, который выбирает из очереди другой процесс, готовый к реализации.

* FIFO — это готовый к выполнению процесс, который встает в очередь готовых процессов. При прекращении выполнения текущего процесса для следующего выполнения отбирается процесс, находившийся в очереди дольше остальных.
* EDF – это алгоритм с динамическим планированием задач, в котором чем приоритет выше, тем срок выполнения задачи меньше.

*Определение 2.8.3.*

Вытесняющая планировка — это система, которая отличается от невытесняющей планировки тем, что решение о переведении процессора с выполнения одного процесса на выполнение другого определяется планировщиком операционной системы, а не самой активной задачей.

* RMS – это алгоритм с фиксированным приоритетом, в котором приоритет задачи присваивается исходя из принципа: чем реже, тем выше ее приоритет.
* LSTF – это алгоритм планирования, в котором приоритет задачи определяется исходя из соответствия с временем связывания задачи: чем меньше, тем больше приоритет.

Механизмы переключения задач.

В многозадачной системе существует принцип сохранения состояния задачи на момент ее останова (см. Определение 3.3.9). Контекстом называют информацию о задаче при последующем восстановлении прерванного процесса. В такой системе отводится определенная область оперативной памяти, которая доступна только ОС, где хранятся контексты задач.

Для минимизации времени при переключении необходимо восстановить и сохранить минимальное количество информации для каждой задачи. Важно помнить, что задачи сами по себе не являются рентабельными, так как не имеют возможности вызвать сами себя, ведь контексты сохраняются не в стеке, а в фиксированной области памяти в специальной структуре данных, то есть в сегменте состояния задачи (Task State Segment, TSS). Сегмент TSS определяется дескриптором, который хранится только в GDT (Global Descriptor Table).

Термин «старая задача» - это та задача, при выполнении которой она прекращается. Термин «новая задача» - это такая задача, при выполнении которой начинает работать новый процессор.

Процедура, которая обращается к дескриптору TSS, может вызвать переключение задачи. Переключение задачи может повлечь следующие события:

* старая задача выполняет команду FAR CALL, обеспечивая перенаправление всех указывающих на него вызовов, или FAR JMP, обеспечивая перенаправление всех указывающих на него переходов, в обоих случаях селектор выбирает шлюз задачи;
* старая задача выполняет команду FAR CALL, обеспечивая перенаправление всех указывающих на него вызовов, или FAR JMP, обеспечивая перенаправление всех указывающих на него переходов, в обоих случаях селектор вызывает дескриптор TSS;
* старая задача вызывает команду IRET для возврата в предыдущую команду, что приводит к переключению задачи, если в регистре EFLAGS бит вложенной задачи NT =1;
* возникло аппаратное или программное прерывание и соответствующий элемент дескрипторной таблицы содержит шлюз задачи.

Существуют две модели переключения задач:

* Прямое переключение задач;
* Косвенное переключение задач с использованием шлюзов.

Принцип работы механизма переключения задач показан на рис. 2.8.1.

Процесс выполнения переключения задачи можно представить следующим образом:

При наличии TR (task register), использующего регистр дескриптора TSS, который определяет TSS старой задачи, если селектор при использовании команд FAR JMP, FAR CALL, IRET(NT = 1), INT указывает прямо, через дескриптор TSS, или косвенно, используя шлюз задачи, в GDT на системный объект переключения задачи, то происходит переключение задач:

* процессор сохранит контекст старой задачи в сегмент TSS старой задачи;
* процессор загрузит в TR селектор сегмента новой задачи;
* процессор загрузит в сегмент TSS новой задачи селектор TSS старой задачи;
* при получении доступа к сегменту TSS новой задачи, процессор загрузит текст новой задачи в регистры;
* процессор установит флаги NT, используя регистр EFLAGS, и TS, и установит бит занятости задачи в дескрипторе TSS новой задачи.

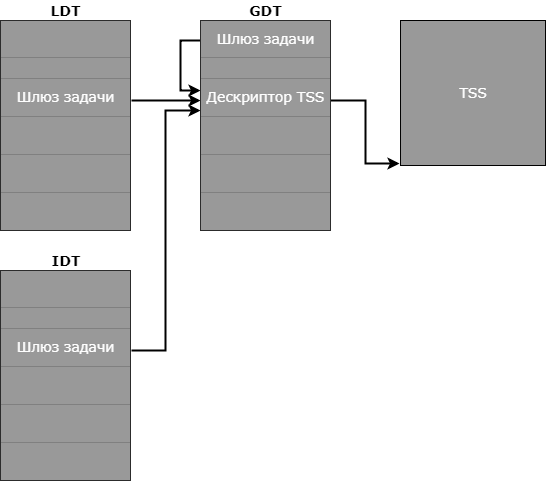
****

Рис. 2.8.1. Принцип работы механизма переключения задач

# Глава 3. Процессор (программирование)

## 3.1. Понятие о параллельных и распределенных вычислениях. Программно-аппаратные переходы и аспекты их применения.

*Определение 3.1.1*

Распределённые вычисления — метод построения компьютерных вычислений, отличающийся от параллельных тем, что вычислительные процессы выполняются не на одном компьютере, а распределяются по разным, но при этом показатели работы такой системы активно изменяются и не всегда известны.

*Определение 3.1.2*

Распределенная система — [система](https://ru.wikipedia.org/wiki/%D0%A1%D0%B8%D1%81%D1%82%D0%B5%D0%BC%D0%B0), для которой отношения [местоположений](https://ru.wikipedia.org/wiki/%D0%9C%D0%B5%D1%82%D1%80%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%BE) элементов (или групп элементов) играют существенную роль с точки зрения функционирования системы, а также анализа и синтеза системы.

Для распределенных систем характерно распределение функций, ресурсов между некоторым количеством элементов (узлов) и отсутствие единого управляющего центра, поэтому выход из строя одного из узлов не приводит к полной остановке всей системы. Например, интернет – распределенная система.

*Определение 3.1.3*

Грид-вычисления ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *grid* – решётка, сеть) — это форма [распределённых вычислений](https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D0%BF%D1%80%D0%B5%D0%B4%D0%B5%D0%BB%D1%91%D0%BD%D0%BD%D1%8B%D0%B5_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D1%8F). Здесь «виртуальный [суперкомпьютер](https://ru.wikipedia.org/wiki/%D0%A1%D1%83%D0%BF%D0%B5%D1%80%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80)» представлен в виде [кластеров](https://ru.wikipedia.org/wiki/%D0%9A%D0%BB%D0%B0%D1%81%D1%82%D0%B5%D1%80_(%D0%B3%D1%80%D1%83%D0%BF%D0%BF%D0%B0_%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BE%D0%B2)), соединенных с помощью сети, слабосвязанных [гетерогенных](https://ru.wikipedia.org/wiki/%D0%93%D0%B5%D1%82%D0%B5%D1%80%D0%BE%D0%B3%D0%B5%D0%BD%D0%BD%D0%BE%D1%81%D1%82%D1%8C) компьютеров, работающих вместе для выполнения огромного количества заданий (операций, работ). Эта технология применяется для решения научных, математических задач, требующих больших вычислительных ресурсов.

*Определение 3.1.4*

Облачные вычисления ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) *cloud computing*) — модель обеспечения удобного сетевого доступа по требованию к некоторому общему фонду конфигурируемых вычислительных ресурсов. Они могут быть оперативно предоставлены и освобождены с минимальными эксплуатационными затратами или обращениями к провайдеру.

Аппаратные и программные средства, реализующие распределенную модель вычислений.

*Определение 3.1.5.*

Программным средством, реализующим модель распределенных вычислений, является распределенная операционная система.

Распределенная ОС динамически и автоматически распределяет работы по разным машинам системы для обработки и заставляет набор сетевых машин работать как виртуальный [процессор](https://en.wikipedia.org/wiki/Uniprocessor_system) (представление ядра физического процессора в ОС логического раздела, использующего общие процессоры). У пользователя распределенной ОС нет информации о том, на какой машине выполняется его работа. Появление сетей, которые предназначены для взаимной связи разных компьютеров, привело к разработке средств и ОС, позволяющих осуществлять управление мультимашинной архитектурой, то есть совокупности полносоставных компьютеров (процессоры, память, вводы-выводы и т.д.), связанных в сеть.

*Определение 3.1.6*

Аппаратным средством модели распределенных вычислений могут служить грид-вычисления.

Грид с точки зрения сетевой организации представляет собой согласованную, открытую и стандартизованную среду, которая обеспечивает гибкое, безопасное, скоординированное разделение вычислительных ресурсов и ресурсов хранения информации, являющихся частью этой среды, в рамках одной виртуальной организации.

Многопортовая память, зеркалирование памяти, шины с поддержкой режимов групповой адресации (*multicasting*) и широковещания (*broadcasting)*

*Определение 3.1.7*

Многопортовая память — это статическое ОЗУ с двумя или более независимыми интерфейсами, которые обеспечивают доступ к пространству памяти через разделенные шины адреса, данных и управления. Структура двухпортового статического ОЗУ (Рис. 3.1.1) содержит один массив памяти (COMMON CENTRAL MEMORY) и два независимых порта (PORT\_L и PORT\_R) для обращения к этому массиву.

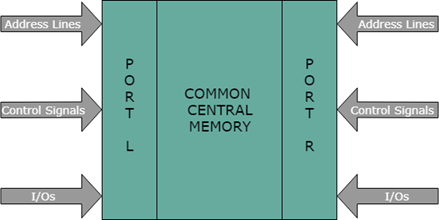
****

Рис. 3.1.1. Структура двухпортового статического ОЗУ

*Определение 3.1.8*

Зеркалирование памяти — это процесс разделения памяти на 2 канала, где первый канал отражается на второй, на втором канале создается избыточная копия памяти. Данный метод позволяет сделать ввод/вывод регистров и памяти в нескольких адресных пространствах, так как один и тот же байт доступен по нескольким адресам.

*Определение 3.1.9*

Broadcast-шина — термин, который описывает способ обмена данными, где некоторый объем информации передается из одной точки во все другие точки. Существует лишь один отправитель, информация передается всем соединенным получателям.

*Определение 3.1.10*

Multicast-шина — термин, который описывает способ обмена данными, где некоторый объем информации передается из одной или нескольких точек в набор других точек. В данном случае может существовать один или несколько отправителей, а информация распространяется по набору получателей (которых может и не быть).

## 3.2.Компиляция и интерпретация исходных текстов программ. Объектный код, исполняемый код, динамическое и статическое связывание модулей программы. Байт-код.

Компиляция и интерпретация исходных текстов программ: сравнительный анализ механизмов и возможностей.

Для того чтобы программа исполнялась компьютером, ее необходимо привести к понятному для «машины» виду.

Существует два способа сделать программный код понятным для компьютера:

* компиляция;
* интерпретация.

*Определение 3.2.1*

Компилятор, не выполняя,  переводит в машинный код сразу всю программу и помещает его в память компьютера.

*Определение 3.2.2*

Интерпретатор переводит небольшой фрагмент исходной программы в машинный код. И только после того, как процессор выполнит этот код, он переходит к обработке следующего фрагмента.

*Определение 3.2.3*

Компиляцией исходных текстов называется процесс преобразования программы, написанной на одной языке, в полностью эквивалентную программу на другом языке.

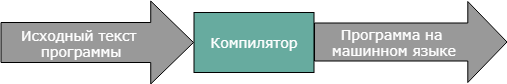
******

Рис.3.2.1. Компиляция исходных текстов

*Определение 3.2.4*

Интерпретация исходного текста программы — это процесс ее выполнения программой-интерпретатором последовательно, оператор за оператором.

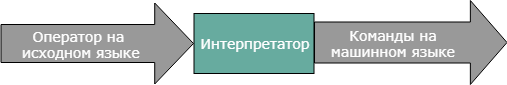
******

Рис.3.2.2. Интерпретация исходного текста программы

Основные различия компиляции и интерпретации:

* Компилятор работает со всей программой целиком, после чего выполняет ее. Интерпретатор делает это построчно.
* Интерпретатор прекратит свою работу после первой найденной ошибки.
* У компилятора анализ и обработка языка высокого уровня занимает больше времени чем у интерпретатора.
* В тоже время, компилятор выполняет обработанный код быстрее, нежели интерпретатор.

Понятие об объектном коде, исполнимом коде, динамическом и статическом связывании модулей программы.

*Определение 3.2.5*

Низкоуровневое представление компьютерной программы в форме, включающей в себя информацию о процессорных командах, информацию о размещении программы, называется объектным кодом.

Исходный код транслируется в исполнимый код с помощью компилятора до начала запуска программы, также он может исполняться сразу с помощью интерпретатора.

Динамическое и статическое связывание – это методы обмена библиотечным кодом в современных компьютерных операционных системах.

*Определение 3.2.6*

Динамическое связывание происходит во время выполнения. Он запускает небольшую дополнительную работу во время запуска для некоторых хороших функций, таких как возможность исправления/замены библиотеки, которая может исправлять или добавлять функции к приложениям, которые связаны с ней.

*Определение 3.2.7*

Статическое связывание — метод, где части библиотеки копируются в исполняемый файл (см. *Определение 3.4.5*). Этот метод в сравнении с динамическим связыванием быстрее для запуска, а в некоторых случаях может быть более удобным для распространения и может сэкономить место.

Байт-код: его назначение, получение, оптимизация и выполнение.

Байт-код, также называемый переносимым кодом или p-кодом, является формой набора команд, разработанного для эффективного выполнения программным интерпретатором. В отличие от читаемого человеком исходного кода, байт-коды представляют собой компактные числовые коды, константы и ссылки (обычно числовые адреса), которые кодируют результат анализа компилятора и выполнения семантического анализа таких вещей, как тип, область действия и глубина вложенности программных объектов.

Сначала исходный текст программы компилируется. Результат этой компиляции – это некий условный машинный код, который не зависит от вычислительной архитектуры и может быть выполнен на разных операционных системах, разными процессорами и его результат все равно будет корректным. Далее этот байт-код запускается на некотором подобии виртуальной машины, то есть эта программа, которая принимает в себя операции на байт-коде и переводит в последовательность команд данной архитектуры процессора и вызовов к данной операционной системе. Таким образом происходит интерпретация байт-кода на конкретной платформе, превращая этот код в исполнимые на данном процессоре действия.

Байт-код используется современными языками программирования (особенно интерпретируемыми) для ускорения и облегчения работы интерпретатора. Метод трансляции в байт-код является промежуточным по эффективности между прямой интерпретацией и компиляцией в машинный код.

Длина любого кода операции равна одному байту. Например, инструкция — как раз однобайтовый код или байт-код. За ней уже могут следовать различные параметры, такие как номер регистра или адрес в памяти.

Программа байт-кода может быть выполнена путем анализа и непосредственного выполнения инструкций, по одной за раз. Преимуществами байт-кода являются большая эффективность и портируемость. Это означает, что определенный байт-код может исполняться на различных платформах и архитектурах, для которых реализован интерпретатор. Обычно эффективность интерпретации байт-кода выше, чем чистая интерпретация исходного кода или интерпретация АСД. Из-за простоты интерпретатора байт-кода его легко переносить на другую архитектуру компьютера.

Прямая оптимизация байт-кода: некоторые инструкции байт-кода намного трудозатратнее, чем другие. Например, загрузка локальной переменной в стек недолгая, но вызовы виртуальных методов, вызовы интерфейса, распределение объектов и перехват исключений идет затратнее. Традиционные C-подобные оптимизации, такие как копирование, имеют незначительный эффект, потому что они не нацелены на дорогие байт-коды. Чтобы выполнять эффективно оптимизации на этом уровне, необходимо рассмотреть более продвинутые оптимизации, такие как встраивание методов и статическое разрешение вызовов виртуальных методов, которые напрямую уменьшают использование этих дорогих байт-кодов.

****

Рис.3.2.3. Принцип работы байт-кода

## 

## 3.3. Специализированные средства разработки программного обеспечения

*Определение 3.3.1*

Ассемблеры — компьютерные программы, которые переводят программы, написанные на языке ассемблера в машинные команды.

*Определение 3.3.2*

Трансляторы — технические средства, которые преобразуют программу в бинарный файл программы, состоящий из машинных команд, или выполняют непосредственно действия программы.

*Определение 3.3.3*

Препроцессоры исходных текстов — это компьютерные программы, обрабатывающие текст исходной программы до компилятора. С помощью препроцессора возможно выполнение следующих операций:

* включение в программу текстов из указанных файлов;
* замена идентификаторов последовательностями символов;
* макроподстановка (замена обозначения параметризованным текстом, формируемым препроцессором с учетом конкретных аргументов);
* исключение из программы отдельных частей текста (условная компиляция).

*Определение 3.3.4*

Отладчик является модулем среды разработки, предназначенным для поиска ошибок в программе.

*Определение 3.3.5*

Текстовые редакторы — компьютерные программы для создания, изменения, просмотра, вывода, поиска фрагментов текста.

*Определение 3.3.6*

Специализированные редакторы исходных текстов — текстовые редакторы для создания или редактирования исходного кода программ.

*Определение 3.3.7*

Библиотеки подпрограмм — сборники подпрограмм или объектов для использования в качестве целых частей при составлении новых программ. Библиотека подпрограмм обеспечивает ускорение и облегчает программирование задач.

*Определение 3.3.8*

Профилирование — процесс анализа программы, нацеленный на ее оптимизацию. В ходе него обычно выясняют следующие характеристики:

* временную сложность программы;
* частоту вызовов функции;
* число верно спрогнозированных ветвлений;
* число кэш-промахов.

и др.

Инструмент, анализирующий работу, называют профилировщиком или профайлером (*profiler*).

Благодаря отладке и профилированию можно обнаружить тормозящую функцию, переписать её на C, вместо C++, а затем тем же способом найти её самую «тормозящую» часть и переписать её на языке ассемблера.

*Определение 3.3.9*

Точка останова (англ. *breakpoint*) — выбранная программистом позиция в коде, по достижении которой работа программы временно останавливается в целях отладки и тестирования. Потом она либо продолжает выполняться, либо завершается.

Точка останова может быть программной или поддерживаться процессором на аппаратном уровне. Всреде *Visual C++*, например, предусмотрена соответствующая инструкция для процессора. Для *Intel* это инструкция *int 3*, и ее выполнение приводит к исключению *EXCEPTION\_BREAKPOINT*.

Если разместить несколько точек останова, программа будет останавливаться через положенные промежутки времени. Так можно поэтапно проанализировать ее состояние: регистры, логи, файлы и т.д. Эту информацию выдает отладчик, считывая необходимые значения. Кроме того, именно отладчик оставляет по нужному адресу инструкцию, описанную выше. Хотя ее можно написать и вручную: для версий со встроенным ассемблером подойдет *\_\_asm int 3*; для других — функция *\_\_debugbreak()*.

При кодировании команды на ассемблере сначала пишется метка (если требуется), потом сама команда, а в конце операнды и комментарий. Команды содержатся в памяти по определенным адресам, а метки на эти адреса указывают. В современном ассемблере они мало чем отличаются от меток в С. В их названии (до 31 знака длиной) допустимы буквы, цифры и спецсимволы. При этом начинаться с цифр они не могут, символы «*$»* и «*?»* не рекомендуются, а регистр значения не имеет. Метки используются, например, при прыжках (jmp) или с командой call.

Пример использования метки для того, чтобы перепрыгнуть процедуру, которую мы хотим реализовать в теле другой процедуры:

|  |  |
| --- | --- |
| code  start  proc  jmp ml  myproc  proc near  ret  myproc  endp  ml:  ...  start  endp  end start | Начало кода программы  В этой (главной) процедуре мы хотим сделать еще одну процедуру  Перепрыгиваем процедуру, которую мы делаем (безусловный переход)  Наша процедура  Она выполняет всего одну команду - ret      Конец ее описания  Это метка, сюда прыгаем  ... Теперь мы можем вызывать нашу процедуру  Конец главного процесса  Указание на то, что программа начинается с метки start |

Примеры самих меток:COUNTER, PAGE69, $GR8.

Метке лучше дать имя, имеющее смысл. Оно не должно совпадать с зарезервированными именами, например, регистров AX и BX и т.д.

Метка, стоящая перед командой процессора, завершается двоеточием, *«:»*. Так ассемблер понимает, что нужно создать переменную с именем метки и значением адреса этой команды:

some\_loop:

lodsw; *считать слово из строки*

cmp ax,7; *если это 7* - *выйти из цикла*

loopne some\_loop

Метка, стоящая перед директивой ассемблера, уже является операндом этой директивы. Двоеточие не нужно:

codesg segment

lodsw; *считать слово из строки*

cmp ax,7; *если это 7 — выйти из цикла*

codesg ends

Некоторые метки уже предопределены, например, *«$»* возвращает текущий адрес. Посредством этой метки легко написать вечный цикл:

jmp $; *выполнять переход на себя*

Далее приведены директивы, работающие с метками.

1) Директива LABEL

С помощью этой директивы можно определить имя метки и ее тип. А типы бывают следующими:

* BYTE (байт);
* WORD (слово);
* DWORD (двойное слово);
* FWORD (6 байт);
* QWORD (учетверенное слово);
* TBYTE (10 байт);
* NEAR (ближняя метка);
* FAR (дальняя метка).

В случае NEAR и FAR значению метки присваивается адрес следующей команды, в остальных случаях — значение следующих данных. Также метка получает тип, указанный явно.

2) Директива EQU

С помощью этой директивы можно определить значение, которое она связывает с именем метки. Это значение представляет собой адрес, целое число или строку:

метка equ выражение

truth equ 1

message1 equ 'Try again$'

var2 equ 4[si]

cmp ax,truth ; cmp ax,1

db message1 ; db 'Try again$'

mov ax,var2 ; mov ax, 4[si]

По своему назначению *EQU* похожа на директиву *#define* языка *C*. Она также отвечает за создание общих для программы параметров.

3) Директива =

С помощью этой директивы тоже определяют значение метки, но оно бывает только целочисленным. При этом такую метку можно переопределять.

## 3.4. Загрузчики и компоновщики. Форматы объектных модулей. Оверлейные загрузчики и оверлеи.

*Определение 3.4.1*

Загрузчик — составляющая операционной системы, которая позволяет загружать программы и библиотеки. Именно он помещает программы в память и готовит их к исполнению, поэтому в процессе запуска этап загрузки считается одним из важнейших. Этот этап включает в себя считывание данных из запускаемого файла; загрузку отсутствующих динамических библиотек и связывание адресов (в случае необходимости); создание нового процесса в памяти.

Количество обязанностей загрузчика может отличаться в зависимости от операционной системы. Например, некоторым из них требуется загрузчик, корректирующий адреса в исполняемом файле (см. *Определение 3.4.5*), потому что программа не всегда загружается в одно и то же место в адресном пространстве. Пусть существует программа, содержащая некоторые абсолютные адреса, которые имеют смысл, только когда она находится по определенному адресу A. В случае если эта программа загружена по другому адресу — B, нужно обновить все адреса, переведя их с B на A. Такое действие и называется связыванием адресов.

Загрузчик также может быть самостоятельной программой, которая позволяет запускать программы, скомпилированные для других ОС. Есть загрузчик и самой операционной системы.

*Определение 3.4.2*

Компоновщик (редактор связей, линкер) — утилита, которая принимает на вход объектные модули (см. *Определение 3.4.4*) и объединяет их в исполняемый файл, то есть совершает компоновку. Это требуется, если части компьютерной программы содержатся не в одном модуле, а ссылаются друг на друга с помощью таблицы символов.

*Определение 3.4.3*

Таблица символов — структура данных, которую заполняет компилятор каждым встреченным символом при компиляции кода. В ней хранится информация о переменной или функции: об их типе данных, области видимости и т.п. Эта таблица позволяет соотнести элементы объектного модуля с их адресами в памяти.

Символы могут быть определенными, неопределенными или локальными. Под определенными (экспортируемыми) чаще всего понимают имена функций и переменных, которые определены в данном модуле, и при этом ими могут пользоваться другие модули. Под неопределенными (импортируемыми) — функции и переменные, ссылающиеся на те модули, где они определены. И, наконец, локальные используются внутри объектного модуля, чтобы упростить процесс связывания адресов.

Компоновка бывает статической и динамической. При статической компоновщик записывает в файл программы все необходимые части. То есть код, требуемый для работы программы, целиком находится в ОЗУ. Например, пусть в системе одновременно работает N приложений, и все они пользуются несколькими одинаковыми функциями. Тогда в памяти будет храниться N копий таких функций. А при динамической компоновке эти функции находятся в памяти в единственном экземпляре; и параллельно работающие программы вызывают их из библиотек, а не из файлов программ. В библиотеках компоновщик может найти дефиниции тех символов, которые по умолчанию не определены, но на них ссылаются пользовательские объектные модули. Найденный модуль, в котором определен искомый символ, компоновщик либо включает в исполняемый файл (в случае статической компоновки), либо откладывает его до запуска программы (в случае динамической).

*Определение 3.4.4*

Объектный модуль (объектный файл) — это фрагмент машинного кода, который может использоваться программой или другими объектными модулями. Он является результатом работы компилятора и служит основой для получения исполняемого файла при помощи компоновщика.

Помимо собственных данных и перечня процедур, объектные модули хранят неопределенные адреса ссылок на подобную информацию в других модулях. Эти адреса вычисляет компоновщик. А именно: при вызове функции компилятор не помещает окончательный адрес программы в объектный файл; но он дает наставление компоновщику искать ссылку в таблицах всех обрабатываемых объектных модулей и уже там указывать окончательное местоположение.

Форматы объектных модулей

Файл объектного модуля содержит отдельные секции данных, каждая из которых включает в себя конкретный тип данных:

* Заголовок (описание файла и нужные компоновщику сведения);
* Сегмент кода (исполнимый код);
* Сегмент данных (инициализация переменных);
* Сегмент констант (инициализация констант, описание глобальных переменных);
* Внешние определения и ссылки на связи;
* Информация о динамических связях;
* Отладочная информация.

*Определение 3.4.5*

Исполняемый модуль (исполняемый файл) — файл, содержащий закодированную последовательность инструкций, которые способен выполнить компьютер. Исполняемые файлы бывают разных форматов, таких как ELF (используется в версиях unix), PE (в 32- и 64-разрядных Windows) и Mach-O (в случае macOS).

*Определение 3.4.6*

Overlay — метод программирования, при котором программы могут занимать больше места в памяти, чем предусмотрено системой. Также оверлеями называют фрагменты кода, на которые делят программу при реализации этого метода.

Процессору отводится ограниченное количество памяти. Оверлеи были придуманы для того, чтобы обеспечить эффективность загрузки ЦП. Идея заключается в том, что процесс можно поделить: пусть выполняется процесс A-E; и в данный момент времени требуется обработать только сегменты A, B, C; а после того, как с ними закончили, — D и E. Оверлеи помогают менять их местами между ЦП и памятью без дополнительной помощи со стороны операционной системы. Место в памяти, куда загружается оверлей, называется регионом*.* При загрузке в регион сохраняются все необходимые связи, например, с библиотеками.

# Глава 4. Параллельные вычисления

*Определение 4.1.*

Параллельные вычисления — подход к организации компьютерных вычислений, в котором вычисления производятся параллельно и на разных конвейерах/ядрах/процессорах, но физически находящихся на одном компьютере.

При таком типе вычислений все узлы тесно соединены между собой, поэтому параметры их работы известны программисту.

Параллельные вычисления используют более одного вычислительного ядра, т.к. все управляющие потоки работают одновременно и занимают весь рабочий цикл ядра на время исполнения — параллельное вычисление невозможно на одноядерном компьютере.  
 Поскольку целью параллельных вычислений является ускорение выполнения алгоритма благодаря совмещению выполнения множества вычислительных операций — главная проблема при проектировании параллельных программ — обеспечить правильную последовательность взаимодействий между различными вычислительными процессами, а также координацию ресурсов, разделяемых между процессами. Это осуществляется с помощью применения обменных средств распараллеливания или же дополнительного выделения блоков алгоритма, предназначенных для выполнения параллельных вычислений, при этом обращаемых к разным областям общей памяти.

Параллельные машины различают по типу памяти — общая (*shared*) для всех процессоров и распределенная (*distributed*), а также по типу управления — SIMD и MIMD (см. *Определение 1.3.3. и Определение 1.3.2.*).

Для SIMD архитектуры характерно наличие одного устройства управления, которое посылает инструкции всем остальным процессорам, причем у каждого процессора свой набор данных для работы. SIMD–системы, как правило, применяются для конкретных задач, которые требуют обычно не столько гибкости и универсальности вычислительной машины, сколько самой вычислительной силы.

В MIMD архитектуре к подобному классу систем относится большинство параллельных многопроцессорных вычислительных систем. У каждого процессора – собственный управляющий узел, а процессор может выполнять разные инструкции. Класс MIMD–машин имеет более широкий функционал, поэтому в пользовательских компьютерах применяются именно они.

Говоря об этих двух типах управления, в первую очередь стоит понимать, что они напрямую связаны с набором команд SSE и пришедшим ему на смену AVX, в результате чего SSE стал частным случаем AVX. Об этом мы уже упоминали ранее в главе 1.3. Сейчас же обратимся к современным версиям этих технологий — AVX2 и AVX-512.

*Определение 4.2*

AVX2 (*Advanced Vector Extensions 2*) и AVX-512 — развитие технологии AVX, 264-битные и 512-битные расширения для SIMD инструкций.

Поясним, спецификация AVX и AVX2 расширяет 128-битные регистры SSE XMM0…XMM15 до 256-битных регистров YMM0…YMM15.

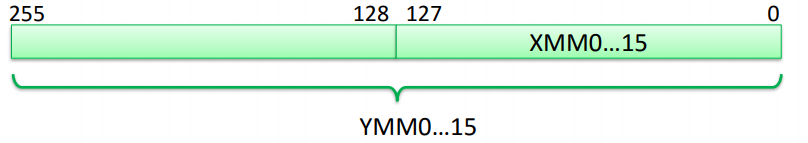
****

Рис.4.1. Регистры AVX

Каждый из 16 YMM регистров вмещает:

* 8 32-битных чисел с плавающей точкой или
* 4 64-битных числа двойной точности с плавающей точкой

А в AVX-512 же регистры SSE расширены до 512 разрядов, как показано на рисунке ниже, и их количество увеличено до 32 регистров ZMM0-ZMM31.

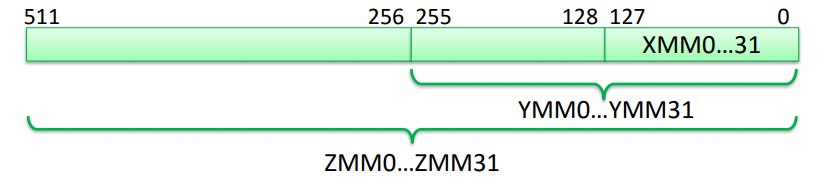
****

Рис.4.2. Регистры AVX2 и AVX-512.

Этот новый набор инструкций повышает производительность различных рабочих нагрузок, включая научное моделирование, финансовую аналитику, искусственный интеллект и глубинное обучение, 3D-моделирование и анализ данных, обработку изображений, аудио и видео, сжатие данных и шифрование. Впервые использован в 2013 г. в процессорах Intel на ядре Haswell.

Практическая польза для клиентского применения — прирост производительности при работе c мультимедийными данными, при этом нагрузка на процессор, естественно, снизится. Это касается и программ, в которых применяются алгоритмы распознавания голоса, лиц, жестов (только при использовании аппаратных ресурсов, совместимых с AVX2).

Далее перечислим новые возможности в AVX2 в сравнении с предыдущими технологиями:

* добавлена инструкция Fused Multiply Add (FMA), объединяющая в себе операции сложения и умножения (а это ключевые действия в цифровой обработке сигналов);
* реализована возможность операций с 256-битными целочисленными векторами;
* реализована поддержка так называемых gather-инструкций, благодаря которым перестает действовать строгое требование непрерывного расположения данных в памяти. Теперь данные могут собираться из разных адресов памяти;
* добавлены инструкции манипуляций/операций над битами.

В целом, инструкции AVX2 ориентированы на повышение быстродействия процессора в целочисленных вычислениях (при условии использования соответствующего программного обеспечения).

На смену AVX2 пришел новый набор инструкций AVX-512, в котором появилась концепция масочных регистров.

Opmask регистры

Большинство инструкций AVX-512 могут указывать на один из 8 opmask регистров (k0-k7). Первый (k0) является жестко запрограммированной константой, позволяющей исключать из операции отдельные элементы вектора. В большинстве инструкций opmask используется для управления, которые записывают значения адресату. Флаг управляет поведением opmask, который может быть либо «ноль», который зануляет все, что не выбрано маской, или «merge», что оставляет все, что было не тронуто. Поведение слияния идентично инструкциям смешивания. Регистры opmask обычно имеют ширину 16 бит, но могут быть до 64 бит с расширением AVX-512BW. Однако многие из битов фактически используются в зависимости от типа вектора маскированных команд. Для 32-битных однополярных или двойных слов 16 бит используются для маскировки 16 элементов в 512-битном регистре. Для double float и четырехзначных слов используются не более 8 бит маски.

AVX-512 позволяет более удобно работать с масочными регистрами и манипулировать с битами.

Сжатие и расширение

Инструкции сжатия и развертывания соответствуют операциям APL с тем же именем. Они используют маску немного иначе, чем другие инструкции AVX-512. Сжатие сохраняет только значения, отмеченные в маске, но сохраняет их сжатыми, пропуская и не резервируя место для немаркированных значений. Expand работает противоположным образом, загружая столько значений, сколько указано в маске, а затем распространяя их на выбранные позиции.

Инструкции с плавающей точкой

Введены три новые операции с плавающей точкой. Поскольку они не только новички в AVX-512, они имеют как упакованные / SIMD, так и скалярные версии.

Инструкции VFPCLASS проверяют, является ли значение с плавающей точкой одним из восьми специальных значений с плавающей точкой, какое из восьми значений будет запускать бит в регистре выходной маски, управляется непосредственным полем. Инструкции VRANGE выполняют минимальные или максимальные операции в зависимости от значения непосредственного поля, которое также может контролировать, выполняется ли операция абсолютно или нет, и отдельно, как обрабатывается знак. Инструкции VREDUCE работают с одним источником и вычитают из этого целую часть значения источника плюс количество битов, указанное в непосредственном поле его дроби.

# Глава 5. Оптимизация кода

## 5.1. Оптимизация циклов (loop optimization)

5.1.1. Развертка (loop unrolling)

Данный вид оптимизации используется при малом теле цикла (с известной длиной). Вместо цикла с n-итерациями, компилятор будет производить код, который просто повторяется n раз.

Метод может повысить производительность, так как устраняет любые инструкции перехода, что является целью многих оптимизаций. Однако, следует учесть, что, если увеличенный цикл не помещается в кэш, то данный способ не целесообразен.

Пример:

Количество итераций/размер массива: 10 000

До: (111 мкс)

for (int i = 0; i < iN; i++){  
 res \*= a[i]; Х  
}

После: (48 мкс)

for (int i = 0; i < iN; i+=3){  
 res1 \*= a[i];   
 res2 \*= a[i+1];  
 res3 \*= a[i+2];  
}  
res = res1 \* res2 \* res3; //вынесли из цикла, так как операция однократная

5.1.2. Объединение циклов (loop fusion)

Метод используется, когда:

1. количество итераций циклов одинаково либо кратно

2. циклы не зависимы по данным, то есть, каждая последующая операция второго цикла не зависит от данных первого цикла

Это позволяет уменьшить дополнительные расходы на циклы. Такая оптимизация не всегда выигрывает в производительности, так как тело цикла может «вывалиться» из кэша, и в этом случае лучше применить [разрезание циклов](#_Разрезание_циклов_(loop).

Инициализировать два массива в отдельных циклах, иногда лучше, чем оба массива одновременно в одном.

Пример:

Количество итераций/размер массива: 10 000

До: (267 мкс)

for (int i = 0; i < iN; i++){

a[i] = b[i] - 5;}

for(int i = 0; i < iN-1; i++){

d[i] = e[i] \* 3;}

После: (125 мкс)

for (int i = 0; i < iN-1; i++){

 a[i] = b[i] - 5;

d[i] = e[i] \* 3;

}

a[iN-1] = b[iN-1] - 5;

5.1.3. Разрезание циклов (loop distribution)

Данный вид оптимизации используется, когда большое тело программы не помещается в кэш, начало вытесняет конец и наоборот, в худшей ситуации данные вытесняются в оперативную память (ОП), что существенно замедляет программу.

Важно понять какая часть/процент попадет в кэш и вывести эту часть в отдельный цикл. Это возможно тогда, когда вычисления внутри тела цикла не зависят друг от друга.

Метод помогает удалить последовательные операторы в отдельный цикл и собрать параллелизуемые операторы в другой цикл.

Применение может мешать зависимости по данным, в таком случае лучше применить [объединение циклов](#_Объединение_циклов_(loop).

Пример:

До:

for (i=0; i < n; i++) {

x[i] = y[i] + z[i]\*w[i]; /\* S1 \*/

a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /\* S2 \*/

y[i] = z[i] - x[i]; /\* S3 \*/

}

После:

/\* L1: parallel loop \*/

for (i=0; i < n; i++) {

x[i] = y[i] + z[i]\*w[i]; /\* S1 \*/

y[i] = z[i] - x[i]; /\* S3 \*/

}

/\* L2: sequential loop \*/

for (i=0; i < n; i++) {

a[i+1] = (a[i-1] + a[i] + a[i+1]/3.0; /\* S2 \*/

}

5.1.4. Перестановка вложенных в друг друга циклов (loop interchange)

Выделяют 2 основные цели:

1. тело самого внутреннего цикла должно находиться в кэш памяти

2. цикл, который находиться внутри, должен быть максимальной длины

Тут мы встречаем противоречие. Важно понять, что даст больший выигрыш.

Больший выигрыш даст «погружение» в кэш память, значит, возможно, вам придется применить [разрезание циклов](#_Разрезание_циклов_(loop).

Рассмотрим несколько случаев:

* «Идеальный». Всё помещается в кэш память. В этом случае самое выгодное сделать самый длинный цикл – внутренним, а самый короткий – самым внешним.
* «Лучший». Вся или большая часть данных находится в памяти непрерывно и самым внутренним циклом проходится, помещаясь в кэш память. Такая ситуация может не соответствовать идеи о том, что самый большой цикл должен находиться внутри. Главная задача «сманить» данные в кэш последовательным чтением, чем больше обращений к этим данным, тем больше выигрыш.
* «Огромные циклы». В любом случае данные не попадут в кэш. Внутрь вложенных циклов разумно вставить самый маленький цикл, потому что он работает с одной областью данных, которая уже находится в кэше. Но если у вас так устроены данные, что из кэша ничего смогли извлечь, тогда есть смысл сделать внутренним самым длинный, а внешним самый короткий, чтобы уменьшить накладные расходы на организацию циклов.
* Важно: соизмеряйте выигрыш от разных оптимизаций, так как универсального приема не существует.

Метод нужен для того, чтобы помочь компилятору распараллеливать код.

Пример:

Количество итераций/размер массива: 200\*200

До: (50516 мкс)  
for(int i = 0; i < iN; i++){  
for(int j = 0; j < jN; j++){  
for(int k = 0; k < kN; k++){  
 c[i][j] = c[i][j] + a[i][k] \* b[k][j];}}}

После: (47038 мкс)  
for(int i = 0; i < iN; i++){   
for(int k = 0; k < kN; k++){ //меняем местами, чтобы данные шли последовательно  
for(int j = 0; j < jN; j++){   
  c[i][j] = c[i][j] + a[i][k] \* b[k][j];}}}

К конечному циклу можно применить [объединение](#_Объединение_циклов_(loop).

5.1.5. Де-факторизация циклов (Loop De-factorization)

Метод используется, когда можем преобразовать математические формулы, которые будут более «дешёвыми» для вычисления.

Математические справочники:

1. Бронштейн И. Н. и др. Справочник по математике.

2. Корн Г., Корн Т. Справочник по математике.

3. Абрамовиц М., Стиган И. Справочник по специальным функциям с формулами, графиками и таблицами

Используя, выигрываем в быстроте, к примеру, умножение ~ 4 такта, а деление ~ 1.5 десятка тактов.

Пример:

Количество итераций/размер массива: 100

До: (66 мкс)

cos (x+ y)

sin (x)

a/x

b/x

c/x

После: (37 мкс)

cos(x)\*cos(y) – sin(x)\*sin(y)

eix-e-ix/2\*i

tmp = 1/x

a\*tmp

b\*tmp

c\*tmp

5.1.6. Разбивка на блоки (Loop tiling)

Если тело цикла сложное, то можно применить эту оптимизацию, и разбить цикл на более маленькие части. Разбиение пространства итерирования цикла приводит к разбиению массива на меньшие блоки, которые помещаются в кэш, что приводит к улучшению использования кэша, снижению количества промахов. Можно применять вместе с [перестановкой циклов](#_Перестановка_вложенных_в), но не во всех случаях.

Данный метод может увеличить накладные расходы на циклы.

Пример:

Количество итераций/размер массива: 100\*100

До: (383 мкс)

for (i = 0; i < N; i++)

for (j = 0; j < N; j++)

c[i] = c[i] + a[i, j] \* b[j];

После: (127 мкс)

for (i = 0; i < N; i += 2)

for (j = 0; j < N; j += 2)

for (ii = i; ii < min(i + 2, N); ii++)

for (jj = j; jj < min(j + 2,N); jj++)

c[ii] = c[ii] + a[ii, jj] \* b[ii];

5.1.7. Циклическое преобразование кода (Loop-invariant code motion)

Применяем, если можно вынести одноразовые вычисления за цикл, чтобы не вычислять их несколько раз. Этот метод значительно повышает производительность.

Пример:

Количество итераций/размер массива: 10 000

До: (125 мкс)

for (int i = 0; i < n; ++i) {

x = y + z;

a[i] = 6 \* i + x \* x;

}

После: (113 мкс)

x = y + z;

t1 = x \* x;

for (int i = 0; i < n; ++i) {

a[i] = 6 \* i + t1;

}

5.1.8. Loop unswitching

Используем, когда условие внутри тела цикла мешает его распараллеливанию. На современных процессорах ради скорости, так как они быстро работают на векторах.

Пример:

Количество итераций/размер массива: 100 000

До: (760 мкс)

int i, w, x[1000], y[1000];

for (i = 0; i < 1000; i++) {

x[i] += y[i];

if (w)

y[i] = 0;

}

После: (527 мкс)

int i, w, x[1000], y[1000];

if (w) {

for (i = 0; i < 1000; i++) {

x[i] += y[i];

y[i] = 0;

}

} else {

for (i = 0; i < 1000; i++) {

x[i] += y[i];

} }

5.2. Индексация циклов к нулю

Важно помнить, что сравнения i (счётчик цикла) с произвольной константой нет, можно сравнивать только с 0. Такое преобразование цикла существенного упростит проверку условия цикла, она будет более быстрой.

5.2.1. Обрыв цикла

Делайте так мало итераций цикла, как это возможно. Можете закончить – заканчивайте. Однако не стоит чрезмерно усложнять условные переходы.

5.2.2. Оператор continue

Существует несколько моментов использования данного оператора:

* Он используется, чтобы не считать лишнего на данной итерации. В этом есть смысл тогда, когда экономите больше 30 итераций, так как continue используется с if и в ином случае это бесполезно
* Он используется, чтобы пропустить недопустимые действия. Это вполне обосновано, однако можно избежать следующими способами: [разрезать](#_Разрезание_циклов_(loop) цикл на два (первый до этого элемента, второй после), если он один; найти закономерность (к примеру, идти с шагом два)

Старайтесь устранить «неизбежный» continue, если не помогает – оставьте всё как есть.

5.2.3. Способы избежать if/else в циклах

1. Если считать лишнее дешевле чем if/else – считайте.

2. Можно перенести if/else вверх или вниз по циклу, чтобы он попал в проверку цикла

3. [Разрезать](#_Разрезание_циклов_(loop) тело цикла на несколько независимых друг от друга циклов, каждый из которых свободен от if/else

4. Заменить на тернарную операцию или switch

5. Уменьшить количество if/else и сделать условие до else более вероятным, чем условие else

Данные рекомендации помогают избежать конструкции if/else в большинстве случаев, однако есть ситуации, когда избежать этого нельзя.

## 5.3. Ветвления

В состав современных процессоров входит модуль предсказания ветвлений – branch predictor. Всякий раз, когда код имеет ветвь (например, if-структуру) или если мы находимся внутри цикла, микропроцессор не знает заранее, какая из двух ветвей нужна для подачи в pipeline. Branch predictor нужен для того, чтобы предсказать по какой ветке пойдет if – останемся ли мы внутри тела цикла или выйдем из него.

Сам процессор работает быстрее, чем оперативная память, из которой он берет данные. Поэтому желательно обращаться к памяти таким образом, чтобы данные заведомо находились либо в регистрах процессора, либо в кэш памяти. В таком случае, процессор может эти данные подтянуть, если будет знать, куда пойдут вычисления.

Вычисления развиваются линейно, поэтому предсказать, что понадобится через некоторое количество тактов процессора достаточно легко, если в этом коде нет ветвлений.

Например, конструкция ветвления с оператором if – самый плохой случай, т.к. оператор if предусматривает различные длинные последовательности вычислительных команд.

Пример конструкции ветвления с оператором  if:

if (num < 10) { *// Если введенное число меньше 10.*

cout << "Это число меньше 10." << endl;

} else { *// иначе*

cout << "Это число больше либо равно 10." << endl;

}

Усовершенствуем программу следующим образом:

if (num < 10) { *// Если введенное число меньше 10.*

cout << "Это число меньше 10." << endl;

} else if (num == 10) {

cout << "Это число равно 10." << endl;

} else { *// иначе*

cout << "Это число больше 10." << endl;

}

`Рассмотрим короткие ветвления, которые в большинстве случаев имеют преимущества над длинными. Их разумно выполнять в виде конструкции – с оператором “?:” ( x = a>b ? y : z; ).

Пример:

if (a > 0)

{x = 1, y = 2, z++ }

else {q = 3 z = 4 y = 5}

Можно переписать с помощью одной тернарной операции.

Для начала нужно найти общие присваемые части в обоих случаях – это z и y. В z выполняются разные операции: z++ - инкрементируется; z = 4 записывается. А y в обоих случаях просто записывается, что довольно удобно.

y = a > 0 ? { x = 1, z++, 2} – непрерывная тернарная операция

y = a > 0 ? ( x = 1, z++, 2) : ( q = 3, z = 4, 5)

В данном методе исключается предсказатель ветвления – просто выполняется логическое выражение. Данный метод выигрывает в скорости выполнения.

y = a > 0 ? 2:5;

x = a > 0 ? 1:x;

z = a > 0 ? z++:4;

q = a > 0 ? q:3

Другим методом оптимизации ветвлений является устранение ветвления, а именно замена на выбор из таблицы. Вместо условного выражения, в котором оба случая являются константами, используется справочная таблица. Если есть оператор, подобный следующему, где c и d представляют константные выражения, а b представляет логическое выражение:

a = b ? c : d

Это эквивалентно следующему:

if (b) a = c;

else a = d;

Заменим следующим эквивалентным и более быстрым кодом:

static const type lookup\_table[] = { d, c };

a = lookup\_table[b];

Условное выражение компилируется в ветвь. Если такая ветвь не очень хорошо предсказана, это займет больше времени, чем таблица соответствия.

a = b1 ? c : b2 ? d : b3 ? e : f;

Это эквивалентно следующему:

if (b1) a = c;

else if (b2) a = d;

else if (b3) a = e;

else a = f;

Заменим следующим эквивалентным кодом:

static constant type lookup\_table [] = { f, e, d, d, c, c, c, c };

a = lookup\_table [b1 \* 4 + b2 \* 2 + b3];

Оператор switch. switch, всегда можно заменить на константный массив.

Под последовательными значениями представляется:

switch (a) {

case 0:

case 1:

case 2:

…

}

Безусловно switch работает быстрее многоуровневых if else, но все-равно это условие. Для повышения производительности всегда можно воспользоваться константным массивом.   
Пример:

int a, b = 0;

/\*switch (a) {

case 0: b\* = 10;

case 2: b\* = 20;

case 3: b\* = 30;

…

}\*/

const int array[3] = {10,0,20,30};

b \*= array[a];

Самый основной (неоптимизированный) способ составления оператора switch - это трактовать его как цепочку if … else if … операторов. Обычный способ, которым компиляторы оптимизируют переключатель, - это преобразовать его в таблицу переходов, которая может выглядеть примерно так:

if (condition1) goto label1;

if (condition2) goto label2;

if (condition3) goto label3;

else goto default;

label1:

<<<code from first `case statement`>>>

goto end;

label2:

<<<code from first `case statement`>>>

goto end;

label3:

<<<code from first `case statement`>>>

goto end;

default:

<<<code from `default` case>>>

goto end;

end:

Одна из причин, по которой этот метод работает быстрее, заключается в том, что код внутри условных выражений меньше (поэтому при неправильном прогнозировании условного кэша будет меньше штраф). Кроме того, «провальный» случай становится более тривиальным для реализации (компилятор пропускает goto end оператор).

Компиляторы могут дополнительно оптимизировать таблицу переходов путем создания массива указателей (на места, отмеченные метками) и использовать значение, которое вы включаете, в качестве индекса в этом массиве. Это исключило бы почти все условные выражения из кода (за исключением того, что было необходимо для проверки того, соответствует ли значение, которое вы включаете, одному из ваших случаев или нет).

Важно отметить, что вложенные таблицы переходов трудно генерировать, и некоторые компиляторы отказываются даже пытаться их создать. По этой причине стоит избегать вложения switch внутрь другого, switch если важен максимально оптимизированный код.

5.4. Оптимизация массивов

Массив – это структура данных, представленная в виде ячеек определенного типа, объединенных одним именем. Массивы используются, когда необходимо обработать большое количество данных. Его имя является постоянным указателем, который указывает на первый элемент массива. Вследствие чего можно использовать указатели и их арифметику.

Обращение к элементам массива выглядеть, как показано в примере 1.

Пример 1.

for (int i=0; i<n; i++)

array [i] = n\_some\_value;

Вместо приведенного выше кода эффективнее сделать, как показано в примере 2.

Пример 2.

int\* arrn=array +n;

for (int \* ptr\_int=array; ptr\_int<arrn; ptr\_int ++)

\* ptr\_int= n\_some\_value;

Причина этого в операциях с указателями. В примере 2 есть указатель на тип int, который берет адрес из имени массива array и увеличивает его для каждого элемента.

В примере 3 рассмотрена очевидная обработка двумерного массива. ( В данном случает его обнуление).

Пример 3

for (int i=0; i<n\_rows; i++) {

for (int j=0; j<n\_cols; j++) {

array[i][j]=0;

}

}

Использование указателей сделает код из примера 3 более эффективным.

Пример 4

int \*p = &array[0][0];

int \*end = &array[0][0] + n\_rows \*n\_cols;

for (p = [\*array](https://vk.com/array); p < end; p++) {

\*p = 0;

}

Используя оптимизацию из примера 4, можно ускорить обработку массива приблизительно в 10 раз. Вместо того, чтобы вычислять позицию каждого элемента с помощью индексов, указатель перемещался по массиву, увеличивая на одну целую позицию после каждой итерации до конца – адреса, который был вычислен один раз, заранее. Конечно, хороший оптимизирующий компилятор упростил бы код примера 4.

В примере 5 требуется посчитать сумму элементов двумерного массива.

Пример 5.

int a[n][m];

int sum =0;

for (int i=0; i<n;i++){

for (int j=0; j<m;j++ )

sum+=a[i][j];

}

Время выполнения фрагмента кода из примера 5 может сильно увеличить при достаточно больших n и m, потому что на каждой итерации цикла при определении адреса [i][j] – го элемента компилятор использует много вычислений.

Зная о том, что элементы массива расположены в памяти последовательно, можно оптимизировать пример 5, введя вспомогательный указатель.

Пример 6.

int \*p= &a[0][0];

float size=sizeof(a)/sizeof(int);

for (int i=0; i<size;i++){

sum+=\*p;

p++;

}

Так же можно использовать эквивалентное объявление указателя \*p= a[0].

Пример 7.

int \*p; //указатель на массив элементов

x=\*(p++);

у=\*(++p);

В примере 7 первое присваивание значительно эффективней, чем второе. В первом случае будет осуществляться разыменование указателя и его инкремента параллельно, а во втором – последовательно.

Еще один совет по оптимизации работы с двумерными массивами.

Количество столбцов двумерного массива желательно должно быть равно степени двойки при небольших типах данных. Это ускорит работу с массивом, поскольку это выравнивает указатели на первые элементы каждой строки, что ускоряет доступ к элементам.

Пример 8.

double array [7][8]; // 8 =23

## 5.5. Оптимизация switch

Задача: имеется много разных параметров (param), от которых зависит решение задачи. Все параметры независимы друг от друга. Каждый из них принимает конечное количество вариантов.

Поставленную задачу можно решить несколькими способами:

I способ. *Кодирование всех возможных вариантов.*

Этот способ применяется только тогда, когда возможно закодировать все возможные варианты.

1. Каждый имеющийся параметр необходимо закодировать достаточным (но не более того) количеством битов. На рисунке 1 представлен пример подобной ситуации.

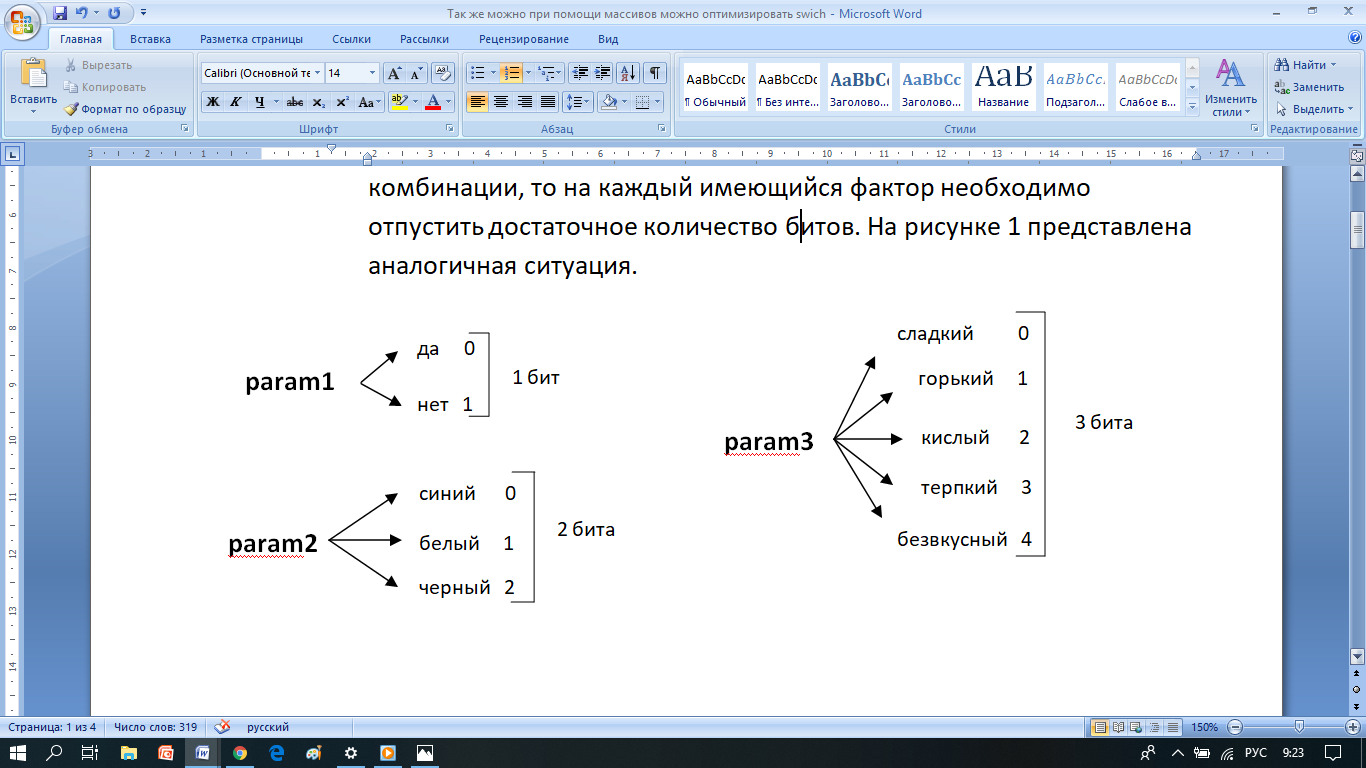


Рис. 5.5.1. Кодирование факторов

2. Из этих коротких целых чисел, каждое из которых содержит маленькое количество битов, при помощи сдвигов собирается одно число (hash), которое в итоге попадает на выражение switch.

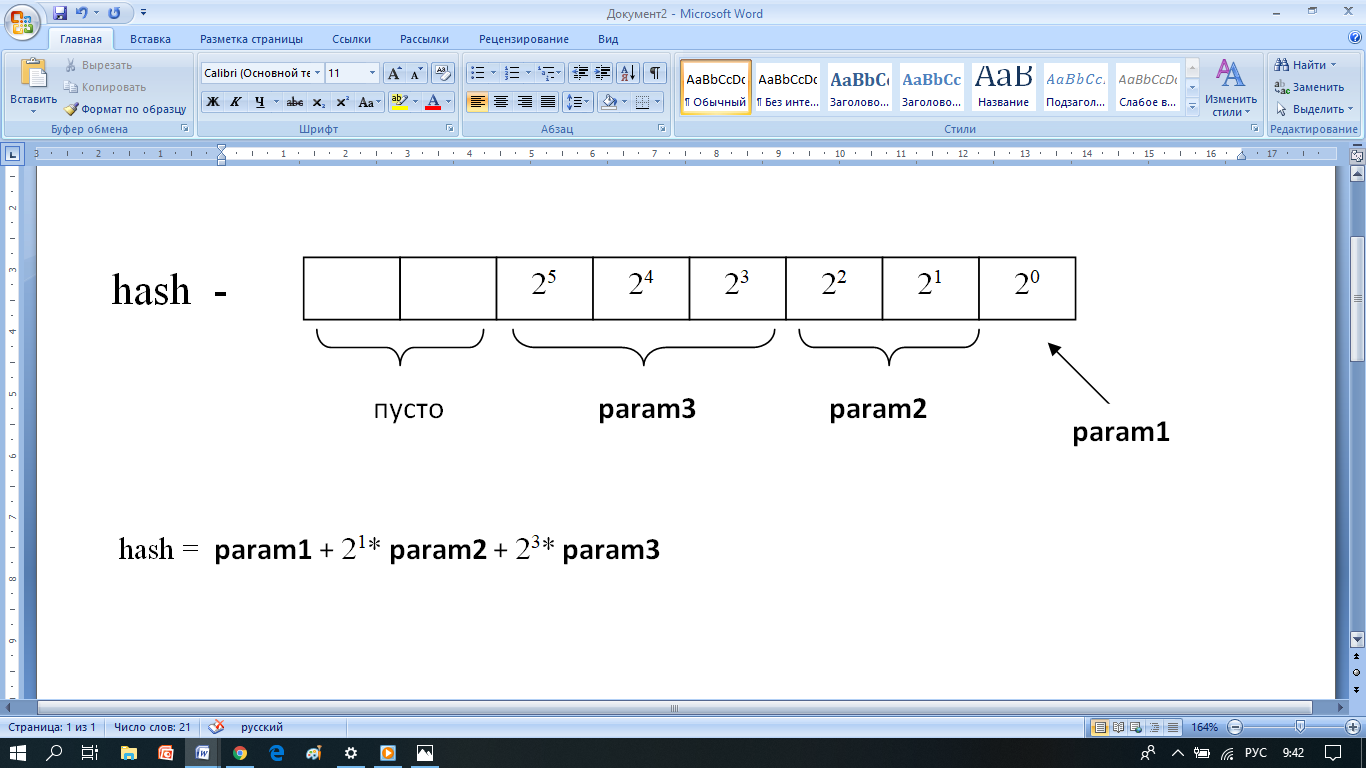


Рис. 5.5.2. Формирование hash

II способ. *Кодирование только возможных сочетаний.*

Этот способ применятся только тогда, когда нет возможности закодировать все варианты, так как некоторые из них невозможны. Например, не все варианты param1 можно сопоставить со всеми вариантами параметра param2 и т.д. В этом случае hash можно образуется следующим образом:

a) логическое выражение

b) a = a0 + (сложное логическое выражение 1)\* a1

b = b0 + (сложное логическое выражение 2)\* b1

c = c0 + (сложное логическое выражение 3)\* c1

Например, если положить a0 = b0 = c0 = 0, а a1 = b1 = c1 = 1, то

hash = a+2\*b+4\*c

с) закодировать возможные сочетания при помощи индексного массива (index arrays/ Look Up Table).

1. Создадим вспомогательный массив. Исходные ситуации описываются числами, которые идут на вход switch. На рисунке 3 представлена визуализация индексного массива.



Рис. 5.5.3. Индексный массив

Это необходимо для того, чтобы непрерывно идущий числовой ряд (элемент 0, 1, …,4) преобразовать в необходимые для switch значения, которые могут идти в различном порядке.

2. Выполняем промежуточную перекодировку (рисунок 5.5.4).

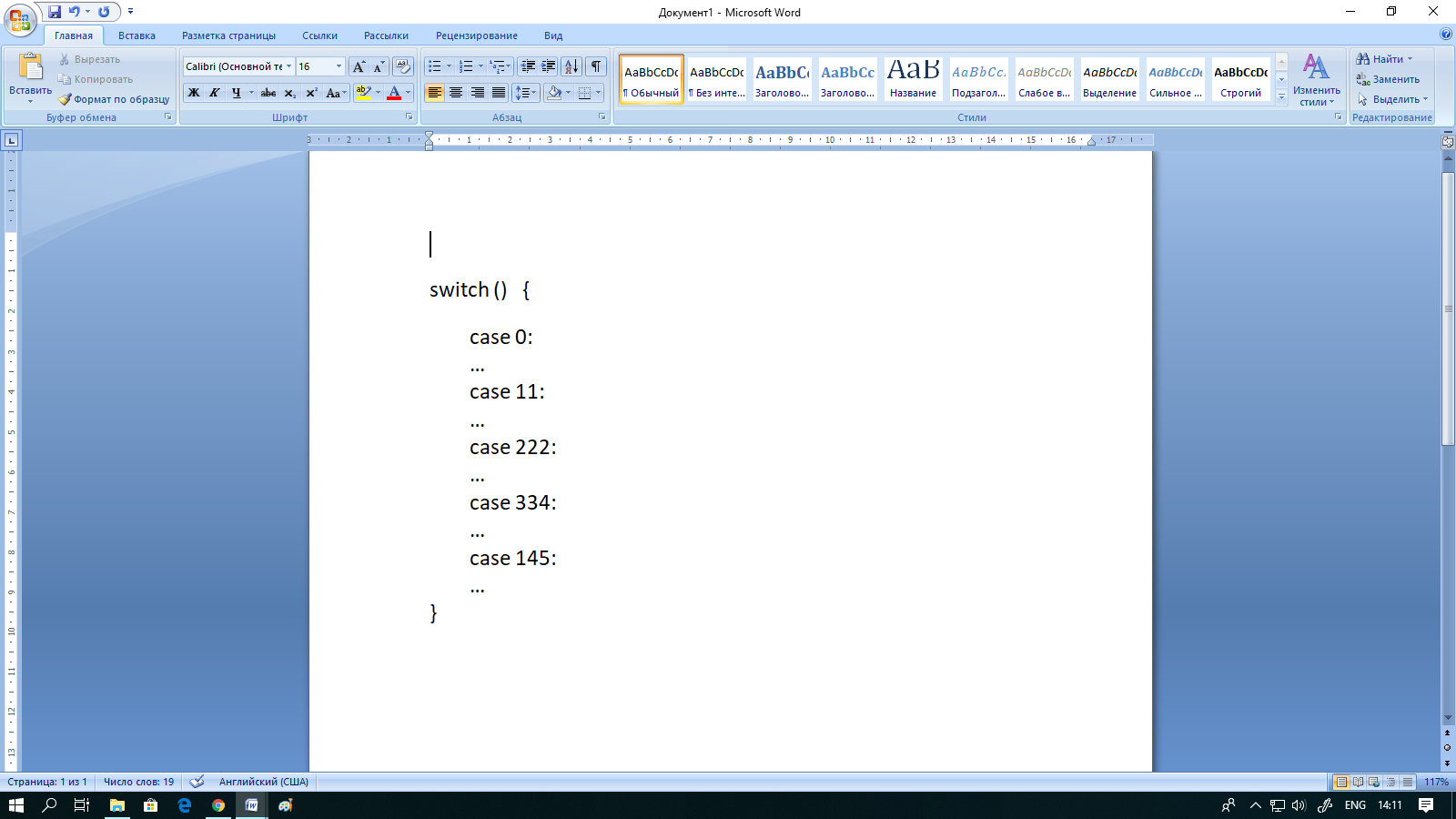


Рис. 5.5.4. Использование индексного массива

Рассмотрим, как можно применить данный метод на практике.

*Задача*. Есть определенное правило, по которому осуществляется транслитерация, т.е. на вход поступает русский текст, известно правило, по которому всякой русской букве или буквосочетанию можно поставить в соответствие латинскую букву или буквосочетание. Необходимо преобразовать входной тест в соответствии с заданным правилом. (Например, ‘с’ ‘s’, ‘ш’ ‘sh’).

Нужно максимально оптимизировать транслитерацию.

1. Формируем вспомогательный массив таким образом, чтобы индексы массива получались посредством вычитания из кода буквы кода буквы ‘а’.

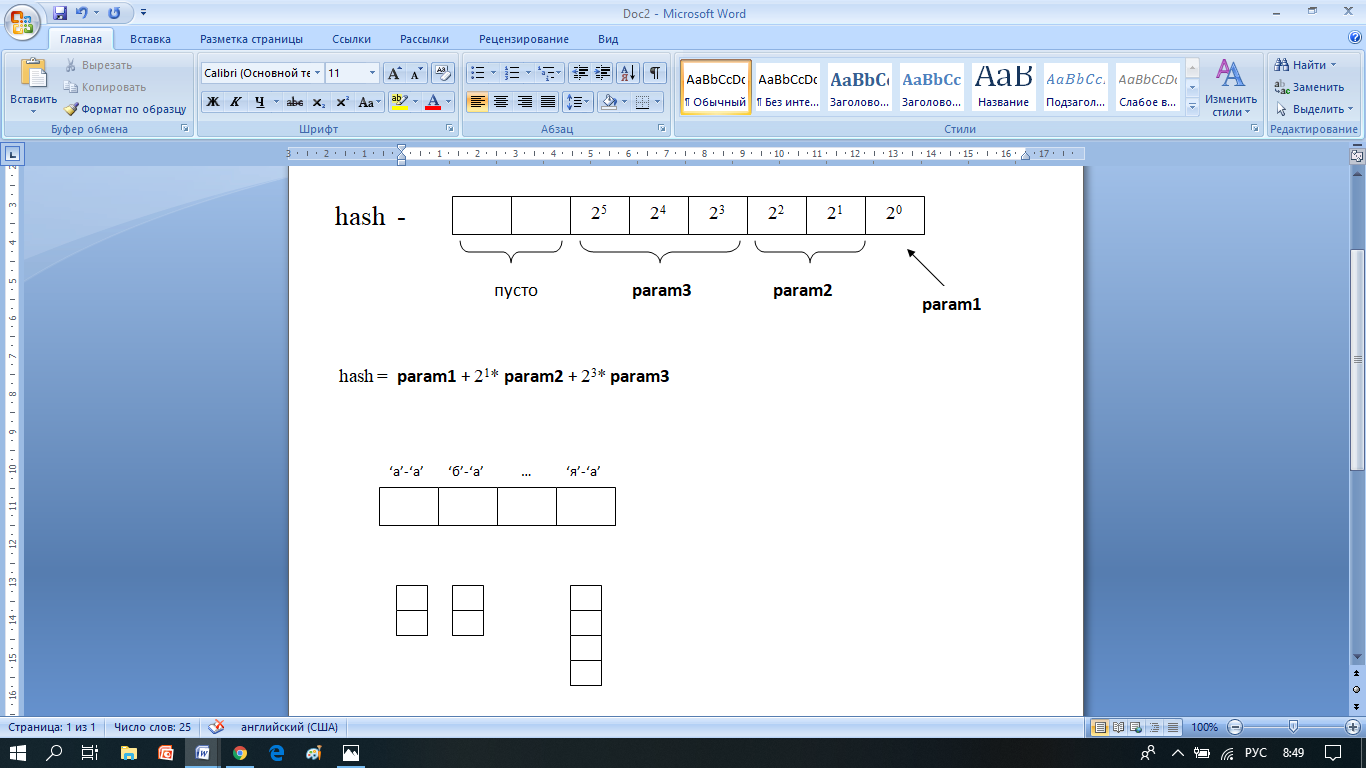


Рис. 5.5.5. Индексы массива

2. В каждую ячейку массива помещаем указатель на массив, где хранится транслитерация отдельно взятой буквы.

3. Массивы с транслитерацией буквы могут быть разной длины, потому что некоторые буквы транслитерируются несколькими символами. Например, ‘щ’ ‘sch’ (Рис. 5.5.6).

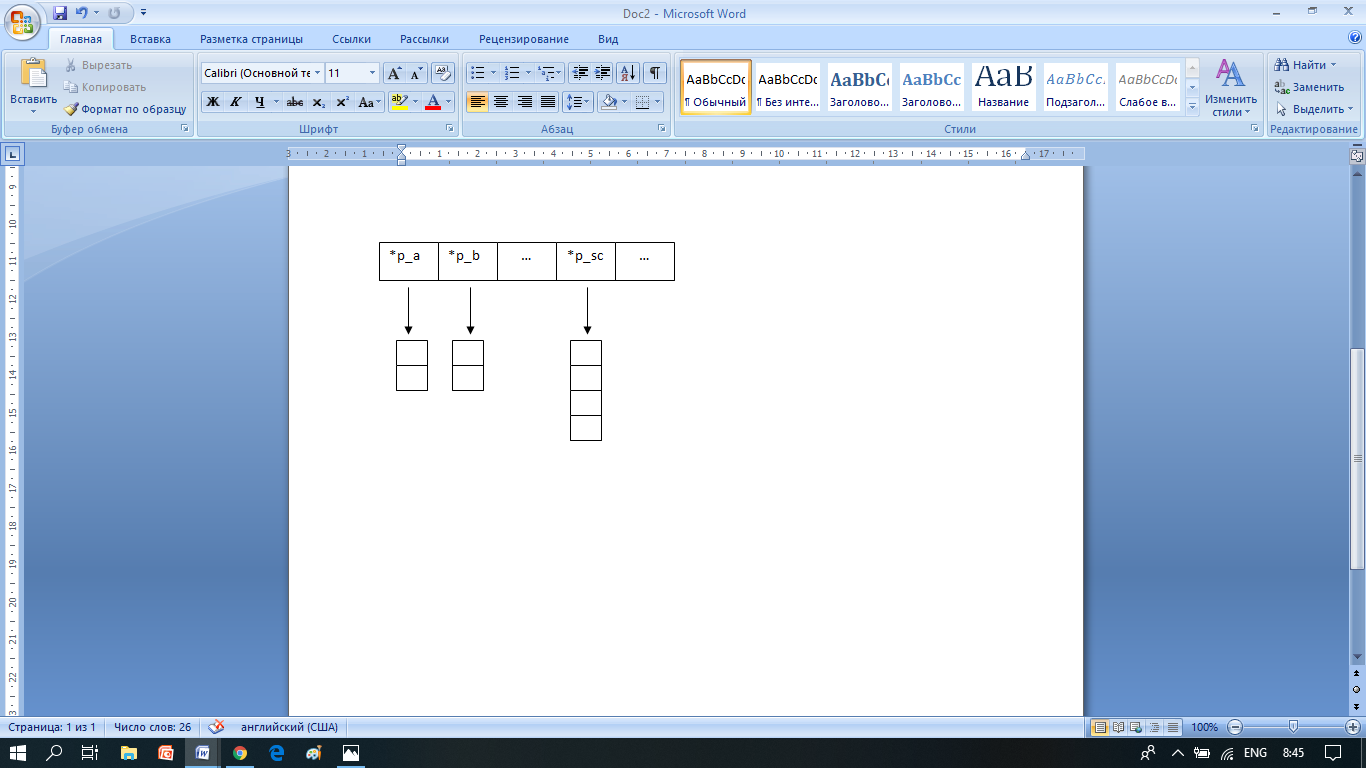


Рис. 5.5.6. Указатели на массивы

4. Результат изображен на рисунке 5.5.7, а в примере 1 представлен фрагмент кода.

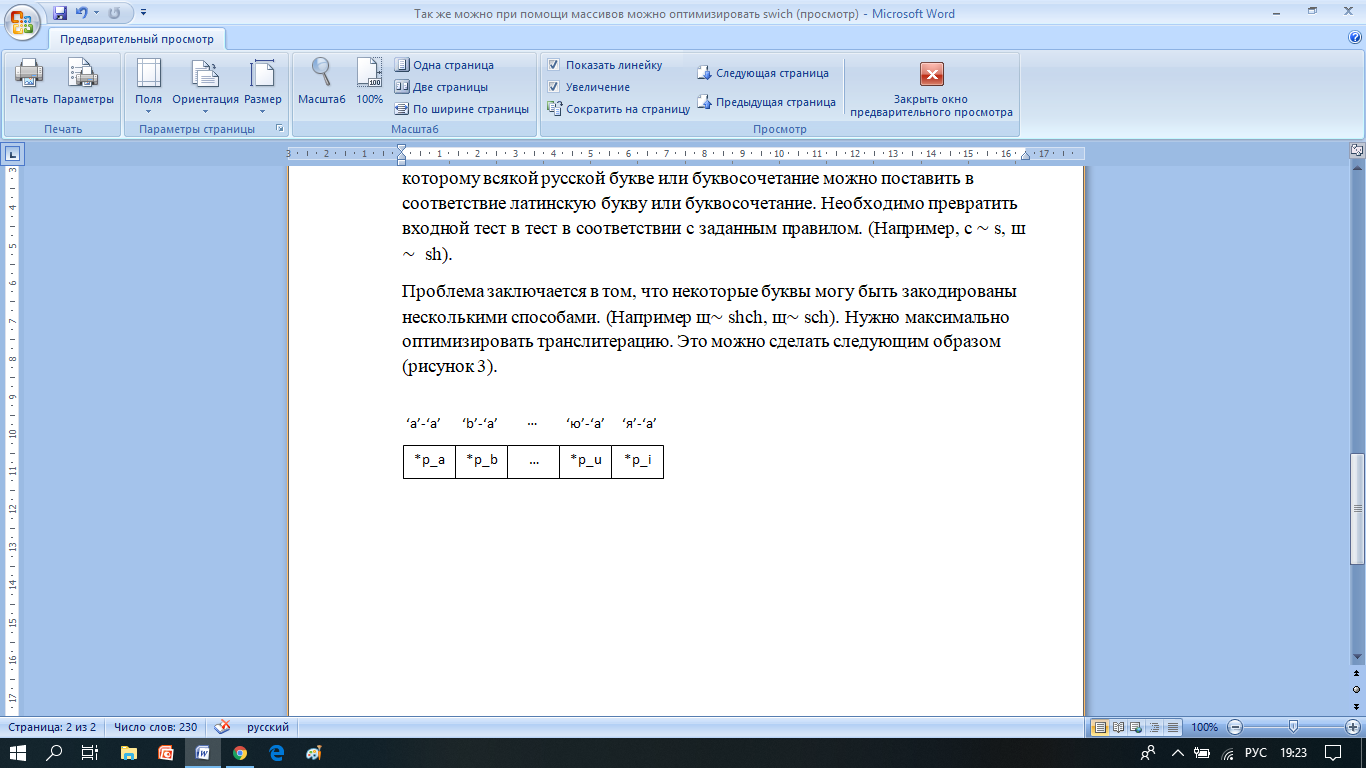


Рис. 5.5.7. Использование индексного массива для транслитерации

Пример 1.

switch (character){  
 case 'а'-'а':  
 \*p\_a;  
 break;  
 ...  
 case 'ж'-'а':  
 \*p\_zh;  
 break;  
 …

}

## 5.6 Устройство кэш памяти

* Кэш память – сверхбыстрая память, расположенная ближе всех к процессору. Обычно она быстрее оперативной памяти примерно в 2 или более раз. Хранит в себе копии часто используемых данных из обычной памяти.
* Кэш память обычно представляет собой трехуровневую систему:

Первый уровень(L1) – самая быстрая память, но самая маленькая. Нет задержек, бывает двух типов: пространство инструкций и пространство данных. На каждое ядро выделено по каждому из этих типов.

Второй уровень(L2) – немного медленней L1, но объем ее при этом в десятки раз больше чем у L1. Хранит в себе менее часто используемые данные, по сравнению с L1, нет разделения на инструкции и данные. Для каждого ядра процессора устанавливается свой кэш L2, то есть у одного ядра нет доступа к кэшу другого.

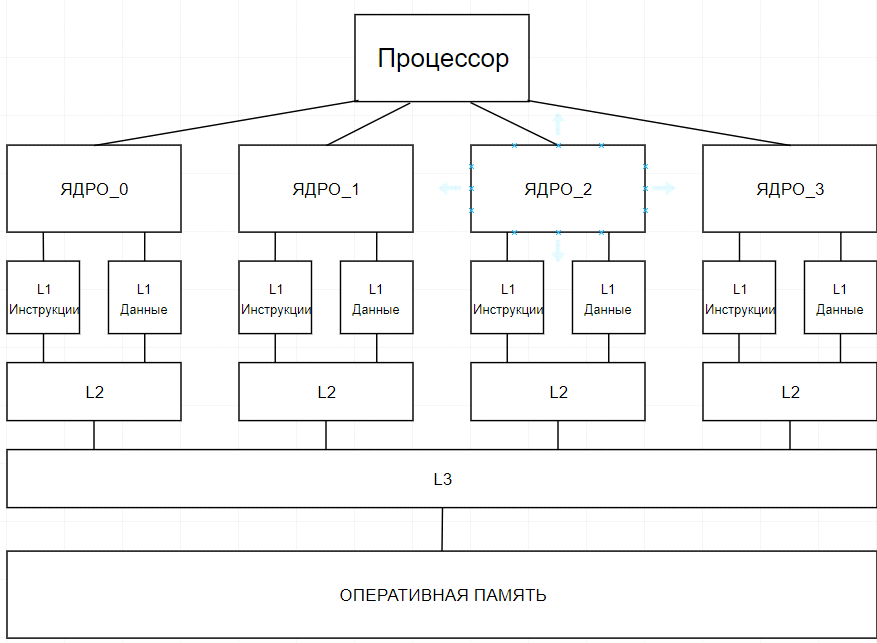


Рис. 5.6.1. Архитектура кэш памяти процессора

Третий уровень(L3) – медленнее, чем L2, но по объему является самой большой. Размер может достигать нескольких мегабайт. Является общей для всех ядер процессора.

* Контроллер кэш-памяти.

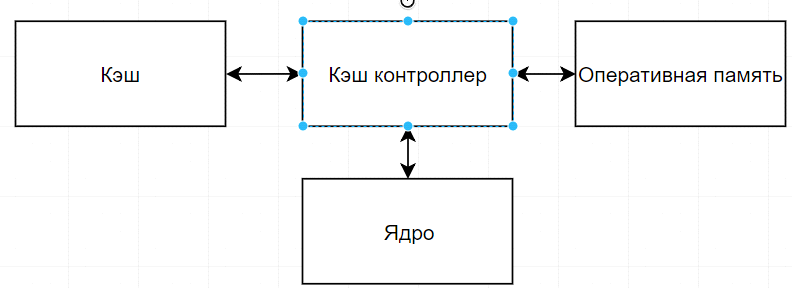


Рис. 5.6.2. Взаимодействие контроллера с процессором и памятью

Это устройство, передающее данные между кэш памятью, оперативной памятью и процессором. Когда ядро обращается за данными, контроллер проверяет их наличие в кэш памяти. Если данные на месте, то происходит немедленная передача данных ядру, в противном случае (*промах кэша-cache miss*) придется ожидать поступления данных из оперативной памяти. Задача контроллера – сделать так, чтобы промахов кэша было как можно меньше.

Организация кэш памяти

Кэш-память — это набор кэш-строк. Каждая кэш-строка хранит блок данных определенного размера и дополнительную информацию (например, тэг). Под размером кэш-строки понимают обычно размер блока данных, который в ней хранится.

Для задания адреса каждого байта памяти требуется x-байтный адрес. Предположим, что наш кэш работает на уровне отдельных байтов, то есть может сохранять в качестве элемента байт оперативной памяти. Тогда каждая единица кэша должна сохранять байт данных оперативной памяти и его x-байтный адрес в оперативной памяти. Получается строка, которая называется кэш­строкой. Адрес сохраняемого байта – тэг(tag). При чтении данных из кэша процессор формирует адрес, который сравнивается кэш-контроллером с тегом кэш­строки. В случае совпадения кэш-контроллер выдает требуемый байт данных, если же совпадения адреса с тегом нет (кэш­промах) — производится обращение к оперативной памяти, что не есть хорошо, так как ядро. Примечание: размер адреса – х, зависит от разрядности системы.

Подробнее о тэгах, их оптимизации.

Тэги очень дорого обходятся для системы по следующим причинам:

* логика сравнения тегов большая (операция XOR занимает лишний бит)
* логика сравнения тегов медленная
* слишком большие затраты памяти на хранение тэгов

Например, в полностью ассоциативном отображении RAM на кэш запись тега в кэш обходится в 30 битов, а запись данных к этому тэгу – 32, то есть 48% памяти кэша расходуется на хранение тэгов!

Уменьшение затрат на тэг: больше данных на один тэг

Сделать кэш-линии больше: увеличить каждую строку в полностью ассоциативном кэше.

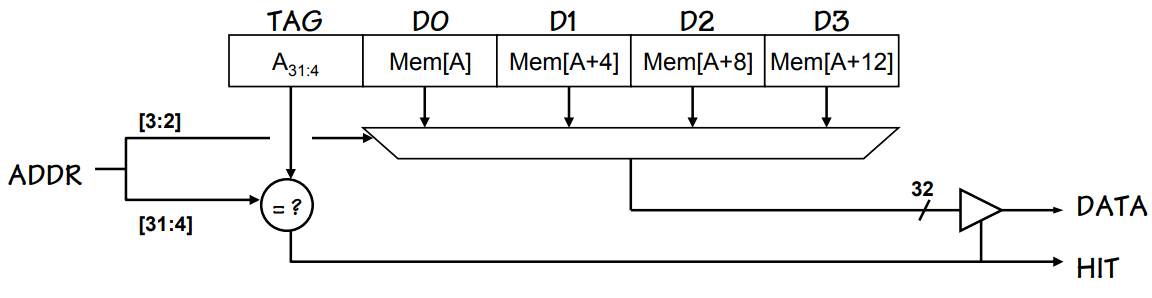


Рис. 5.6.3. Увеличение кэш-линии

* блоки из двоичных слов
* всегда читает/записывает двоичный слово-блок из/в память
* использует пространственную локализацию: соседние слова в блоке, вероятно, доступны
* стоимость: присутствуют некоторые наборы недоступных слов
* большой профит, если путь к памяти широк или последовательный доступ быстрый

В итоге получается, что на хранение тэгов необходимо 32 - 2 = 30 битов, а на хранение блока данных 4\*32 бита. Отсюда следует, что всего 18% от всей кэш памяти использовано на хранение тегов.

Соотношение промахов кэша(cache miss) к размеру блоков

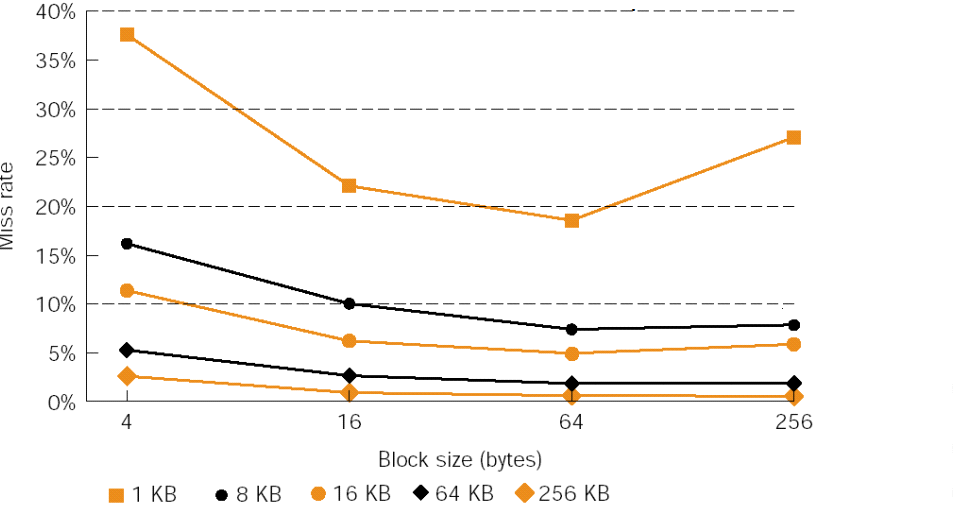


Рис. 5.6.4. Соотношение промахов к размеру блоков

Если блоки данных сделать очень большими, то количество промахов кэша в конечном итоге сильно увеличится и потому следует использовать оптимальный размер блоков.

5.6.1. Три различные стратегии по заполнению кэша

1) LRU (Least-recently used): заменяет объект, к которому дольше всего не обращались и предпочитает самые часто используемые данные. Лучшая производительность / высокая стоимость;

2) FIFO/LRR (first-in, first-out/least-recently replaced): заменяет самый старый элемент в кэше, предпочитает недавно загруженные элементы старым. Низкая производительность / экономичность;

3) Random: заменяет объекты случайным образом, нет фаворитов – все равны, использует псевдослучайный генератор, чтобы получить воспроизводимое поведение. Средняя производительность / самая низкая стоимость, избегает патологических последовательностей, но производительность может варьироваться.

5.6.2. Отображение RAM на кеш-память

1) Direct-Mapped Cache (Прямое отображение): RAM делится на сегменты, размер каждого сегмента равен размеру кэша, а каждый сегмент в свою очередь делится на блоки, размер каждого блока равен размеру кэш строки. Блоки RAM из разных сегментов, но с одинаковыми номерами в этих сегментах, всегда будут отображаться на одну и ту же кэш строку.

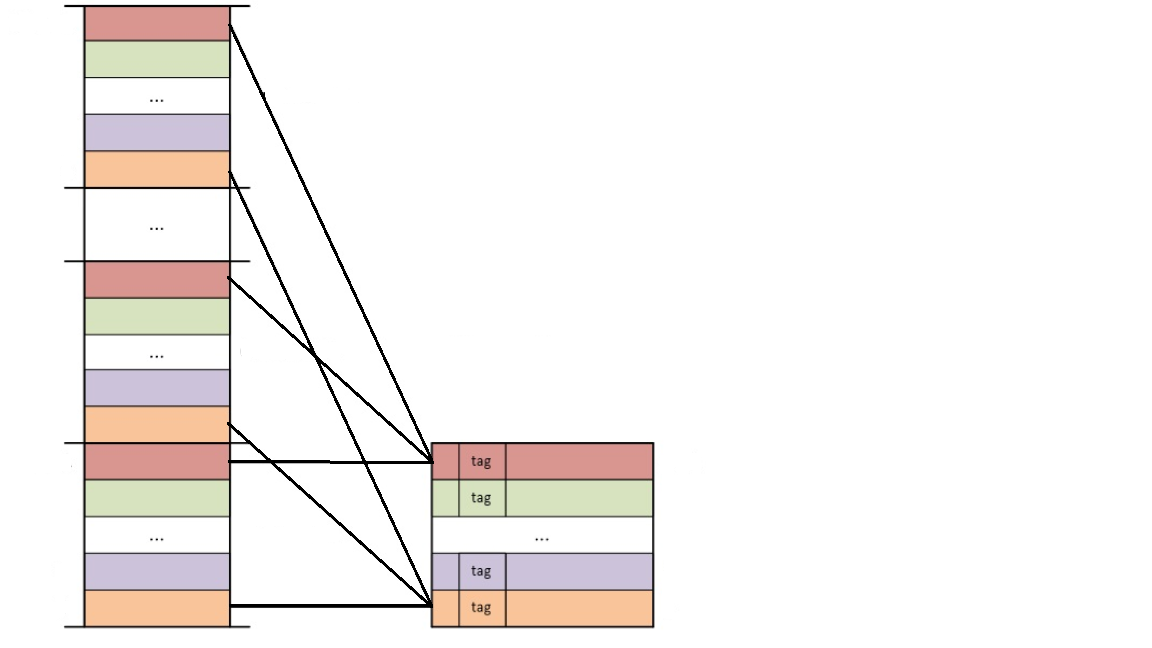


Рис. 5.6.5. Устройство прямого отображения

Использовать стратегию для заполнения нет нужды, так как каждый блок ссылается на определенную кэш строку. Для хранения тэгов/данных используется SRAM.

2) Fully Associative Cache (Полностью ассоциативное отображение): RAM делится на блоки, размер которых равен размеру кэш строки. Каждый блок имеет своё сравнивающее устройство и может отображаться в любую кэш строку. Устройство сравнивает адрес с тегом кэш строки, в случае успеха возвращает данные процессору.

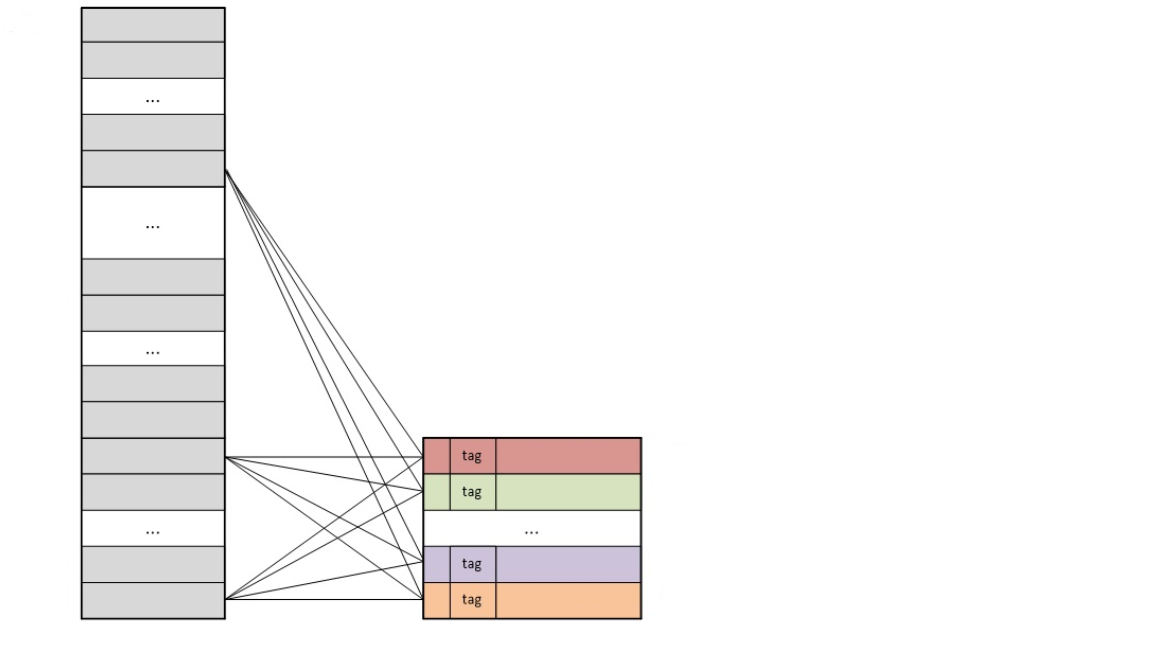


Рис. 5.6.6. Устройство полностью ассоциативного отображения

Используются стратегии заполнения памяти для поиска кэш строки при добавлении новых данных. Для хранения тэгов/данных используются регистры.

3) N-Way Set-Associative Cache (Наборно-ассоциативное отображение): совмещает в себе два предыдущих способа отображения. RAM делится также, как и в прямом отображении, а сам кэш состоит из k кэшей, использующих прямое отображение. Кэш строки, имеющие одинаковые номера во всех каналах, образуют набор.

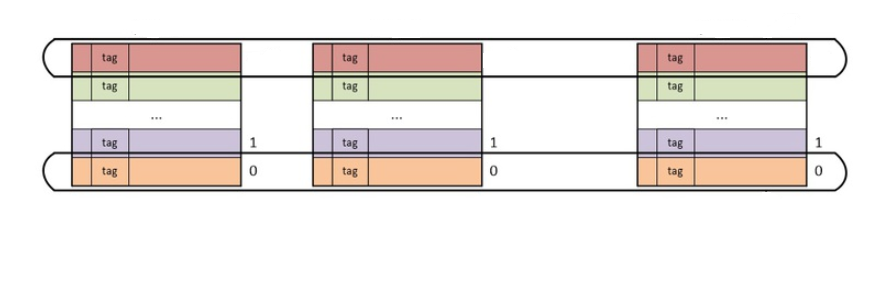


Рис. 5.6.7. Набор в наборно-ассоциативном отображении

Набор представляет собой кэш с полностью ассоциативным отображением. Блоки RAM из разных сегментов, но с одинаковыми номерами в этих сегментах, всегда будут отображаться на один и тот же набор кэша.

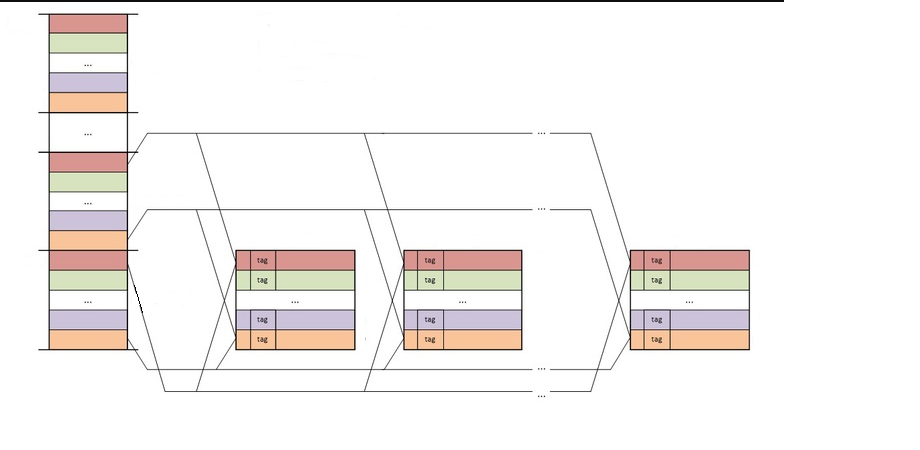


Рис. 5.6.8. Устройство наборно-ассоциативного отображения

Соотношение промахов к ассоциативности

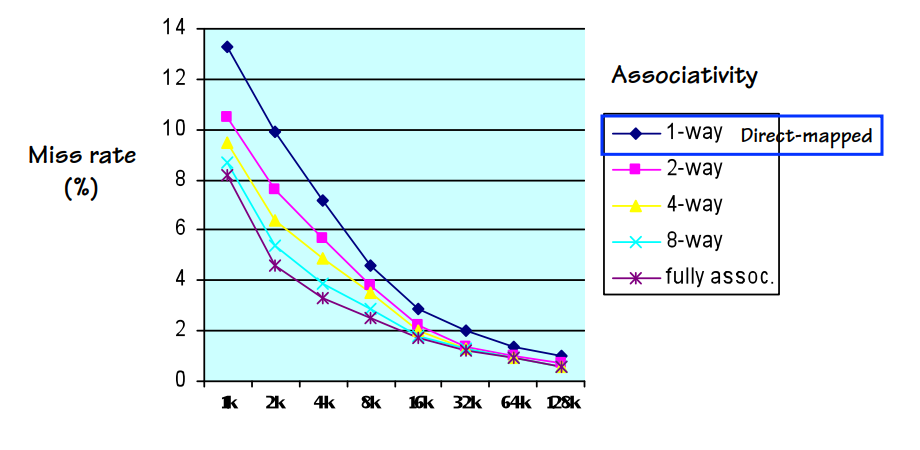


Рис. 5.6.9. Соотношение промахов к ассоциативности

Видим, что более ассоциативное отображение способствует меньшее количество промахов.

## 5.7 Выравнивание данных (data alignment)

Выравнивание ускоряет доступ к памяти за счет генерации кода, в котором на чтение и запись ячейки памяти требуется по одной инструкции. Без выравнивания мы можем столкнуться с ситуацией, когда процессору придется использовать две и более инструкции для доступа к данным, расположенным между адресами, кратными размеру машинного слова. char — особый случай, они занимают ровно одно машинное слово и всегда требуют одинакового количества инструкций для доступа. Поэтому для них нет предпочтительного выравнивания. Сhar может начинаться с любого адреса, однако двухбайтовый short должен начинаться только с четного адреса, четырехбайтный int или float — с адреса, кратного 4, восьмибайтные long или double — с адреса, кратного 8. Наличие или отсутствие знака значения не имеет. Указатели — 32-битные (4 байта) или 64-битные (8 байт) — также выравниваются.



Рис. 5.7.1.

При чтении данных с адреса 0 процессору требуется две операции.

А при считывании данных с адреса 1, так как адрес не кратен размеру считываемых данных, процессору приходится выполнять лишнюю работу, считывая при этом 6 байт вместо 4. Такой адрес называется не выровненным. При доступе к данным по не выровненным адресам процессору приходится делать лишние обращения к памяти, что в итоге приводит к снижению скорости работы программы.

Методы:

1. Использование \_\_declspec( align( # ) )

\_\_declspec( align( # ) ) declarator

# — значение выравнивания. Допустимые записи — целые степени двойки. declarator — данные, которые объявляются как выровненные.

\_\_declspec ( align(#) ) можно использовать при определении struct, union или class либо при объявлении переменной. Однако, \_\_declspec(align(#)) влияет только на выравнивание структуры, а не каждого поля в ней. Если необходимо выровнять каждый член структуры, нужно указать выравнивание на уровне элементов или путем введения дополнительных переменных.

Например, можно запросить 64-байтное выравнивание для массива следующим образом:

\_\_declspec( align( 64 ) ) double a[N];

2. Заполнение

char \*p;

char c;

int x;

Рассмотрим, как располагаются в памяти данные переменные. Память для p начинается с адреса, кратного 4. Выравнивания указателя — самое строгое. Следом за ним идет c. Но четырехбайтный x требует заполнения пустыми байтами. Происходит примерно то же самое, как если бы была добавлена еще одна переменная:

char \*p /\* 4 or 8 байт \*/

char c /\* 1 байт \*/

char pad[3] /\* 3 байт \*/

int x /\* 4 байт \*/

Массив символов pad[3] в данном случае указывает на то, что пространство заполняется тремя пустыми байтами.

Если тип x будет short, который занимает 2 байта, данные будут располагаться так:

char \*p /\* 4 or 8 байт \*/

char c /\* 1 байт \*/

char pad[1] /\* 1 байт \*/

short x /\* 2 байт \*/

С другой стороны, на 64-битной машине эти данные расположатся в памяти следующим образом:

char \*p /\* 8 байт \*/

char c /\* 1 байт \*/

char pad[7] /\* 7 байт \*/

long x /\* 8 байт \*/

Случай, если переменная с меньшим размером будет объявлена в начале:

char c;

char \*p;

int x;

Чтобы эти переменные занимали меньше памяти, можно поменять местами x и c:

char \*p /\* 8 байт \*/

long x /\* 8 байт \*/

char c /\* 1 байт \*/

Путем реорганизации скалярных величин можно сэкономить несколько байт памяти.

## 5.8 Все данные размером машинного слова

В современных процессорах, в отличии от компиляторов, выравнивание данных присутствует всегда, происходит оно по размеру машинного слова. Машинное слово (от англ. «word» - слово) – это количество данных, которые процессор может обработать за одну операцию. Слово – это некоторое количество байт, если точнее, то байт. Его размер указывается в разрядности процессора, которую мы часто слышим, к примеру, Intel Pentium – 32 разрядный процессор, то имеют в виду, что длина машинного слова равняется 32 бита или 4 байта.

Процессор может адресовать данные исключительно с квантом, равным длине его регистра. Размер процессорных регистров общего назначения равен размеру машинного слова этого процессора. То есть, если у вас 64 разрядный процессор, то регистры будут иметь длину 32 бита, либо 64. Например, если у процессора имеется 32 разрядный регистр для хранения Integer’a, то это значит, что все целые числа он хранит посредством погружения в эти 32битные регистры, то есть для хранения типа «двойного длинного слова» – long long (64 бита) будут использоваться два подряд идущих регистра. Но при этом, для хранения адресов может быть 64 битный регистры, это значит, что все адреса могут начинаться только с адреса кратным 8 байт (64 бита). Таким образом разрядность в отношении целых чисел может быть не равна разрядности в отношении адресов, либо чисел с плавающей точкой (float), потому что с ними работают разные регистры.

В итоге, если использовать для данных только регистры размером машинного слова, то вы работаете быстро, всегда ли это оправдано? Не всегда. Как правило, в большинстве ситуаций, стоит гоняться за выигрышем скорости, но при это вы проигрываете в памяти, и избежать этого никак не получится.

## 5.9 Борьба с глобальными объектами

Часто часть данных, которая нужна самым разным функциям в вашей программе оформляют в виде глобальных данных.

Проблема заключается в том, что глобальный объект может использоваться одной и той же функцией (или разными) на разных процессорах.

Рассмотрим 2 случая, чтобы увидеть проблему глубже:

1. Глобальный объект является константой. Вы не можете его поменять и на первый взгляд ничего плохого нет.

Первая проблема. Объект может требоваться разным процессорам, в кэши которых копируется одновременно. Если он будет копироваться в разные кэши разных процессоров, то это приведет к многочисленным копиям глобального объекта. В нем много информации, включаю ту, которая не нужна в данный момент.

Вторая проблема. Вам требуется улучшение исходного текста. К примеру, сделать так, чтобы каждый [инстанс](https://ru.wikipedia.org/wiki/%D0%98%D0%BD%D1%81%D1%82%D0%B0%D0%BD%D1%81) получал свой вариант этого объекта. Тогда логично не делать глобальный объект, а передавать через указать каждую конкретный экземпляр.

2. Глобальные объект не является константой. Проблем возникает больше, и они имеет более значительные последствия.

Первая проблема. К примеру, одна функция стала менять в вашем объекте какое-то поле, и оно не защищено от синхронных изменений, если вы того специально не сделали. Тогда может возникнуть ситуация, когда поменяли одну половину объекта, а вторая осталось прежней. Это называется нарушением атомарности вычислений.

Вторая проблема вытекает из первой, если вы будете защищать объект синхронизации, но он глобальный. В этом случае [мьютекс](https://ru.wikipedia.org/wiki/%D0%9C%D1%8C%D1%8E%D1%82%D0%B5%D0%BA%D1%81) блокирует доступ, а все остальные функции будут ждать. Такую ситуации можно сравнить с большим грузовиком, который встал посередине эстакады. Избежать проблемы можно, разбив глобальный объект и иметь его в нескольких копиях, передавая функции указатель на одну из них.

Как итог, глобальных объектов стоит избежать, даже если мы, с помощью методов выше, сделаем его работу корректной, он не будет эффективным.

Попытайтесь разбить на несколько частей и передавать указателями, не используя глобальных объектов. В ряде случаев, это невозможно, так как универсального совета не существует.

## 5.10 Ускорение вызова функций

Каждая программа состоит из кода и вызова функций. При вызове функций возникают накладные расходы, такие как: передача параметров и управление им, создание и удаление внутренних переменных, возвращение управления и результата. Чтобы минимизировать данные расходы используют такие методы как макросы и inline-подстановки. Они похожи по производимому эффекту, но различаются в плане реализации.

Рассмотрим эти два метода подробно.

5.10.1 Макросы

Макрос – это способ обработки и замены программного кода. Эта замена происходит на этапе препроцессирования, то есть на этапе обработки исходных текстов, до компиляции.

Пусть макрос осуществляет циклическую перестановку (a ). Сложность заключается в том, что для перестановки переменных в данном случае необходимо завести временный объект.

*Способ 1.*

Объявляем временный объект вне тела макроса, например, сделав ее глобальной или в той функции, из которой вызывается макрос.

Пример 1.

Q temp;

# define cyc\_comp (a, b, c) do { temp = a; a = b; b = c; c = temp; \

… \

} while (0);

Чтобы обезопасить код и не допустить путаницы со скобками, макрос вкладывают в цикл do … while. Обосновывается это тем, что если при компиляции обнаруживается ошибка в коде, то компилятор сообщит, что она обнаружена в цикле do … while. Это поможет быстро понять, что ошибка связана именно с макросом.

Достоинства: Эффективность: внутри макроса не происходит выделение памяти, благодаря этому экономится процессорное время.

Недостатки: Временный объект можно будет повредить в результате использования в другом месте программы.

*Способ 2.*

Объявляем временный объект в теле макроса.

Пример 2.

# define cyc\_comp (a, b, c) do { Q temp = a; a = b; b = c; c = temp; \

… \

} while (0);

Достоинства: Безопасность: временный объект не сможет повредиться другими функциями.

Недостатки: Макрос начинает зависеть от типа данных Q и каждый раз, при его вызове, будет выделяться память для создания временного объекта.

Нет конкретного совета, когда надо использовать способ 1, а когда – способ 2, все зависит от решения программиста и от задачи, которую он решает.

*Использование макроса:*

Плюсы: Компилятор в любом случае осуществит оптимизацию, что позволяет организовать макрос любой сложности.

Минусы: При работе с временными переменными необходимо определять их тип. Из-за этого макрос становится зависимым от типа переменных.

5.10.2 Inline-подстановки

Чтобы организовать inline-подстановку необходимо перед типом функции добавить *inline static*.

*Static* означает, что функция исключена из таблицы экспорта. Inline –это просьба к компилятор сделает эту функцию подстановкой. Этого может не произойти, потому что не все функции для этого подходят. Если где-то происходит вызов функции, то ее тело вставится в нужное место с автоматической загрузкой нужных данных, что быстрее их выкладки на стек.

Для примера рассмотри ту же задачу, что и при использовании макроса.

*Способ 1.*

Пример 3.

inline static void cyc\_comp (Q & a, Q & b, Q &c )

{

Q temp = a; a = b; b = c; c = temp; return;

};

Здесь присутствуют те же недостатки и достоинства временного объекта, что и с макросами.

*Способ 2.*

Чтобы избавиться от привязки к типу переменных, используются шаблоны.

Пример 4.

template <class Q> inline static void cyc\_comp (Q & a, Q & b, Q & c)

{

Q temp = a; a = b; b = c; c = temp; return;

};

Достоинства: Временный объект не сможет быть испорчен другими функциями. Также, чтобы избавиться от привязки к типу переменных, используются шаблоны.

Недостатки: Выделение памяти для временного объекта каждый раз при использовании inline подстановки.

*Использование inline-подстановки:*

Плюсы: Более гибкая в использовании, в отличие от макросов.

Минусы: Компилятор может не сделать эту подстановку и использовать ее как обычную функцию с параметрами.

## 5.11 Передача большого количества параметров

В этом разделе также необходимо рассмотреть еще один важный вопрос – передача большого количества параметров в функцию.

Пример 5.

void my\_fun (int a, double b, char c, … и т.д.)

{

…

};

Необходимо данные параметры объединить в класс или структуру. Достаточно будет передать указатель или ссылку на структуру или класс на вход в функцию и указать, можно ли изменять данные параметры. Для наглядности в примере 6 реализована структура, так как ее можно реализовать на С и С++ в отличии от класса.

Пример 6.

struct Q{

int a;

double b;

char c;

…

};

void my\_fun (Q & q)

{

…

};

Достоинства: Отсутствие явной передачи параметров через стек.

Недостатки: Внутри функции придется пройти по указателю на передаваемую структуру или класс, что приведет к затратам на выполнение адресной арифметики и работой со структурой.

Нет явного улучшения при использовании структуры, но код выглядит более читабельно и аккуратно.

## 5.12 Как избегать замедления работы конвейера.

В современных процессорах для организации вычислительных операций используется способ «конвейера команд». Назначение этого способа в том, чтобы ускорить работу процессора. Его идея заключается в том, чтобы распараллелить вычисление нескольких инструкций. Так как каждая ступень обработки требует своих собственных устройств, своих собственных ресурсов, которые, в свою очередь мало пересекаются, что позволяет в обработке держать несколько команд процессора, в разных стадиях выполнения, также менять их местами внутри конвейера.

Рассмотрим ситуации, когда выполнение инструкций может замедлиться.

1) Dependency (от англ. зависимость) - «что-то следующее зависит от предыдущего». Когда инструкция на стадии конвейера зависит от результата предыдущей стадии.

Пример:

Здесь мы видим, что для выполнения второй строки необходимо знать результат первой, также для выполнения третьей строки, необходимо знать результат и первой, и третьей. Мы может пройтись последовательно в конвейере, но тогда некоторые стадии будут ждать другие, что замедлит работу. Решением является переупорядочивание инструкций между стадиями конвейера, которые не зависят друг от друга, то есть, между вычислениями второй и третьей строки поставить команду, не зависящую от “a”.

2) Stall (от англ. заклинивание) Когда происходит обращение к памяти (не кэша) приходится останавливать выполнение всего конвейера. Случается, это потому что ваша инструкция не может выполниться до конца и освободить устройство, на котором она выполняется. При этом следующая инструкция, которая не может получить доступ к этому устройству вынуждена ждать, и также со следующей за ней инструкцией, что приводит к «пробке». Решением данной проблемы является распределение между стадиями так, чтобы положить одни инструкции в кэш, и, пока они будут ждать свои данные, остальные могли выполняться.

Также есть stall, связанный с устройствами выполнения операции, например, устройство для выполнения операций с плавающей точкой, устройства, которые занимаются ветвлениями. Несмотря на то, что в процессоре может быть много ядер, количество таких устройств определенно, то есть несколько ядер разделяют между собой общее устройство, и появляется проблема, когда несколько ядер хотят использовать одно и то же устройство. Допустим, несколько стадий используют устройство с плавающей точкой, и, чтобы не создавать stall, нужно переупорядочить операции так, чтобы операции с целочисленными шли рядом с операциями с плавающей точкой, указателями и т.д.

3) If. Блок Prefetch, который позволяет обнаружить инструкции до их входа в конвейер. Prefetch заранее готовит данные, нужные инструкции, также заранее подбирает и подсчитывает из памяти команды нужные этой инструкции. Но при возникновении if, появляется ветвление, что приводит к необходимости делать speculative execution (умозрительное заключение). Избежать этой проблемы можно, если мы будем считать ветви условия на разных ядрах, проблема этого решения в том, что мы, как минимум, проделаем половину работы просто так (в случае двух ветвей условия), что является энергозатратным.

Однако, данное решение относится больше к сильным процессорам, а для таких CPU как Pentium M(Mobile) или ARM CPU’s, предназначенных для мобильных устройств энергозатраты будут слишком велики, что делает данное решение непригодным. В этой ситуации остается только стараться избегать If’ов.

## 5.13 Виртуальная машина

В случаях, когда необходимо запустить некое приложение, разработанное под ОС (операционную систему), отличной от установленной на рабочей машине и нет возможности/желания ставить ее параллельно, есть специальный инструмент, именуемый виртуальной машиной. Виртуальная машина(ВМ) – процессы, запущенные из хостовой (базовой) ОС, установленное на «голое железо» компьютера. Они формируют некую среду, изображающая себя «голым железом». Такая среда имеет все средства для получения для себя ресурсов: выделение памяти, доступ к графическому адаптеру, доступ к сетевому адаптеру и т.д. Для хостовой системы ВМ – всего лишь программа, а для своих внутренностей – это самое настоящее железо.

Внутрь ВМ ставится одна или несколько внутренних для нее ОС, которые называются гостевыми. Например, рядом со внешней ОС почти невозможно установить другую ОС, тогда можно на хостовой системе запустить ВМ, внутри которой можно без проблем установить две разные ОС, которые не «видят» друг друга и внешнюю машину. Иначе говоря, они полностью изолированы.

Также, в некоторых программах есть специальных режим песочницы (sandbox). Это своего рода примитивная ВМ, возможности которой сильно ограничены, но можно делать что угодно без последствий для основной системы. Такое решение используется в антивирусах для просмотра поведения потенциальной угрозы посредством ее запуска в песочнице без последствий. Отсюда следует, что хорошая ВМ должна эмулировать некое подобие жестких дисков. ВМ создает большие файлы в хостовой системе, в рамках которых изображаются жесткие диски для ОС внутри ВМ. Из них увидеть внешнюю ОС невозможно.

Контейнер – созданная в докере имитация окружения ОС + переменные среды, библиотеки. Таким образом, в докере может существовать много различных контейнеров.

Сравнительная таблица докера и ВМ:

|  |  |  |
| --- | --- | --- |
| **Свойство** | **Докер** | **ВМ** |
| **Изолированность** | Нет полной изоляции внутренней системы и внешней. Нет возможности имитировать другую архитектуру, в силу того, что программа докера использует те же самые библиотеки, ресурсы, которые предоставляет хост | Полная изоляция от внешней ОС |
| **Ресурсы и библиотеки** | Поскольку докер «заслоняет» собой некоторую часть окружения хостовой системы, делает некоторую часть библиотек невидимым, вместо них для программы он предлагает другой набор (например, другие версии). | Использование всех ресурсов гостевой ОС |
| **Скорость работы** | Эффективность в отношении процессорного времени близка к 100%. | Виртуальная машина в среднем замедляет содержимое гостевой системы до десятков процентов, потому что оболочка ВМ – сложная, эмулирует каждое событие, то есть на поступающие изнутри запросы выдает ответ, изображающий что-то правдоподобное поведению оригинального железа. Это требует больших накладных расходов (процессорное время, память). |
| **Сохранение результата работы** | В случае с докером все, что делалось в его контейнере, не сохранится. | Поработав с ВМ, вы можете сохранять или не сохранять результат работы. |
| **Сценарий использования** | Поддержка многих разных сред для отладки, написания ПО | Необходимость полностью изолировать систему при неизбежности работы с опасным ПО |

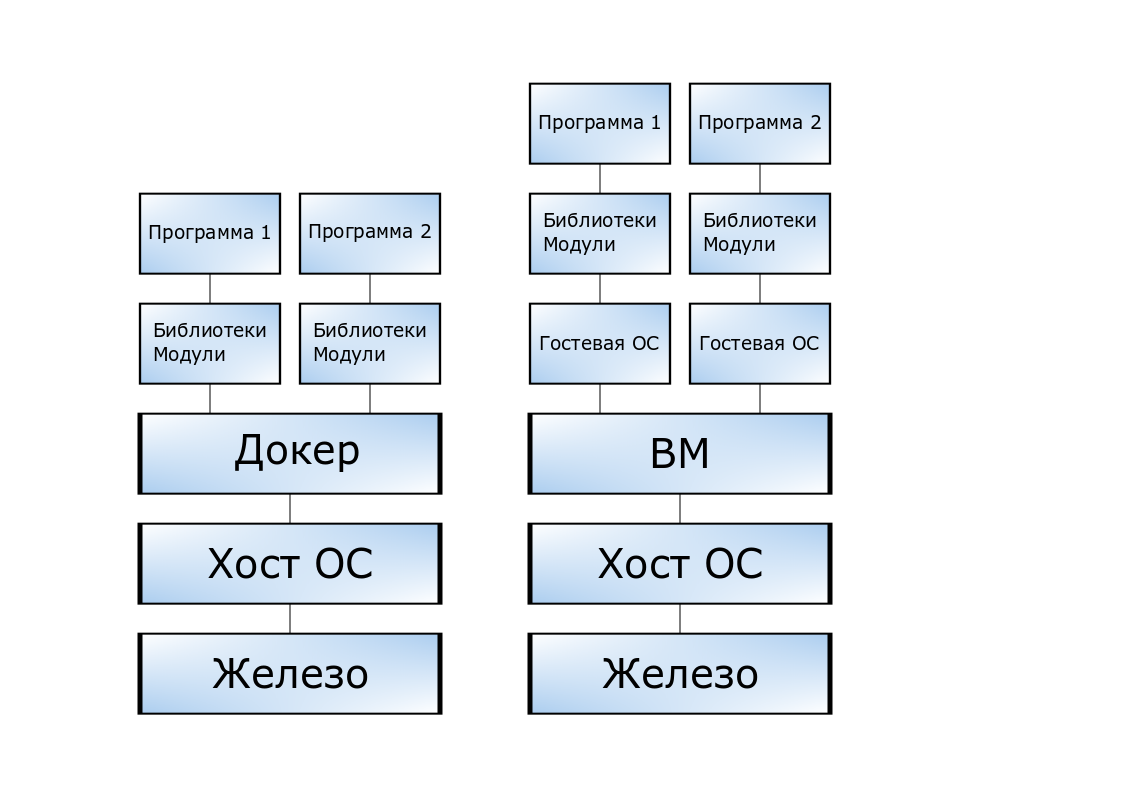


Рис 5.13.1. Сравнительные диаграммы докера и ВМ

## 5.14 Хранение или вычисление заново?

Иногда в коде возникает необходимость вычислять одно и тоже значение от какой-нибудь функции несколько раз. Эту процедуру можно реализовать двумя способами.

1 способ.

Необходимо завести временную переменную, в которую один раз запишется вычисленный результат. И далее в коде используется уже имеющийся готовый результат.

Пример 1

double PI = 3.14;

double t = tan (45 \*PI/180);

for (int i=0; i<n;i++)

{

…

a = t;

…

}

2 способ.

Каждый раз вычислять заново результат функции.

Пример 2

double PI = 3.14;

for (int i=0; i<n;i++)

{

…

a = tan (45 \*PI/180);

…

}

Применение:

1 способ.

* Если таких значений много и они идут поблочно. Считав один раз в кэш значения, не надо их пересчитывать.
* Если этих значения мало и они встречаются очень часто. Желательно попытаться поместить переменную, в которую будет записываться результат, в регистр.

2 способ.

* Если таких значений мало и они встречаются нечасто. Не надо хранить в памяти временную переменную с результатом. Тяжеловесная функция вычисляется за несколько десятков тактов (~50). Адресация в памяти – порядка 100 тактов, если данный результат не попадает в кэш.

## 5.15 Утечка памяти

Утечка памяти (memory leak) — процесс неконтролируемого уменьшения объёма свободной оперативной или виртуальной памяти компьютера, связанный с ошибками в работающих программах, вовремя не освобождающих ненужные участки памяти, или с ошибками системных служб контроля памяти.

Если программист забудет выделить память под какой-то объект и что-то попробует записать, процессор определит факт, что в невыделенную область памяти пишут и вызовет исключение, которое программа вынуждена будет обработать. Обратная ситуация, если память была выделена, но не очищена после использования. Пока таких «ненужных» данных находится в памяти немного, программа может продолжать исправно работать до тех пор, пока они не заполнят лимит выделенной для приложения оперативной памяти. Это приводит к замедлению программы и всей операционной системы или полному отказу программы вследствие нехватки памяти.

Если вы запустили диспетчер задач и, наблюдая за приложением, которое простаивает, заметили, что оно потребляет все больше памяти – это верный признак, что в приложении есть ошибки такого рода.

Могу ли утечки возникнуть из-за работы со стеком? На стеке автоматически создается новый объект, когда открывается новый блок и происходит определение этого объекта и по закрытии блока объект автоматически удаляется своим деструктором. Таким образом, если все в порядке с логикой программы, всякая память, очистится, когда она не нужна, то есть, когда закрылся соответствующий блок.

Иная ситуация при работе с кучей. Утечка происходит, когда программа теряет адрес некоторой динамически выделенной части памяти, прежде чем вернуть её обратно в операционную систему. Когда это происходит, то программа уже не может удалить эту динамически выделенную память, поскольку она больше не знает, где та находится. Операционная система также не может использовать эту память, поскольку считается, что та по-прежнему используется вашей программой.

Хотя утечка памяти может возникнуть и из-за того, что указатель выходит из области видимости, возможны и другие способы, которые могут привести к утечкам памяти. Например, если указателю, хранящему адрес динамически выделенной памяти, присвоить другое значение:

int value = 7;

int \*ptr = new int; // выделяем память

ptr = &value; // старый адрес утерян - произойдет утечка памяти

Это легко решается удалением указателя перед операцией переприсваивания:

int value = 7;

int \*ptr = new int; // выделяем память

delete ptr; // возвращаем память обратно в операционную систему

ptr = &value; // переприсваиваем указателю адрес value

Кроме того, утечка памяти также может произойти и через двойное выделение памяти:

int \*ptr = new int;

ptr = new int; // старый адрес утерян - произойдет утечка памяти

Адрес, возвращаемый из второго выделения памяти, перезаписывает адрес из первого выделения. Следовательно, первое динамическое выделение становится утечкой памяти.

Чтобы найти место утечки в программе нужно очень хорошо представлять логику ее работы и постоянно следить за выделением и освобождением памяти.

Программа «» позволяет найти ошибки работы с памятью. Вместо использования стандартной библиотеки языка память заказывается через специальную виртуальную машину: она контролирует и «запоминает» куда и сколько памяти ушло и сколько возвращено в каждый момент времени работы программы.

## 5.16 Треды

Современные операционные системы и микропроцессоры давно поддерживает многозадачность и вместе с тем, каждая из этих задач может выполняться в несколько потоков. Это дает прирост производительности вычислений и позволяет лучше масштабировать пользовательские приложения и сервера, но за это — усложняется разработка программы и ее отладка.

Как минимум, в процессе должен быть хотя бы один поток (но их может быть несколько), так как в конкретном потоке происходит выполнение логики, которое описано у нас в программе. Поток определяет последовательность исполнения кода программы, даже если программа содержит функцию, либо создание какого-то класса, выполнение методов этих классов, либо цикл – все эти действия будут выполняться последовательно, согласно предусмотренной логике. С помощью создания дополнительных потоков и передачи в них части какой-то логики, например, выполнения какой-то функции, мы можем обеспечить выполнение нашего кода не последовательно, а параллельно или асинхронно, что ускорит выполнение нашей программы.

Рассмотрим пример:

#include <iostream>

#include <thread>

#include <chrono>

int main()

{

setLocale(LC\_ALL, "ru");

cout << “START MAIN” << endl;

this tread::sleep\_for(chrono::milliseconds(1000));

cout << this\_thread::get(id) << endl;

cout << “END MAIN” << endl;

return 0;

}

get(id) – функция получения идентификатора потока.

sleep\_for(chrono::milliseconds) – метод, который приостанавливает работу текущего потока на определенный период времени, то есть задерживает выполнение нашего кода. Благодаря такой возможности можно симулировать длительное выполнение какой-либо сложной задачи.

Рассмотрим следующий пример:

#include <iostream>

#include <thread>

#include <chrono>

int main()

{

setLocale(LC\_ALL, "ru");

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tmain” << endl;

this tread::sleep\_for(chrono::milliseconds(3000));

}

return 0;

}

Строка с функцией sleep\_for(chrono::milliseconds(3000)) задерживает выполнение программы на 3 сек, если эту строку закомментировать, то программа выполнится мгновенно.

Для того, чтобы отследить, как это работает, что за чем выполняется, и в чем их преимущество – потребуется замедлять их выполнение.

Рассмотрим метод, который будет эмулировать сложную задачу, которая будет выполняться длительный период времени

#include <iostream>

#include <thread>

#include <chrono>

void Dowork()

{

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tDowork” << endl;

this tread::sleep\_for(chrono::milliseconds(1000));

}

}

int main()

{

setLocale(LC\_ALL, "ru");

DoWork()

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tmain” << endl;

this tread::sleep\_for(chrono::milliseconds(500));

}

return 0;

}

Как результат: сначала выполняется метод  DoWork(), а затем main. Обратим внимание, что значение id потока при выполнении этих двух методов одинаковое – это означает, что эти два метода выполнялись в одном потоке.

Если изменить sleep\_for(chrono::milliseconds(500)) на sleep\_for(chrono::milliseconds(1000)), то одна итерация метода DoWork() занимает 1 сек. Main работает немного быстрее.

Таким образом, для метода DoWork() была симулирована нагрузка.

В такой ситуации, когда есть долго выполняющаяся задача, которая блокирует работы с основным потоком, мы модем использовать дополнительный поток, чтобы распараллелить эти две задачи. Метод DoWork() можем выполнять отдельно.

#include <iostream>

#include <thread>

#include <chrono>

void Dowork()

{

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tDowork” << endl;

this tread::sleep\_for(chrono::milliseconds(500));

}

}

int main()

{

setLocale(LC\_ALL, "ru");

thread th(DoWork);

for (size\_t i = 0; i < 10; i++)

{

cout << “ID потока = “ << this\_thread::get(id)<< “\tmain” << endl;

this tread::sleep\_for(chrono::milliseconds(500));

}

th.join();

return 0;

}

С помощью этого кода программа будет запускаться в двух потоках. Метод DoWork выполняется одновременно с методом main.

Метод th.join() позволяет дождаться выполнения той задачи, которую мы поставили в отдельном потоке.

Таким образом, задача была распараллелена.

В нескольких задачах мы можем запускать столько потоков, сколько нам нужно. Выигрыш будет в том, случае, если процессор сможет обеспечить преимущество выполнения такой задачи.

# Приложение

Специализированные алгоритмы и вычисления с повышенной точностью.

Существуют задачи, такие как вычисления криптографических ключей, задачи астрономии и др., которые требуют большой точности. Точность целых чисел измеряется до . Для чисел с плавающей точкой необходимо изобразить много разрядов целых чисел и большую мантиссу. Существует большое количество библиотек, таких как Bignum, которые выстраивают эти данные друг за другом. И есть специальные разработанные алгоритмы, особенность которых – использование нетривиальных констант, заданных с фиксированной точностью, позволяющие работать с такими кортежами, интерпретируя их как число либо целое, либо с плавающей точкой. Процесс занимает длительное время, но повышает точность операции. Метод Рунге-Ромберга, Ньютона, Лакс-Вендорфа, дихотомии – частные случаи, но все же реализующие невысокую точность по сравнению с необходимой.

Представление данных с фиксированной точкой и правила вычислений и преобразования.

q – основание системы счисления

r – количество разрядов под дробную часть

n – количество разрядов под целую часть

, где - коэффициенты

*,*

При представлении числа в форме с фиксированной точкой указываются знак числа (*sign a*) и модуль числа (*mod a*) в q-ичном коде. Место точки (запятой) постоянно для всех чисел и в процессе решения задач не меняется. Знак положительного числа кодируется цифрой «0», а знак отрицательного числа — цифрой «1».

*Определение 1*

Код числа в форме с фиксированной точкой, состоящий из кода знака и q-ичного кода его модуля, называется прямым кодом.

*Определение 2*

Разряд прямого кода числа, в котором располагается код знака, называется знаковым разрядом кода.

Разряды прямого кода числа, в которых располагается q–ичный код модуля числа, называются цифровыми разрядами кода. При записи прямого кода знаковый разряд располагается левее старшего цифрового разряда и обычно отделяется от цифровых разрядов точкой.

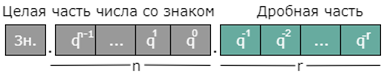
****

Рис. 1. Разрядная сетка компьютера для размещения чисел в форме с фиксированной точкой

На рисунке показано n разрядов целой части числа и r разрядов — для дробной части числа. Использование формы с фиксированной точкой для представления чисел с целой и дробной частью в компьютерах не используется. В основном, используются компьютеры либо с дробной арифметикой (n=0), либо с целочисленной арифметикой (r=0).

Форма представления чисел с фиксированной точкой упрощает аппаратную реализацию компьютера, уменьшает время выполнения машинных операций, однако при решении задач необходимо постоянно следить за тем, чтобы все исходные данные, промежуточные и окончательные результаты находились в допустимом диапазоне представления. Если этого не соблюдать, то возможно переполнение разрядной сетки, и результат вычислений будет неверным. От этих недостатков в значительной степени свободны компьютеры, использующие форму представления чисел с плавающей точкой.

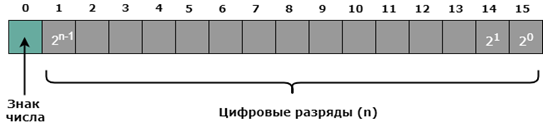
****

Рис. 2. Пример числа с фиксированной точкой. Ячейка с записью целого числа

К достоинствам использования чисел с фиксированной точкой относятся простота выполнения алгоритмов арифметических операций и высокая точность представления чисел. К недостаткам – небольшой диапазон представления чисел.

Пример числа с плавающей точкой:

2,62510 = 10,1012 = 0,101012\*2210 (нормальная форма)

Если число х положительное, то его прямой код представлен как х2.

Если число х отрицательное, то его прямой код представлен как 1 – х2.

Правила вычислений:

Сложение

Операции сложения и умножения обычно выполняются над двоичными, числами, являющимися правильными дробями. Последовательность выполнения операции сложения, следующая:

* исходные числа записываются в принятом для данной машины коде;
* производится поразрядное сложение кодов чисел, включая и знаковые разряды;
* производится анализ на переполнение разрядной сетки. В случае переполнения поступает сигнал на прерывание программы.

Сложим числа: x=0,101001, y=0,011011.

Сочетание 01 в знаковых разрядах свидетельствует о переполнении.

Умножение

Умножение производится в столбик, как обычное умножение: путем сложения со сдвигами и последующим сложением этих чисел. Знак результирующего числа определяется путем сложения знаков изначальных чисел.

Пример: x=0,1101, y=1,1011. (Т.е. x – положительное число, у – avx отрицательное). Знак их произведения: 0 + 1 = 1 ⇒ число отрицательное.

Ответ: ху = 1,1000111

# 

# Список сокращений

|  |  |  |
| --- | --- | --- |
| АВМ | — | аналоговые вычислительные машины |
| АЛУ | — | арифметико-логическое устройство |
| АЦП | — | аналого-цифровой преобразователь |
| ВМ | — | вычислительные машины |
| ВУ | — | внешним устройством |
| ГВМ | — | гибридные вычислительные машины |
| ИВВ | — | интерфейса ввода-вывода |
| ОВВ | — | окружение времени выполнения |
| ОЗУ | — | оперативное запоминающее устройство |
| ОСРВ | — | операционные системы реального времени |
| ПДП | — | прямой доступ к памяти |
| ПЗУ | — | постоянное запоминающее устройство |
| ПЭ | — | процессорный элемент |
| УВВ | — | устройства ввода-вывода |
| УУ | — | устройство управления |
| ФБ | — | функциональные блоки |
| ЦАП | — | цифро-аналоговый преобразователь |
| ЦВМ | — | цифровые вычислительные машины |
| ЦП | — | центральный процессор |
| ЦСП | — | цифровой сигнальный процессор |
| ЭВМ | — | электронная вычислительная машина |
| ЭВС | — | электронно-вычислительные средства |
| ADC | — | analog-to-digital converter (аналого-цифровой преобразователь) |
| AF | — | auxiliary carry flag (дополнительный флаг переноса) |
| AMD | — | advanced micro devices (американская компания) |
| ASIP | — | application-specific instruction-set processor (проблемно-ориентированный процессор) |
| AVX | — | advanced vector extensions (расширение системы команд процессора) |
| BIOS | — | basic input output system (базовая система ввода-вывода) |
| CF | — | carry flag (флаг переноса) |
| CISC | — | complex instruction set computing (компьютер с полным набором команд) |
| CNN | — | cable news network (кабельная новостная сеть) |
| CPU | — | central processing unit (центральный процессор) |
| DF | — | direction flag (флаг направления) |
| DRAM | — | dynamic random access memory (память динамического типа) |
| EDF | — | earliest deadline first (алгоритм планирования по ближайшему сроку завершения) |
| ENOB | — | effective number of bits (эффективная разрядность) |
| EPIC | — | explicitly parallel instruction computing (вычисление с явным параллелизмом машинных команд) |
| FIFO | — | first-in, first-out («первым пришел, первым вышел») |
| FPU | — | floating point unit (сопроцессор с плавающей точкой) |
| FSB | — | front side bus (системная шина) |
| GDT | — | global descriptor table (глобальная таблица дескрипторов) |
| GPU | — | graphics processing unit (графический процессор) |
| HDD | — | hard disk drive (жесткий диск) |
| HTT | — | hyper-threading technology (гиперпоточная технология) |
| IBM | — | international business machines (название американской компании) |
| IF | — | interrupt flag (флаг прерывания) |
| ILP | — | instruction-level parallelism (параллелизм на уровне команд) |
| IOPL | — | input/output privilege level (флаг уровня приоритета ввода-вывода) |
| IPS | — | inner product step (шаг в скалярном произведении) |
| ISA | — | industry standard architecture (архитектура промышленного стандарта) |
| LSTF | — | least slack time first («чем меньше слабое время, тем выше приоритет») |
| MESI | — | modified exclusive shared invalid (протокол) |
| MIMD | — | multiple instruction stream, multiple data stream (множественный поток команд) |
| MISD | — | multiple instruction stream, single data stream (множественный поток команд,одиночный поток данных) |
| MMX | — | multimedia extensions (мультимедийные расширения) |
| MSP | — | multi-streaming processor (многопотоковый процессор) |
| NT | — | nested task (флаг вложенности задач) |
| OC | — | операционная система |
| OF | — | overflow flag (флаг переполнения) |
| PCI | — | peripheral component interconnect (дословно: взаимосвязь периферийных компонентов) |
| PF | — | parity flag (флаг четности) |
| POST | — | power on self test (процесс самопроверки компьютера) |
| PSW | — | program status word (регистра слова состояния программы) |
| RAM | — | random access memory (оперативное запоминающее устройство) |
| RISC | — | restricted (reduced) instruction set computer (компьютер с сокращенным набором команд) |
| RMS | — | rate monotonic scheduling (алгоритм с фиксированным приоритетом) |
| ROM | — | read only memory (постоянное запоминающее устройство) |
| RTE | — | run-time environment (окружение времени выполнения) |
| RTOS | — | real-time operating system (операционные системы реального времени) |
| SATA | — | serial advanced technology attachment (последовательный интерфейс) |
| SDRAM | — | synchronous dynamic random access memory (синхронная динамическая память с произвольным доступом) |
| SF | — | sign flag (знаковый флаг) |
| SIMD | — | single instruction stream & multiple data stream (одиночный поток команд и множественный поток данных) |
| SISD | — | single instruction stream & single data stream (одиночный поток команд и одиночный поток данных) |
| SMT | — | simultaneous multithreading (одновременная многопоточность) |
| SRAM | — | static random access memory (память статического типа) |
| SSD | — | solid-state drive (твердотельный накопитель) |
| SSE | — | streaming simd extensions (потоковое simd-расширение процессора) |
| SSP | — | single-streaming processor (отдельный потоковый процессор) |
| TF | — | trap flag (флаг трассировки) |
| TR | — | task register (регистр задачи) |
| TSS | — | task state segment (сегмент состояния задачи) |
| USB | — | universal serial bus (универсальная последовательная шина) |
| VLIW | — | very long instruction word («очень длинная машинная команда») |
| ZF | — | zero flag (флаг нуля) |