1.Что такое компьютерная система? Отличие информационной и управляющей системы? Почему большинство современных компьютерных систем считаются системами с преобладающей программной составляющей? Примеры.

1. **Современная компьютерная система** - это гетерогенная система со сложным поведением, которое непредсказуемо без учета кибернетической части и логики работы компьютера.

   Основная часть современных систем - это компьютеры. Все они содержат свои вычислительные части, которые управляют "пассивными" компонентами и объединяют их в систему.

   Система - это суперпозиция многих проблем с разных точек зрения. Иногда можно сказать, что компьютер - это точка соединения системы, где разработчики должны соединить все компоненты в единый цельный объект - система, как тело, является куском мяса без нейронной системы и мозга.

   Обычно создание современной вычислительной системы требует разработки, бизнес-процессов (обучение персонала, контрольные точки, регламенты и т.д.) и аппаратного обеспечения (центры обработки данных, кластеры, компьютерные/контроллерные сети, контроллеры, заказной IP-код и процессоры и т.д.)

   Программная система (software system) - система, состоящая из программного обеспечения, аппаратных средств и данных, которая обеспечивает свою основную ценность за счет выполнения программного обеспечения.

2. Компьютерные системы можно условно разделить на две большие группы с точки зрения их назначения:
   - информационные компьютерные системы, основная задача которых получить набор данных на вход, преобразовать/накопить его, и выдать в изменено/обработанном виде;
   - управляющие компьютерные системы, основная задача которых взаимодействовать с реальным физическим миром с целью контроля или управления за ним.
     - имеют специальное аппаратное исполнение (набор портов ввода-вывода адаптирован под то окружение, в котором система будет работать; корпус компьютера адаптирован под место установки; процессора подобраны с учетом используемых алгоритмов и оптимизации энергопотребления; есть дополнительное экранирование и т.п.)
     - требование систем реального времени

3. Программное обеспечение позволяет нам применять стандарты программной инженерии для компьютерных систем и системной инженерии в целом (с небольшими модификациями). Одним из важнейших стандартов в этой области является OMG Essence, который определяет основы программной инженерии и общую точку зрения.

   Основная часть большинства сложных компьютерных систем в наши дни относится к классу систем с преобладающей программной составляющей.

   Системы с преобладающей программной составляющей (Software-Intensive system) - это системы, в которых разработка и/или интеграция программного обеспечения являются доминирующими факторами. Это связано с тем, что в первую очередь рассматривается то, на что и как тратятся ресурсы, и значительную часть стоимости таких систем составляет разработка программного обеспечения. Так как ключевым свойством компьютерной системы являются ее функциональные возможности.

   Примеры:

   - отдельные программные приложения
   - информационные системы
   - встроенные системы
   - линейки программных продуктов и семейств продуктов и систем

2. Понятие системы. Варианты рассмотрения систем. Модульность. Жизненный цикл. Операционное окружение и обеспечивающие системы. Идентичность системы. Заинтересованные стороны (stakeholders).

Концепция системы является одной из фундаментальных концепций в инженерном деле и науке. Она направлена на создание успешной системы.
Системная инженерия (SE) - это междисциплинарный подход и средство, позволяющее реализовать успешные системы:

- целостное и одновременное понимание потребностей заинтересованных сторон;
- изучение возможностей;
- документирование требований;
- синтез, проверка, валидация и эволюция решений при рассмотрении всей проблемы, начиная с изучения концепции системы и заканчивая ее утилизацией.

Система -

1) сочетание взаимодействующих элементов, организованных для достижения одной или более заявленных целей
2) элемент некоторой надсистемы и связанные с ними stakeholders

ПРИМЕЧАНИЕ 1 Система может рассматриваться как продукт или как услуги, которые она предоставляет.

ПРИМЕЧАНИЕ 2 На практике толкование его значения часто уточняется использованием ассоциативного существительного, например, авиационная система. Если говорить в общем то система - это конструкция из взаимодействующих компонентов(полупроводники, конденсаторы, резисторы, триггеры и т.д.), включающая в себя множество информационных компонентов(программы, данные)
Вспомогательная (обеспечивающая) система - система, которая дополняет интересующую систему на этапах ее жизненного цикла, но не обязательно вносит непосредственный вклад в ее функционирование во время эксплуатации.

Модульность - выбор гранулярности компонентов, уровня абстракции и вычислительных платформ, который имеет решающее значение при разработке компьютерной системы. То есть вся внутренняя организация системы состоит из взаимосвязанных модулей, которые в совокупности определяют целевую систему
Этот выбор должен соответствовать компьютерной системе и требованиям бизнеса. Например, если вы делаете веб-приложение - вы должны работать на веб-фреймворке с компонентами веб-страниц;

Жизненный цикл - эволюция во времени интересующей системы от замысла до вывода из эксплуатации.
Стадии ЖЦ:

1. **Этап концепции:** выполняется для оценки новых возможностей бизнеса и разработки предварительных требований к системе и осуществимого проектного решения. Для компьютерной системы это анализ требований и архитектурное проектирование.
2. **Этап разработки:** выполняется для создания интересующей системы, которая отвечает требованиям заказчика и может быть произведена, протестирована, оценена, эксплуатируется, поддерживается и выводится из эксплуатации. Для компьютерной системы это написание исходного кода или организация вычислительного процесса.
3. **Этап производства:** выполняется для производства или изготовления продукта, тестирования продукта и производства соответствующих вспомогательных и обеспечивающих систем по мере необходимости. Для компьютерной системы это разработка аппаратного обеспечения, подготовка исполняемых артефактов и распространение.
4. **Этап использования:** выполняется для эксплуатации продукта, предоставления услуг в намеченных условиях и обеспечения постоянной операционной эффективности.
5. **Этап поддержки:** выполняется для предоставления услуг логистики, технического обслуживания и поддержки, которые обеспечивают непрерывную работу интересующей системы и устойчивое обслуживание.
6. **Этап вывода из эксплуатации:** выполняется для обеспечения удаления интересующей системы и соответствующих эксплуатационных и вспомогательных услуг, а также для эксплуатации и поддержки самой выводимой из эксплуатации системы.

Операционная система
Функциональные возможности системы напрямую зависят от **операционной среды** - среды, в которой люди, другие системы или физические объекты взаимодействуют с

интересующей нас системой. Т.е. ОС определяет конструкцию интересующей нас системы и функциональное место(над систему), в которое наша система встраивается и работает
Для компьютерной системы операционная среда важнее, чем внутренняя организация. Если мы сохраняем интерфейс в операционной среде, обычно мы можем полностью изменить внутреннюю организацию, и всем наплевать. Например, посмотрите на современные мейнфреймы, которые выполняют исходный код семидесятых годов. Что предполагает перфокарты, но работает со скоростным твердотельным накопителем.

Stakeholder - физическое и юридическое лицо, которое может накладывать ограничения на вашу систему т.е. например, пользователь, который определяет дизайн, разработчик, определяющий какие именно технологии будут использоваться, директор, заинтересованный в актуальности и спросе на рынке
Идентичность системы
Короче, в душе не ебу че хочет от нас ПЕНСКОЙ(ПИДАРАС)
Я приведу пример с насосной станцией, чтобы саму концепцию описать
Представим себе насосную станцию, в которой стоит два насоса. Каждый насос имеет собственный агрегат: у первого серийный номер 1.1.1 и у второго 1.2.1. Первый насос ломается и мы заменяем в нем агрегат с другим номером 1.1.2. В результате, мы получаем, что насосная станция и насосы в ней не изменились, это все те же конструкции, но уже используется другой агрегат с другим номером 1.1.2. В этом видимо и заключается идентичность системы: то есть с внешней стороны станция не изменилась, так как она функционирует так же, как и раньше, но если посмотреть внутрь,, то там стоит теперь другой агрегат и это абсолютно другая по составу система.

3.Цели и задачи архитектурного проектирования компьютерных систем. Понятие архитектуры. Различные трактовки и их практическая значимость.

1. Прежде всего, нам нужно определить роль концепции архитектуры. Для этого мы начнем со следующего определения (мы не разделяем компьютерные и программные системы.
   Архитектура - все важное (определение из интернета)
   Архитектура - это набор проектных решений, которые, если они будут приняты неправильно, могут привести к краху проекта. (определение Эойна Вудса (SEI 2010))

   Эти определения говорят о том, что архитектура должна быть выполнена на ранней стадии и выполнена высококвалифицированными специалистами. Она должна принимать все важнейшие решения для создания успешного проекта. Но с помощью этих определений мы можем анализировать то, что является частью архитектуры, только в ретроспективе, когда проект был успешным или неудачным. :(

   Более подробное определение архитектуры можно взять из стандарта ISO 42010.

Архитектура (системы) - фундаментальные концепции или свойства системы в ее среде, воплощенные в ее элементах, взаимосвязях и принципах ее проектирования и эволюции.
Описание архитектуры - рабочий продукт, используемый для выражения архитектуры.

И последнее, классическое определение архитектуры программного обеспечения с точки зрения разработки программного обеспечения.
Архитектура - логическая и физическая структура компонентов системы и их взаимосвязей, сформированная всеми стратегическими и тактическими проектными решениями, применяемыми в процессе разработки.

Логическая структура содержит в себе концепции, созданные в концептуальной модели, и устанавливает существование и значение ключевых абстракций и механизмов, которые будут определять архитектуру и общий дизайн системы.
Физическая структура описывает конкретный программный и аппаратный состав реализации системы. Очевидно, что физическая модель зависит от конкретной технологии.

Данное определение подразумевает превращение компьютерной системы в логический и физический компонент. Логический компонент - это та часть системы, которая необходима для ее применения. Физический компонент - это обеспечивающая часть вычислительной системы, которая позволяет системе функционировать.
Это определение немного устарело для современных реконфигурируемых и облачных систем, где аппаратное обеспечение может динамически изменяться в зависимости от текущих требований.
Ключевыми особенностями архитектуры системы являются ее многочисленные представления. Например, UML, который включает в себя множество типов диаграмм для описания системы для различных целей.
Но на самом деле нам нужно работать с самой архитектурой, а не только в форме спецификаций или документов. Мы также используем ее в схемах на стадии идеи и в отслеживании багов, когда нужно поставить задачу как часть инженерной культуры. Следовательно, мы создаем множество архитектурных описаний "на лету" в разных контекстах и группах людей, что может вызвать множество вопросов.

2. Использование архитектуры в разработке программного обеспечения и систем приводит к следующим последствиям:

а) Аппаратное и программное обеспечение являются компонентами компьютерной системы. Это означает, что, анализируя компьютерную систему,

вы не можете абстрагироваться ни от одной из них. И нужно создать механизм связи между разработчиками программного и аппаратного обеспечения.

б) Понятие “архитектура” включает в себя технические вопросы и вопросы, связанные с проектом. Невозможно спроектировать архитектуру вычислительных систем только с технической точки зрения. Следующие вопросы также являются частью архитектуры:

- команда и компетентность, например, используемый стек технологий, если по нему нет специалистов, то это значительно повышает риски при создании проекта;
- время выхода на рынок: если внедрение системы займет несколько лет, существует высокая вероятность того, что по истечении этих сроков она станет бесполезной;
- обслуживание, поддержка, развертывание и обновление системы;
- и много другого.

в) Система обладает архитектурой независимо от ее описания. Допустим, необходимо добавить новые функциональные возможности в существующую систему без архитектурной документации. Существует два способа выполнения этих задач:

- Добавить новую функцию в систему методом “brutal force”. Обычно это можно быстро сделать за счет обеспечения согласованности системы, но если это долговечный продукт, это значительно увеличит затраты на техническое обслуживание;

.

- Проанализируйте архитектуру системы (если возможно) и интегрируйте новую функцию. Такой подход позволяет элегантно расширить существующую вычислительную систему, не внося в нее концептуальных несоответствий.

4.Реле как базис компьютерной системы. Область применений и принципы построения систем на базе реле. Примеры релейных схем.

Реле́ (фр. *relais*) — коммутационный аппарат, который при воздействии на него внешних физических явлений скачкообразно принимает конечное число значений выходной величины[1].

Назначение реле заключается в автоматизации замыкания или размыкания электрической цепи.

1. В основе вычислительной платформы “релейных схем” лежит относительно простое устройство – реле. Рассмотрим простое электрическое реле со следующим устройством:

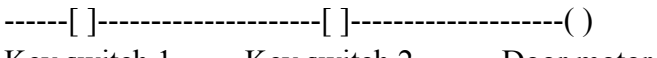
• вход и выход, между ними металлический ключ (нормально разомкнутый или нормально замкнутый);

• магнитная катушка, при подаче тока на которую ключ автоматически замыкается или размыкается, при снятии тока – ключ возвращается в нормальное состояние автоматически.

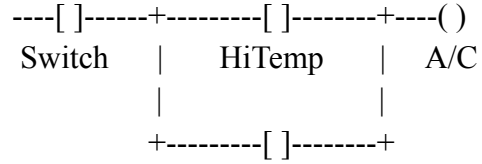
2. С точки зрения современного положения дел:
   • релейные схемы управления как применялись для решения практических задач, так и продолжают применяться;

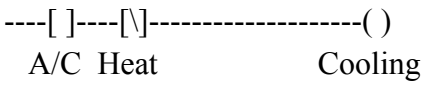
   • реле бывают не только электрические, но и механические (включая пневматические), тепловые, оптические, акустические и магнитные, что позволяет применять их в тех областях, в которых электрические компоненты по тем или иным причинам работать не могут;

   • современные программируемые логические контроллеры (современный высокотехнологичный инструмент для автоматизации производственных процессов) в качестве одного из инструментов программирования поддерживают язык релейных диаграмм
3. Несколько примеров [Wikipedia] простых релейных схем, выполненных в нотации языка IEC 61131.
   Синтаксис и описание в текстовом виде: Ladder logic Syntax and examples



Логическое “И”.

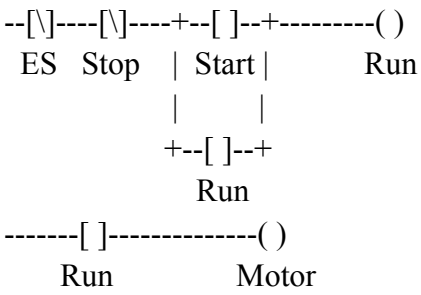
1. 

2. 

Управление холодильником, логическое “И” и “ИЛИ”.

1. Реализует функцию: A/C = Switch AND (HiTemp OR Humid).
2. Реализует функцию: Охлаждение = A/C И (НЕ Тепло).

Это немного более сложная система для ступеной 2. После оценки первой строки выходная катушка «A/C» подается в ступень 2, которая затем оценивается, а выходная катушка «Охлаждание» может подаваться в выходное устройство «Компрессор» или в ступень 3 на лестнице. Эта система позволяет разбивать и оценивать очень сложные логические проекты.



Вышеприведенное реализует функцию: Run = (Start OR Run) AND (NOT Stop)

```
--[\]----[\]----+--[ ]--+---------( )
  ES    Stop    | Start |        Run
                |       |
                +--[ ]--+
                   Run
```

```
-------[ ]--------------( )
       Run             Motor
```

По соображениям безопасности аварийная остановка («ES») может быть жестко подключена последовательно с переключателем «Пуск»

Вышеприведенное реализует функцию: Run = (НЕ ES) AND (NOT Stop) AND (Start OR Run)

5.Что такое комбинационная схема? Состояние и параллелизм в комбинационных схемах и схемах с регистрами. Особенности поведения систем на базе комбинационных схем от программных систем.

1. **Комбинационная схема** – схема составленная из набора логических элементов, в совокупности реализующая заданную таблицу истинности.

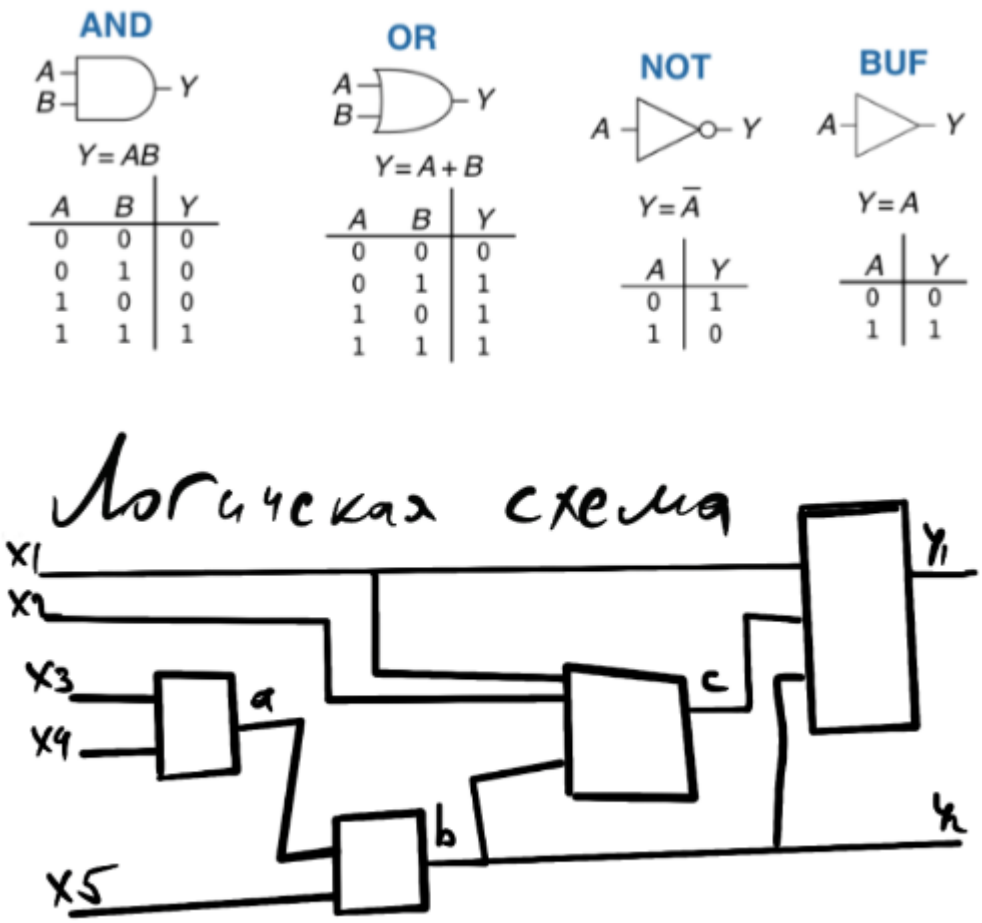


Figure 13: Логическая схема

Логические операции позволяют реализовать произвольную математическую функцию (см. изображение), причем это может быть реализовано для:

- черного ящика, когда мы реализуем таблицу истинности;
- белого ящика, когда мы понимаем особенности входных / выходных данных и их взаимосвязей, а значит на основании этого понимания творчески оптимизировать комбинационную схему или “раскладывать” ее на многошаговый процесс.

Основные свойства любой комбинационной схемы:

- возможность установления стабильного состояния при корректном входе;
- задержка установления стабильного состояния после изменения входных значений (зависит от условий окружающей среды);
- параллельная работа элементов комбинационной схемы;

• накопление ошибки в физическом процессе, что может привести к ошибке на логическом уровне.

Последняя проблема решается при помощи буфера, который “выравнивает” аналоговый сигнал лежащий в основе логического.

2. **Особенности “последовательного выполнения”**

При помощи триггеров, у нас есть возможность реализовать “защелкивание” состояния в схеме.

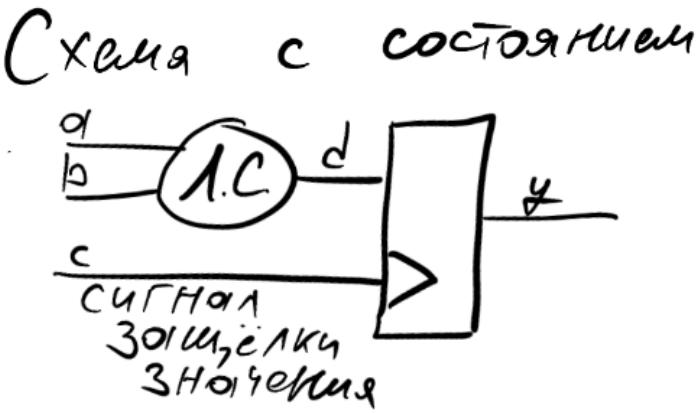


Figure 15: Логическая схема с состоянием

Вход, отмеченный треугольником – управление защелкой состояния. Защелкивание может происходить при разных событиях управляющего сигнала: изменение с 0 на 1 (положительный фронт), с 1 на 0 (отрицательный фронт), по состоянию (пока сигнал единичный, входное значения защелкивается).

Это позволяет следующее:

• разбить большую комбинационную схему на несколько (использование вместо буфера);

• хранить состояние внутри схемы, а значит производить вычислительный процесс в несколько шагов (реализация счетчика, сокращение размера схемы за счет нескольких этапов вычислений).

Важно понимать, что процессы во всех комбинационных схемах происходят параллельно, а значит у нас открываются возможности для синхронных цифровых схем, в которых “защелкивание” значений в триггерах происходит одновременно.

Примечания:

• одновременность весьма условна;

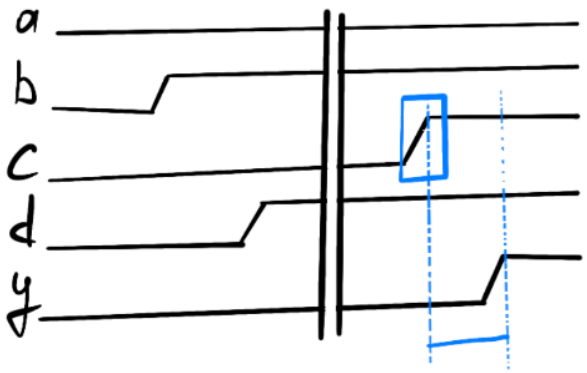


Figure 16: Логическая схема с состоянием, временная диаграмма

• синхронный сигнал должен ориентироваться на самую долгую комбинационную схему, в противном случае схема работать не сможет;

• работа с синхронной схемотехникой приводит к дискретизации аналоговых сигналов, что также вносит погрешность в работу систем (особенно систем управления).

Синхронные схемы позволяют нам делать следующее:

• Конвейеризация вычислений (количество стадий конвейера равно количеству обрабатываемых в такт значений);

• Управлять тактовой частотой схемы, так как мы всегда можем разделить комбинационную схему триггером на часть до и после, при этом если

1) комбинационные схемы делятся ровно пополам, то у нас нет избыточных задержек;

2) триггер имеет свою длительность срабатывания.

Удвоение частоты в данном случае едва увеличит производительность в два раза.

Особенности “условного оператора”

Ветвление в булевых компьютерах реализовано в два подхода:

• через состояние (по сути реализован в современных процессорах, когда состояние регистров определяет следующий шаг вычислительного процесса);

• через спекулятивные вычисления и выбор результата (мультиплексор).

Основы работы логических компьютеров:

• все процессы между регистрами всегда происходят параллельно, последовательность обработки данных – логическая конструкция

которая отсутствует внутри, хотите понять как работает – рисуйте схему;

• нет понятия “система остановилась”, она всегда продолжит функционировать в том или ином виде либо будет выключена / перезагружена внешней системой (по крайней мере для синхронной схемотехники);

• передача сигнала – физический аналоговый процесс, а значит он не может прекратиться, если мы не определили какой сигнал пойдет дальше, это значит что дальше пойдет случайный сигнал.

6.Понятия Hardware и Software, их свойства. Сравнение с понятиями программного и аппаратного обеспечения. Особенности. Причины разделения.

1. Традиционно понятия Hardware и Software переводятся как аппаратное и программное обеспечение:

   Аппаратное обеспечение : электронные и механические части вычислительного устройства, входящие в состав системы или сети, исключая программное обеспечение и данные (информацию, которую вычислительная система хранит и обрабатывает). Аппаратное обеспечение включает: компьютеры и логические устройства, внешние устройства и диагностическую аппаратуру, энергетическое оборудование, батареи и аккумуляторы.

   Программное обеспечение : совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ.

   Компьютерная программа : комбинация компьютерных инструкций и данных, позволяющая аппаратному обеспечению вычислительной системы выполнять вычисления или функции управления.

   Данные определения даются с точки зрения внутреннего устройства / конструкции / содержания рассматриваемых объектов. Как отмечалось ранее, система определяется не столько своим наполнением, сколько своей функцией или ролью в над-системе. Когда мы говорим о HW и SW, в большинстве случаев мы говорим о том что мы можем изменить, что не можем изменить, а что можем, но долго и дорого. В центр внимания попадает жизненный цикл нашей компьютерной системы, в котором мы можем выделить следующие этапы:

   - производство;
   - сборка и комплектация -- современные компьютерные системы как правило состоят из большого количества компонент объединенных стандартными интерфейсами, позволяющих получать, сформировать компьютерную систему с разными возможностями:
     - макетные платы;

        - платы расширения:
            - расширение портов ввода/вывода, включая обработку данных;
            - предоставления специализированных вычислителей под конкретные задачи;
            - фиксация алгоритмов на уровне аппаратуры;
    - [ре-]конфигурирование -- настройка функционирования аппаратных средств, управление режимом работы аппаратных средств;
        - джамперы, перемычки, дип-переключатели;
        - конфигурация аппаратных узлов для реализации требуемой функциональности;
        - конфигурирование данными;
    - программирование -- определение компьютерной программы, определяющей реализуемую компьютерной системой функциональность;
    - настройка / пользовательское программирование -- определение настроек программных продуктов.

Видно, что данные этапы выделены весьма спорно и не могут претендовать ни на четкость проведения границ между ними, ни на полноту, ни на корректность с точки зрения последовательности в жизненном цикле.

Особенности аппаратной составляющей

- У аппаратной части компьютерной системы есть свой срок службы по истечению которого она лишается гарантии, риск внезапного выхода их строя резко возрастает, она требует обслуживания.
- Замена аппаратной составляющей затруднена: сложностью и дороговизной долгого хранения деталей для ремонта или замены.
- Воспроизводство аппаратной составляющей сопряжено с устареванием элементной базы. Устаревшая элементная база вытесняется из производства новой, как следствие она довольно быстро переходит в разряд штучного, который:
    - сперва производится несколькими компаниями за очень большие деньги;
    - потом не производится совсем (периодически можно видеть как крупные компании ищут то или иное оборудование по барахолкам, есть и компании которые на этом специализируются).
- уходит сопутствующее оборудование
- изменение аппаратной составляющей подразумевает физический контакт с устройством, что не всегда физически возможно и не всегда экономически целесообразно

Изменение ПО:

- надо применить патч к программной документации
- повторно собрать проект

- обновить программное обеспечение на компьютерах и перезапустить
- легкость изменения ПО является одной из причин огромного количества уязвимостей компьютерных систем.

Подводя итог, зафиксируем:

- Hardware - то что тяжело/долго/дорого поменять;
- Software - то что легко/быстро/дешево поменять.

Данное разделение зависит от точки зрения, а если точнее -- то от свойств обеспечивающей системы.

7 билет кста полностью неправильный

7.Состав программной системы в соответствии с OMG (Object management group) Essence. Роль данных и аппаратного обеспечения.

1. В основе представления чисел лежит представление 0 и 1 через разные логические уровни.

   Почему выбор пал именно на двоичное представление сигналов:

   - Представление информации посредством только двух состояний надежно и помехоустойчиво.
   - Двоичная арифметика проще десятичной с точки зрения реализации.
   - Диапазоны и точность представлений чисел могут наращиваться за счет разрядности.
   - Погрешности “by design”, а не “by implementation” (одинаковые исправные компьютеры считают одинаково).

   Недостатки:

   - Нечеловеческое представление (“а и не надо”, бинарные часы).
   - Простые десятичные числа записываются в виде бесконечных двоичных дробей. На самом деле, работа с бинарной логикой на практике часто требует работы с тремя, а то и четырьмя состояниями (использован синтаксис языка описания аппаратуры Verilog)
     - очевидные 0 и 1;
     - z – отключено, когда ваш источник данных (провод) висит в воздухе, а не хранит значения логического 0 или 1 (отсюда часто встречаемое на практике кодирование бинарного состояния двумя положительными физическими уровнями);
     - x – неизвестно, когда с точки зрения реализации значение могут быть произвольными (например выход данных делителя после деления на 0)

или захвачено некорректное значение (ошибка синхронизации между защелками).

Аппаратное обеспечение - электронные и механические части вычислительного устройства, входящие в состав системы или сети, исключая программное обеспечение и **данные** (информацию, которую вычислительная система хранит и обрабатывает). Аппаратное обеспечение включает: компьютеры и логические устройства, внешние устройства и диагностическую аппаратуру, энергетическое оборудование, батареи и аккумуляторы.

Любая компьютерная система, хотим мы того или нет, имеет в своем составе аппаратную составляющую. Даже определение системы с преобладающей программной составляющей выглядит следующим образом:

Система с преобладающей программной составляющей - система, состоящая из программного обеспечения, аппаратного обеспечения и данных, которая обеспечивает свою основную ценность за счет выполнения программного обеспечения (OMG Essence).

С этим связано множество трудностей, рассмотрим некоторые их них:
Старение аппаратной составляющей

- У аппаратной части компьютерной системы есть свой **срок службы** по истечению которого она лишается гарантии, риск внезапного выхода их строя резко возрастает, она требует обслуживания.

- **Замена** аппаратной составляющей **затруднена**: сложностью и дороговизной долгого хранения деталей для ремонта или замены.

- Воспроизводство аппаратной составляющей сопряжено с **устареванием элементной базы**. Устаревшая элементная база вытесняется из производства новой, как следствие она довольно быстро переходит в разряд штучного, который:
  – сперва производится несколькими компаниями за очень большие деньги;
  – потом не производится совсем (периодически можно видеть как крупные компании ищут то или иное оборудование по барахолкам, есть и компании которые на этом специализируются).

- Вслед за элементной базой из оборота **уходит сопутствующее оборудование** (кто бы мог подумать, что в компьютерах не будет CD-ROM дисков, а производство дискет 2.5 дюйма будет остановлено).

В результате, современные компьютерные системы с аппаратной точки зрения вынуждены претерпевать постоянное перепроектирование просто для того, чтобы быть тиражируемыми. Другие варианты:

- **разделение системы** на части по стандартным интерфейсам (аппаратным, информационным или программным), относительно

которых модернизация может происходить условно независимо, пока не нарушаются контракты (стандартизация);
- **виртуализация**, на сегодня продолжает существовать такой класс компьютерных систем как “Мэйнфрэймы”, многие из которых по прежнему уверены что работают с магнитными лентами и перфокартами.

Обратная сторона. Программное обеспечение может жить вечно, так как “команды и коды не стареют”, чего к сожалению нельзя сказать о командах разработчиков. Как следствие – держатель компьютерной системы рано или поздно в любом случае будет терять над ней контроль.

8.Закон Мура и закон Деннарда. Их роль в развитии компьютерной техники.

1. **Закон Мура:** количество транзисторов, размещаемых на кристалле интегральной схемы, удваивается каждые 24 месяца.
   К сожалению, он сдерживается следующими факторами:

   • закон Амдала. Данный закон характеризует рост производительности системы относительного того, какой процент задач может быть выполнен параллельно и количества доступных вычислителей (см. картинку);
   • объективная сложность параллельного программирования (современное многопоточное программирование может быть реализовано средним современным программистом только за счет большого количества абстракций, позволяющих просто работать с простыми случаями разработки);
   • доставка данных, так как для работы необходимо не только получить входные данные, но также и выдать полученные результаты (возможно и промежуточные);
   • тепловыделение и питание;
   • ограничения связанные с физическими процессами (к примеру, когда не остается атомов для того, чтобы сложить затвор транзистора).

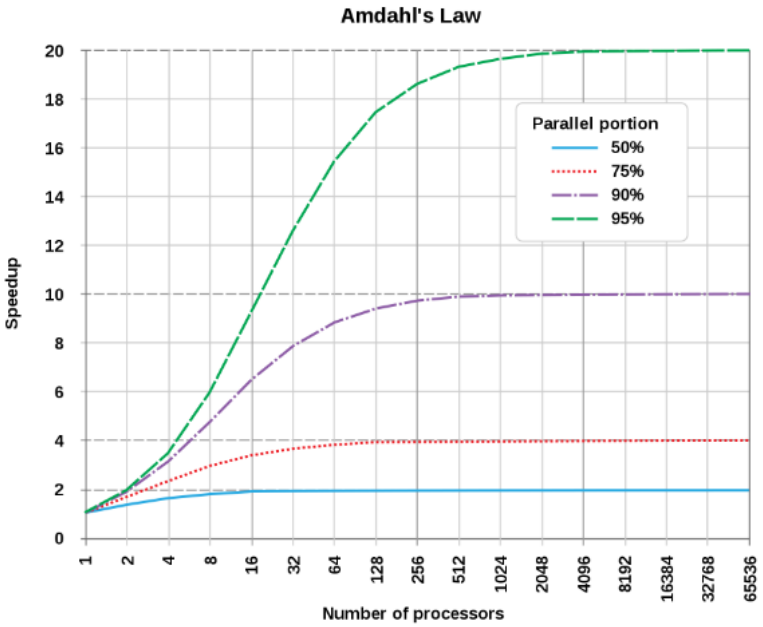


Figure 18: Amdahl's law

Последние 50 лет направление и темп развития электроники определялись законом Мура. С помощью этой гипотезы ученые стремятся вычислить темп развития ОП компьютеров, мощность и вычислительную скорость устройств.

Стоит выделить 3 типа влияния закона Мура на мир:

- соперничество разработчиков, т.е. борьба за первенство в создании более быстрого и более мощного процессора;
- разработка архитектуры вычислительных мощностей: обновление технологических алгоритмов происходит постоянно и регулярно (раз в 2 года);
- прогнозирование рынка.

Начиная с 2010 года, следование закону Мура перестало быть выгодным для разработчиков. На его соблюдение нужно тратить множество ресурсов: материалы, оборудование, увеличение штата и проч. На 2019 год закон Мура не работает эффективно, эра кремниевых транзисторов завершится предположительно до 2030 года.

2. Повышение скорости вычислений может достигаться за счет ускорения исполнения операций / преобразований данных, которое в свою очередь может достигаться за счёт **закона масштабирования Деннарда** – уменьшая размеры транзистора и повышая тактовую частоту процессора, мы можем легко повышать его производительность.
   К сожалению, как и закон Мура, данный закон имеет свою область применимости и срок годности, так как наталкивается на:
   • Ограничения современных технологий (сложность и стоимость производства).
   • Физические ограничения на возможности уменьшения транзисторов (размеры атомов и эффекты связанные с уменьшением транзисторов).

   Правило Деннарда закрепило уменьшение ширины проводника (техпроцесса) в качестве главного показателя прогресса в индустрии микропроцессорной техники. Но закон масштабирования Деннарда перестал действовать примерно в 2006 году. Количество транзисторов в чипах продолжает увеличиваться, но этот факт **не дает значительного прироста** к производительности устройств.

   Например, представители TSMC (производитель полупроводников) говорят, что переход с 7-нм техпроцесса на 5-нм **увеличит** тактовую частоту процессора всего на 15%.

   Причиной замедления роста частоты, являются утечки токов, которые Деннард не учитывал в конце 70-х. При уменьшении размеров транзистора и повышении частоты ток начинает сильнее нагревать микросхему, что может вывести ее из строя. Поэтому производителям приходится балансировать выделяемую процессором мощность. В результате с 2006 года частота массовых чипов установилась на отметке в 4–5 ГГц.

9.Понятие модели вычислений. Примеры моделей вычислений и их роль в разработке компьютерных систем. Модель-ориентированная инженерия.

1. Вычислительный процесс представлен определенным типом модели/описания/спецификации. Каждый из них описан отдельно. Любая модель может быть связана с определенным набором семантических и вычислительных правил, что позволяет нам выполнять модели с предсказуемыми (но не обязательно детерминированными) результатами. Этот набор правил лежит в основе описаний моделей, как архитектурные стили лежат в основе архитектуры.

   Модель вычислений (дальше МВ) - это модель, которая описывает, как вычисляется результат математической функции с учетом входных данных. Модель описывает, как организованы единицы вычислений, памяти и связи. Вычислительная сложность алгоритма может быть измерена с учетом модели вычислений. Использование модели позволяет изучать производительность

алгоритмов независимо от вариаций, характерных для конкретных реализаций и конкретной технологии.

2. Модель вычислений имеет много знакомых понятий:
    - Парадигмы программирования - это способ классификации языков программирования на основе их особенностей. Языки можно разделить на несколько парадигм.
    - Стиль программирования, также известный как стиль кода, представляет собой набор правил или руководящих принципов, используемых при написании исходного кода компьютерной программы. Часто утверждается, что следование определенному стилю программирования поможет программистам прочитать и понять исходный код, соответствующий стилю, и поможет избежать ошибок и неправильных предположений.
    - Языки программирования.
    - Архитектурный стиль.
    - Вычислительная платформа.

Различия между их целями могут подчеркнуть особенности этих концепций. Парадигмы программирования и стили программирования нацелены на то, чтобы быть инструментом разработчика (гибким, практичным, полезным). Архитектурный стиль – инструмент системного дизайнера (гибкость, универсальность, читабельность). Вычислительная платформа – объект многократного использования и точка опоры для разработчиков (универсальность, полезность, ремонтопригодность). Напротив, концепция МВ была разработана для информатики, теории совместимости и программирования с формальными свойствами. Следовательно, МВ обычно имеет прочную математическую основу (что мы можем не говоря уже о многих инструментах программиста) и описывает очень узкий класс вычислительных процессов. Например, большинство современных языков программирования являются “многопарадигмальными”, но типичный МВ - это минималистичная и ограниченная модель, без каких-либо ненужных функций или возможностей. Более того, любая дополнительная функция немедленно вызывает проблемы с возрастающей сложностью.
Модели вычислений можно разделить на три категории:

- Последовательные модели, позволяющие описать последовательный процесс, который может быть представлен в виде последовательности переходов состояний.:
    - Конечные автоматы
    - Нажимные автоматы
    - Машины произвольного доступа
    - Машины Тьюринга.

• Функциональные модели, которые представляют вычислительный процесс в символьной форме и набор правил сокращения:

– Лямбда-исчисление

– Общие рекурсивные функции

– Комбинаторная логика

– Системы переписывания абстрактных текстов.

• Параллельные модели, которые применяются для системы, процесс которой включает в себя несколько взаимодействующих процессов и фокусируется на этих взаимодействиях (обычно поведение внутреннего процесса описывается в разных МВ):

– Клеточный автомат

– Технологические сети Кана

– Сети Петри

– Синхронный Поток Данных

– Сети взаимодействия

– Модель-актер.

Модель вычислений играет определяющую роль в том, какие вычислительные процессы могут исполняться при помощи доступной нам компьютерной системы.

В тоже время она определяет и то, в каких ограничениях должен быть реализован вычислитель. Нарушение ограничений модели вычислений (как в чистом виде, так и как следствия их смешивания) приводит к радикальному росту сложности системы и появлению “дыр” в безопасности или предсказуемости.

3. **Модельно-ориентированная инженерия** (**MDE**) - это методология разработки программного обеспечения, которая фокусируется на создании и использовании моделей предметной области, которые являются концептуальными моделями всех тем, связанных с конкретной проблемой. Таким образом, это выдвигает на первый план и цели на абстрактных представлениях о знании, и деятельности , которые регулируют определенную область применения , а не вычисление (т.е. алгоритмических) понятий.

10.Понятие информационного процессора. Машина Тьюринга. Свойства универсального компьютера.

1. **Информационный процессор (система обработки информации)** - система (будь то электрическая, механическая или биологическая), которая принимает информацию (последовательность перечисленных символов или состояний) в

одной форме и преобразует ее в другую форму, например к статистике через алгоритмический процесс.
Системы обработки информации состоит из четырех основных частей, или подсистем:

- ввод
- процессор
- хранение
- вывод

Процессор - цифровая схема, которая выполняет операции с некоторым внешним источником данных, обычно памятью или каким-либо другим потоком данных. Обычно он принимает форму микропроцессора, который может быть реализован на одном кристалле металл-оксид-полупроводниковой интегральной схемы.

2. Машина Тьюринга содержит следующие элементы:

   • неограниченная двусторонняя лента (возможны машины Тьюринга с несколькими бесконечными лентами), разделенная на ячейки
   • управляющее устройство (также называемое головкой записи-чтения), которое может находиться в конечном числе состояний.

   Управляющее устройство может перемещаться влево и вправо по ленте, считывать и записывать символы некоторого конечного алфавита в ячейки. Выделяется специальный пустой символ, который заполняет все ячейки ленты, кроме ячеек с входными данными (конечное число).

   Управляющее устройство работает в соответствии с правилами перехода, представляющими алгоритм машины Тьюринга. Каждое правило перехода предписывает машине, в зависимости от текущего состояния и символа наблюдаемого в текущей ячейке, записать новый символ в эту ячейку, переключиться в новое состояние, переместиться на одну ячейку влево или вправо. Некоторые состояния машины Тьюринга могут быть помечены как терминальные. Переход к ним означает, что алгоритм заканчивается.

   Существенной частью машины Тьюринга является четкое разделение потока управления (управляющего устройства) и потока данных (ленты). Это свойство будет передано от гипотетического компьютера к наиболее известному семейству процессоров – процессору фон Неймана.

3. В основе программного обеспечения лежит одна простая идея: создание универсального компьютера или универсальной машины, которая будет решать именно те задачи, которые являются актуальными для пользователя прямо сейчас, по щелчку. Когда адаптация компьютера к новой задаче реализуется тривиально или, в случае если все подготовлено, происходит на

лету и автоматически. Как следствие, модель вычислений должна обладать следующими свойствами:

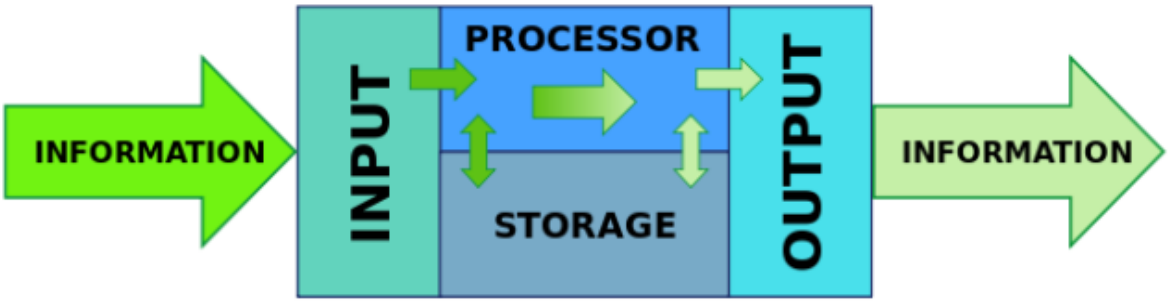
• полнота по Тьюрингу (выразимость алгоритмов) - характеристика исполнителя в теории вычислимости, означающая возможность реализовать на нём любую вычислимую функцию.
• отсутствие “серьезных” ограничений на “объем” программы;
• техническая возможность простого изменения ПО, в теории, должна быть поддержана со стороны модели вычислений (например, если модель вычислений подразумевает полное переписывание исходного, то какой тогда смысл в легкой).

Примечания:
• Универсальность всегда противоречит вопросам эффективности (производительность, энергопотребление);
• Современное Hardware содержит огромное количество программного обеспечения;
• Процессор обязан быть аппаратным?

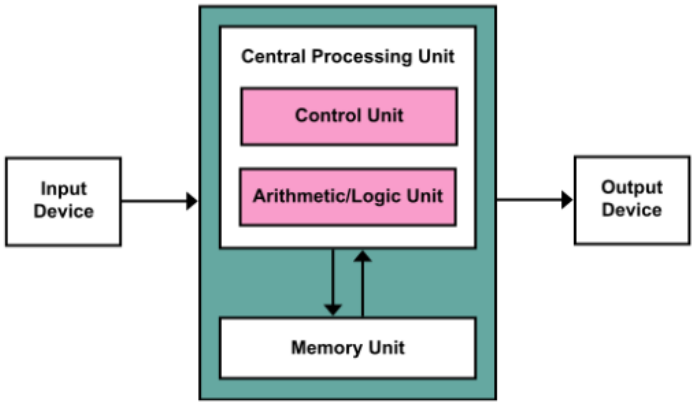
Отсюда появляется понятие “процессора с программным управлением”, устройства:

• которое может “выполнить” программное обеспечение;
• которое может использовать данные для разворачивания вычислительного процесса во времени;
• которое демонстрирует дуализм процесса и механизмов, реализующих его.

Проиллюстрируем с точки зрения внутренней организации:



11.Архитектура фон Неймана. Принципы. Свойства. Особенности и ограничения. Применение на практике.



1. Изображение показывает архитектуру фон Неймана. Его можно рассматривать как логическую эволюцию машины Тьюринга. Память с произвольным доступом заменяет ленту. Устройство управления, по сути, осталось прежним, но оно преобразовало систему команд в более привычную для нас форму.

2. Данная архитектура основана на следующих принципах:

   • Использование двоичного кодирования. Преимущество двоичной системы состоит в том, что устройства могут быть сделаны довольно простыми, а арифметические и логические операции в двоичной системе также довольно просты. (Имеет место быть троичное кодирование и двоично-десятичное кодирование);

   • Управление программным обеспечением. Программа – последовательность команд, управляющих работой компьютера. Команды выполняются последовательно, одна за другой. Создание машины с программой, хранящейся в памяти, было началом того, что мы сегодня называем программированием. (Наиболее условным является пункт о последовательном исполнении команд в современных процессорах);

   • Память компьютера используется для хранения данных и программ. В этом случае как команды программы, так и данные кодируются в двоичном формате. Поэтому в некоторых случаях вы можете выполнять с командами те же действия, что и с данными. (Однородность данных – условна. Информация о характере работы с данными позволяет оптимизировать работу с ними);

   • Ячейки памяти компьютера имеют пронумерованные адреса. В любое время вы можете получить доступ к любой ячейке памяти по ее адресу. Этот принцип позволил использовать переменные в программировании.

– Понятие адреса не сводится к целому числу в реальных процессорах.
– Память в современных процессорах не является пассивным элементом в полном смысле это слова;

• Возможность условного ответвления во время выполнения программы. Команды выполняются последовательно, но программы могут реализовать возможность перехода к любой части кода. (Современные компьютеры этим не ограничиваются).

3. Сегодня чистая архитектура фон Неймана не используется на практике. Но его влияние на отрасль очень трудно переоценить. Его элементы присутствуют в подавляющем большинстве процессоров и средств разработки.
4. **Система команд процессора** – абстрактная модель процессора, формирующая интерфейс взаимодействия между программным обеспечением и процессором, затрагивающая:
   • типы данных;
   • систему регистров;
   • методы адресации;
   • модели памяти;
   • инструкции;
   • способы обработки прерываний и исключений;
   • методов ввода и вывода.

   Система команд процессора описывает поведенческий аспект работы процессора и напрямую не касается вопросов производительности, энергопотребления и временных задержек. Современные требования по информационной безопасности Meltdown and Spectre и применения в области встроенных и мобильных систем требует пересмотра данного подхода.

   Виды инструкций
   • Работа с памятью
   – Запись константных значений в регистры.
   – Копирование данных между памятью и внутренними регистрами процессора.
   – Чтение и запись данных во внешние устройства.
   • Арифметические и логические операции.
   – Сложение, вычитание, умножение и деление с сохранением результатов в регистр процессора. Инкремент и декремент значений.
   – Побитовые И, ИЛИ, НЕ, сдвиги, выборка/установка значений указанных битов.
   – Сравнение значений в регистрах.
   – Операции с плавающей точкой.
   • Управляющие операции

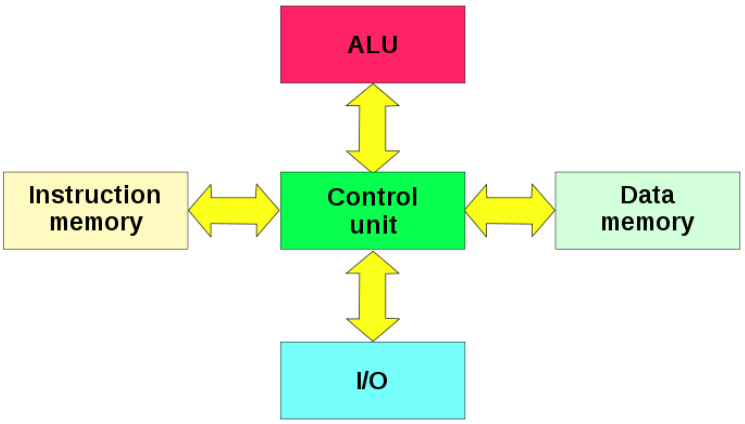
– Безусловный и условных переход.
– Косвенный переход.
– Вызов и возврат из подпрограмм.

• Инструкции для сопроцессоров
– Загрузка данных и получение результатов.
– Управление операциями сопроцессора

Вычислительный процесс построен как последовательное выполнение команд, которые изменяют память процессора.

12.Гарвардская архитектура. Принципы. Свойства. Особенности и ограничения. Применение на практике.

1. **Гарвардская архитектура** – архитектура в которой память команд и память данных физически разделены, имеют собственные наборы шин для взаимодействия. А значит:



- Одновременный доступ к памяти (многопортовая память является редкостью).
- Разные длины машинного слова и адреса для данных и программ.
    - Оптимизация под решаемую задачу.
    - Данные и память программ всегда перемешаны (например, непосредственная адресация и указатели на функции).
- Два физических канала между процессором и памятью.

Модифицированная гарвардская архитектура. Доступ к памяти реализуется через независимые кеши для данных и программ, за счет чего, с точки зрения внутренней организации процессора доступ реализован независимо, при этом канал между процессором и памятью один.

Архитектура "Память инструкций как данные" (Instruction-memory-as-data) – реализуется возможность читать и писать данные в память программ. Позволяет генерировать и запускать машинный код.

Архитектура "Данные как память для инструкций" (Data-memory-as-instruction) – реализует возможность запуска инструкций из памяти команд. Также позволяет генерировать и запускать машинный код, при этом параллельный доступ в некоторых экземплярах реализуется за счет возможности параллельной работы с разными сегментами памяти.

Описание:

- Разрядность процессора: 8 бит.
- Организация памяти: гарвардская, с отдельными блоками памяти для команд и данных.
- Внешние устройства (для данной работы это светодиоды, двухпозиционные переключатели и тактовые кнопки) отображаются в адресное пространство данных. Работа с ними выполняется по запросу.
- Регистры:
    - PC -- счетчик команд
    - IR -- регистр инструкций
    - AR -- регистр адреса операнда
    - C -- флаг переноса/займа
    - Z -- флаг нуля
- Команды выполняются за 2 или 3 такта (в зависимости от типа команды):
  1) Выборка команды

  2) Выборка операндов

  3) Выполнение команды.

- Подсистема обработки прерываний и команды вызова подпрограмм отсутствует.

Гарвардская архитектура широко применяется во внутренней структуре современных высокопроизводительных микропроцессоров, где используется отдельная кэш-память для хранения команд и данных.

13.Механизм микроопераций и его роль в развитии компьютерных систем. Особенности и ограничения. Применение на практике.

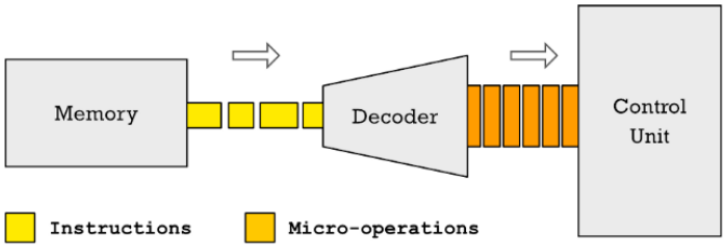
1. Первые процессоры могли быть реализованы полностью на аппаратном уровне. Основным инструментом программирования являлся ассемблер – низкоуровневый язык программирования непосредственно отображаемый на систему команд процессора. У разработчиков был запрос на:

- высокую производительность;
- удобство программирования.

Традиционный на сегодня метод повышения удобства программирования через введение абстракций приводил к:

- высоким накладным расходам (использование подпрограмм);
- раздуванию исходного кода (inline).

Поэтому одним из путей улучшения user-experience стало наращивание системы команд с целью естественной реализации всех необходимых инструкций. Причем данное наращивание было связано не только и не столько с добавлением новых арифметических операций, сколько на создание большого количества типовых команд. К примеру, команды позволяющие использовать большее количество аргументов, сохранять результат не в регистр, а сразу в память и т.п. Как следствие -- для множества команд нет необходимости добавлять новую логику в процессор (шины данных, регистры, сигнальные линии), необходимо только специфическим образом использовать уже имеющиеся.



Инструкции прибывают из памяти, обычно из высокоскоростного кэша. Далее они входят в декодер, который разбивает каждую инструкцию на одну или несколько микроопераций. Хотя они выполняют меньше одной инструкции, они значительно больше.

Следствие – необходимость "программирования" системы команд процессоров, что вылилось в понятие микрокода – программы, реализующей набор инструкций процессора.

14.Что такое CISC? Роль в развитии компьютерных систем. Применение на практике. Достоинства и недостатки. Отличия от архитектуры фон Неймана.

Аббревиатура CISC обозначает Complex Instruction Set Computer.

1. Архитектура CISC в значительной степени стала возможна благодаря возможностям микропрограммного управления. Микропрограмма позволяет относительно легко (по сравнению с разработкой аппаратуры) формировать большое количество команд со сложным поведением. Поднять уровень

машинных инструкций ближе к уровню языков программирования высокого уровня. К примеру:

- относительно легко поддержать все интересующие варианты адресаций для всех типов команд;
- реализовать операции с произвольным набором аргументов, к примеру инструкцию для расчета многочленов в рамках одной инструкции;
- реализовать операции работающие в потоковом режиме.

Основными преимуществами такого подхода является:

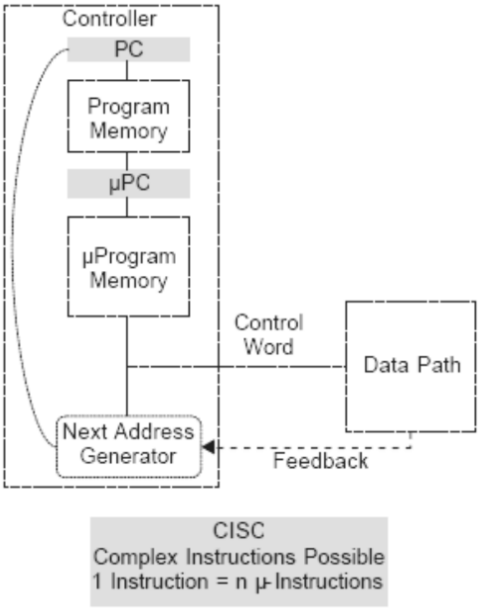
- удобство использования ассемблера как инструмента программирования;
- сокращение затрат на доступ к памяти команд, так как гибкость позволяет сократить количество инструкций;
- возможность обновления микропрограммного управления позволяет обновить "прошивку" процессора, и тем самым улучшить его функциональность уже после производства аппаратуры;
    - Существуют семейства процессоров и инструментальных средств для них активно эксплуатирующие данную возможность, к примеру: УВК «Самсон» – базовая ЭВМ РВСН. За счет оптимизации микрокода под конкретный алгоритм позволяет сократить объем программ (машинное представление), а также оптимизировать лишние операции.

Но также он сопряжен с рядом недостатков:

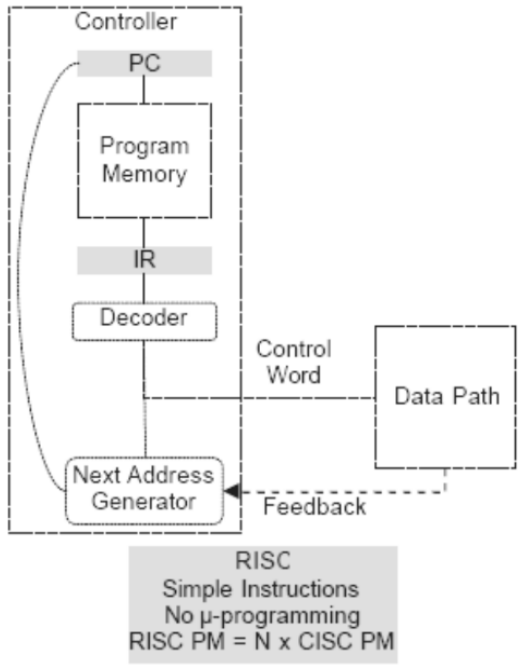
- Необходимо хранить микрокод непосредственно в процессоре (необходим мгновенный доступ к ней), где хранение данных весьма дорогой процесс, по сравнению с внешней памятью. Чем сложнее система команд, тем больше необходимо памяти.
- Развитый ассемблер требует большого объёма знаний о процессоре от разработчика, кроме того лёгкость развития и модернизации системы команд провоцирует рост разнообразия процессорных архитектур, который в свою очередь накладывает серьезные требования на инструментальные средства.
- Наличие большого количества программного обеспечения (пусть и в виде микрокода) сопряжено со всеми сложностями программирования. В данном случае это означает что программное обеспечение нуждается в дорогостоящей отладке и оптимизации, что значительно повышает стоимость разработки.
- Большое разнообразие команд делает их относительно уникальными с точки зрения формата, размера команды, длительности исполнения и количества доступов к памяти. Как следствие это усложняет:
    - реализацию оптимизаций в рамках процессора (конвейерное исполнения, суперскалярность и т.п.);

- поддержка со стороны инструментальных средств (компиляторы, дизассемблеры, отладчики и т.п.), в качестве примера можно посмотреть о сложностях разработки дизассемблеров для современных процессоров x86.

Примечание: CISC процессор не обязательно нуждается в микрокоде, но как правило подразумевает его наличие.



15.Что такое RISC? Роль в развитии компьютерных систем. Применение на практике. Достоинства и недостатки. Отличия от архитектуры фон Неймана.

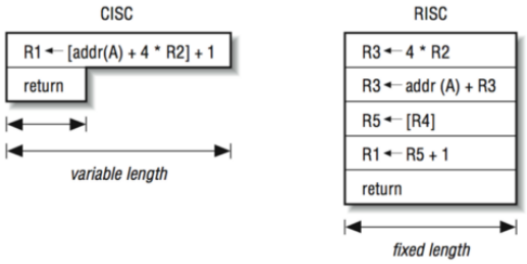


RISC — Reduced Instruction Set Computer

1. Ключевой технологией для RISC процессоров стали языки высокого уровня, которые позволили разработчикам ПО уйти от написания исходного кода на уровне ассемблера, а значит вопросы удобства написания кода сменились вопросам удобства генерации кода.

   Основная идея RISC процессора заключается в том, чтобы лучше сделать минимальный набор унифицированных команд ориентированных на быстрое исполнение, а "сложное поведение" реализовывать их последовательностями. Это позволило:

   - освободить в процессоре площадь от элементов сложных декодеров команд, что в частности позволило расширить регистровые файлы;
   - освободить большое количество памяти микрокоманд для кеша команд, что отчасти компенсировало необходимость более частой выборки команд;
   - добиться малого количества быстро исполняемых команд;
   - "однообразность" команд открыла возможность к оптимизациям производительности на уровне инструкций (примечательно, что традиционный пример RISC процессора, приведенный ниже, уже имеет конвейерную организацию).



Инструкции CISC могут быть любой длины. Максимальная теоретическая длина инструкции x86 может быть бесконечной, но на практике не превышает 15 байт. Инструкции RISC имеют ограниченную длину.

Ещё одним из "триггеров" к формированию RISC процессоров стала необходимость длительной отладки машинного кода, а кроме того, практика показала что переход от "большого CISC" процессора к "маленькому RISC" процессору не только радикально снизило сложность разработки последнего, но и повысило его производительность для "средней программы", так как большинство "сложных команд" относительно редко использовались на практике, а значит их менее эффективная реализация, не могла заметно снизить общую производительность системы.

Другим, следствием простоты и компактности RISC процессоров стало их широкое использование в рамках систем на кристалле, так как включить в неё ещё один RISC процессор – не сложная на сегодня задача при помощи которой можно реализовывать "сервисные модули", а также RISC процессор довольно просто адаптировать под конкретные нужды при помощи специализированных расширений.

Однако на сегодня RISC процессоры не заняли рынок полностью, потому что у CISC процессоров есть следующие преимущества:

- обратная совместимость, в первую очередь на персональных компьютерах (огромное количество написанного ПО должно работать как раньше, смена архитектуры не позволит использовать привычное ПО или снизит его производительность, что будет неприемлемо для большинства пользователей, в качестве примера можно посмотреть на первые попытки перехода Windows на ARM где-то около 2016 года);

- производство процессоров очень сильно зависит от серийности, в результате складывается немного парадоксальная ситуация в рамках которой лидер рынка (самый большой сбыт) будет иметь самую маленькую себестоимость производства, как следствие, преследователи будут находиться в заведомо проигрышной ситуации;

- есть основания полагать, что многие современные CISC процессоры содержат внутри так называемое RISC ядро, которое исполняет наборы инструкций генерируемых в результате интерпретации CISC команд.

Вкратце, если RISC так хороши, то почему CISC до сих пор так популярен?

  - обратная совместимость (ПК, у серверов все немного не так);
  - лидер рынка имеет самую большую серийность, поэтому недорогое производство.

16.Что такое NISC? Роль в развитии компьютерных систем. Применение на практике. Достоинства и недостатки. Отличия от архитектуры фон Неймана.

NISC (*no instruction set computing*) - это компьютерная архитектура и технология компилятора для проектирования высокоэффективных пользовательских процессоров и аппаратных ускорителей, позволяющая компилятору контролировать аппаратные ресурсы на низком уровне.

Роль в компьютерных системах и применение на практике:
Широко применяется во встраиваемых системах так как легко конвейеризуется, Очень хорошо вписывается в автономную архитектуру. Действительно хороший доступ к памяти. Высокая оптимизация.

Преимущества:

1. убрать ISA(instruction set architecture) как уровень абстракции и фактически достигнуть того уровня контроля за процессором, который есть у микрокода (фактически программой становится микропрограмма, единая на весь алгоритм), что позволяет сократить количество команд (не объём, так как из-за большого количества сигналов, как правило, программа становится больше по объёму) и накладные расходы на их реализацию;

2. значительно упростить процессор, по сути оставив в нем только те элементы, которые необходимы для вычислительных задач, перенеся всё что возможно в компилятор;

3. упрощения разработки процессоров, так как исключается крайне сложный и трудоемкий этап проектирования ISA, но в тоже время и ставится крест на бинарной совместимости между разными процессорами (что не критично в случае если речь идет о системах на кристалле и специализированном под задачу оборудовании);

4. простое внутреннее устройство процессора позволяет относительно легко генерировать сам процессор под задачу (допустим, для ПЛИС), а также этим обусловлено применения данной архитектуры в рамках высокоуровневого синтеза.

Недостатки:

Сложность кодирования(это исправляют современные компиляторы)

Отличия от архитектуры фон Неймана:

Исключение системы команд, что дает машинному коду прямой доступ к сигналам и элементам процессора.

17.Что такое стековый процессор? Роль в развитии компьютерных систем. Применение на практике. Достоинства и недостатки. Отличия от архитектуры фон Неймана.

Стек - простейшая структура данных, которая представляет собой коллекцию элементов, к которым применимы две основные операции:

- push - добавление элемента в коллекцию
- pop - исключение последнего добавленного в коллекцию элемента

Но эта простая структура данных позволяет нам осуществлять процедурные вызовы, автоматически выделять память, определять контекстные переменные и делать множество других важных программистских штук.

Лучшим примером стековых вычислений является язык FORTH.

Пример организации стека на функции для вычисления факториала:

```
: fac recursive
  dup 1 > IF
    dup 1 - fac *
  else
    drop 1
  endif ;
```

Let see the computational process step by step for expression `3 fac .`. All items in the list are following by a current data stack state.

- `[]` Push `3` in the stack.
- `[3]` Execute word (in another language, we call it "procedure") `fac`. `[3]` Execute word `dup` -- duplicates the stack's topmost element.
  - `[3, 3]` Push `1`.
  - `[3, 3, 1]` Execute word `>` -- pull two values from the stack, compare them and push the result back.
  - `[3, true]` Pull value from the stack and execute text `dup 1 - fac *` because it is `true`.
  - `[3]` Execute word `dup`.
  - `[3, 3]` Push `1`.
  - `[3, 3, 1]` Execute word `-` -- pull two values from the stack, subtract them and push the result back.
  - `[3, 2]` Execute word `fac` recursively.
    - ...
      - `[3, 2, 1]` Execute word `dup`.
      - `[3, 2, 1, 1]` Push `1`.
      - `[3, 2, 1, 1, 1]` Execute word `>`.
      - `[3, 2, 1, false]` Pull value from the stack and execute text `drop 1` because it is `false`.
      - `[3, 2, 1]` Drop one value from the stack.
      - `[3, 2]` Push 1. Why we drop `1` and push `1` again? We need this in case if someone tries to evaluate `0 fac .`.
      - `[3, 2, 1]` Return.
    - ...
  - `[3, 2]` Execute word `*` -- pull two values from the stack, multiply them and push the result back.
  - `[6]`.

Применение на практике:

Сейчас FORTH используется в основном как встроенный язык для пользовательской кастомизации (например, в Open Firmware project) т. к. это его прямая имплементация (реализация?). Также некоторые проекты всё еще юзают его, как основной ЯП.

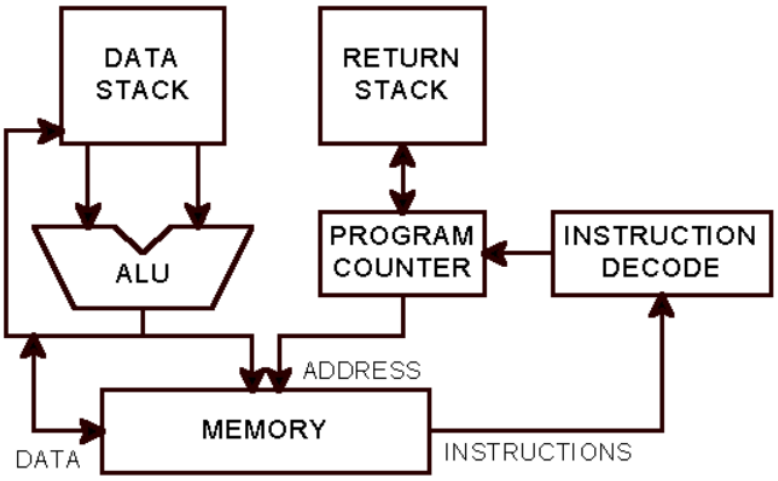
Известный пример успешного применения Форта — его использование в программном обеспечении глубоководного спускаемого аппарата, при поисках «Титаника» в 1985 году. Также Форт был применён в программном обеспечении спектрографа на Шаттле, в микромодулях управления искусственными спутниками Земли, для системы управления в аэропорту Эр-Рияда, системах компьютерного зрения, автоматизации анализа крови и кардиологического контроля, карманных переводчиках.

Плюсы и минусы:

+ К преимуществам стековых машин относят компактность кода, простоту компиляции и интерпретации кода, а также компактность состояния процессора.
- В случае стека фиксированного размера (который у Пенского в теории и на картинке ниже), мы имеем проблемы с компилятором и/или операционной системой, которые обязаны думать о том, что произойдет, если стек переполнится или опустеет и как отработать соответствующее прерывание. Если обработка прерывания отсутствует, мы имеем дело с микроконтроллером, обреченным на то, чтобы быть программируемым вручную.
- Аппаратно реализованный стек подразумевает строгую последовательность происходящих событий, а, следовательно, и невозможность использовать неявный параллелизм программ.
- Программирование для регистровых машин более «технологично». Имея всего лишь несколько регистров общего назначения, «вручную» легко можно создать код, который будет обращаться к памяти лишь тогда, когда этого действительно нельзя избежать.

Отличие от фон Неймана:

Processor architecture:



придумайте

Дополнительно:

Блок-схема стекового процессора приведена на рис.1. В отличие от «обычного» микропроцессора, стековый процессор содержит два стека — стек данных и стек возвратов. Стек возвратов используется для возвратов из подпрограмм, как и у «обычных» микропроцессоров, а вот стек данных — это «привилегия» стекового процессора. Именно через стек данных производится передача параметров при вычислениях.

По сути стековые процессоры это уникальное архитектурное решение. Такой тип процессоров оказал широкое влияние на микроэлектронику в СССР.
Стековые процессоры имеют гораздо меньшую требовательность к памяти и низкое энергопотребление, по сравнению, например с RISC процессором. Однако они не такие мощные. Их было бы логично применять в маленьких встроенных системах, где

не нужна очень высокая производительность. От архитектуры Фон-Неймана отличие очевидно. В архитектуре фон неймана предполагается наличие регистров. Здесь же все взаимодействие с алу осуществляется через стек.

18.Что такое VLIW? Роль в развитии компьютерных систем. Применение на практике. Достоинства и недостатки. Отличия от архитектуры фон Неймана.

Что такое VLIW?

VLIW — Very Long Instruction Word. То есть архитектура процессоров, характеризующаяся возможностью объединения нескольких простых команд в так называемую связку. Входящие в нее команды должны быть независимы друг от друга и выполняться параллельно. Таким образом, из нескольких независимых машинных команд транслятор формирует одно «очень длинное командное слово». Откуда и пошло название архитектуры.

Пример:

Рассмотрим работу модельного VLIW-процессора с двумя арифметическо-логическими устройствами (АЛУ). Пусть нам надо сложить четыре числа, находящиеся в регистрах R1, R2, R3 и R4. Тогда псевдокод может выглядеть так:

```
R5=R1+R2, R6=R3+R4 ; каждое АЛУ складывает свою пару чисел
R0=R5+R6, NOP      ; первое АЛУ находит сумму, второе простаивает
```

Длина командного слова может быть намного длиннее, например в процессоре эльбрус может выполняться до 23 параллельных инструкций

Роль в развитии компьютерных систем.

Архитектура с командными словами сверхбольшой длины или со сверхдлинными командами известна с начала 80-х из ряда университетских проектов. Идея VLIW базируется на том, что задача эффективного планирования параллельного выполнения команд возлагается на «разумный» компилятор. Такой компилятор вначале анализирует исходную программу. Цель анализа: обнаружить все команды, которые могут быть выполнены одновременно, причем так, чтобы между командами не возникали конфликты. В ходе анализа компилятор может даже частично имитировать выполнение рассматриваемой программы. На следующем этапе компилятор пытается объединить такие команды в пакеты (связки), каждый из которых рассматривается как одна сверхдлинная команда.

Де факто VLIW является логическим продолжением RISC архитектуры. VLIW -- делает попытку повторить элементы архитектуры RISC и переложить сложность суперскалярного процессора на компилятор, а именно -- принять решение о том какие операции должны быть параллельны, а какие нет. Для этого формируются очень длинные машинные команды, в которых кодируются операции для всех устройств

обработки (АЛУ), входящие в состав процессора. На данный момент широкое распространение получила в мобильных процессорах Snapdragon, ранее применялась в GPU от компании AMD, но позже (В 2012 году) от нее отказались. Можно сказать что данная архитектура стала логической связью между прошлым и будущим архитектур.

Применение на практике.

GPU от Nvidia имеет архитектуру ARMv8-A в собственном ядре с микроархитектурой Denver. В нём используется комбинация простого аппаратного декодера ARM-кода и технологии «Dynamic Code Optimization» программной рекомпиляции ARM-кода во внутреннюю систему команд. Denver представляет собой суперскалярную архитектуру с широким командным словом VLIW без возможностей по внеочередному исполнению команд (in-order).

GPU от AMD также использовали модифицированную VLIW архитектуру до 2012 года, после которого вместо VLIW/MIMD стали использоваться RISC/SIMD.

Первые же VLIW-процессоры были разработаны в конце 1980-х компаниями Cydrome (1984−1988), MultiFlow (1984—1990), Culler.

Как говорил ранее, на данный момент VLIW архитектура используется в процессорах Qualcomm Snapdragon. Можно сказать это компания которая производит мощнейшие процессоры для мобильных устройств. Хоть многие именитые фирмы, такие как Apple уже используют собственные процессоры. Однако давайте будем честны - большинство людей по крайней мере в России ходят с телефонами от менее известных китайских компаний из нижне-среднего ценового сегмента, а все эти компании как раз берут процессоры Snapdragon, что говорит о большой распространенности VLIW архитектуры на мобильном рынке

Также VLIW архитектуру берут Российские процессоры Эльбрус, но о мертвых либо хорошо либо ничего.

Достоинства и недостатки.

Плюсы

Выполняемый VLIW параллелизм операций позволяет:

- повысить уровень параллелизма программ по сравнению с суперскалярными процессорами, так как компилятор не ограничен относительно небольшим количеством инструкций, а имеет доступ ко всему программному обеспечению;
- упростить процессор, так как он более не требует в своем составе сложного диспетчера;

- снизить энергопотребление.

На сегодня VLIW процессора широко применяются для решения специализированных вычислительных задач, например: обработка мультимедиа информации, расчёты и т.п., в то время как их применение для задач общего назначения, как правило, уступает решениям на базе суперскалярных процессоров.

Минусы

В то же время, код для VLIW обладает невысокой плотностью. Из-за большого количества пустых инструкций для простаивающих устройств программы для VLIW-процессоров могут быть гораздо длиннее, чем аналогичные программы для традиционных архитектур.

Архитектура VLIW выглядит довольно экзотической и непривычной для программиста. Из-за сложных внутренних зависимостей кода, программирование на уровне машинных кодов для VLIW-архитектур человеком вручную является достаточно сложным. Приходится полагаться на оптимизацию компилятора.

То есть основные проблемы VLIW:

- Низкая плотность исходного кода. Вместо того, чтобы компактно представлять сложные параллельные процессы система команд VLIW стала часто представлять простой последовательный смешанный код, оставляя часть инструкции пустыми, но при этом затрачивая энергию на их передачу, обработку и хранение.
- Ширина команды VLIW процессора (архитектура системы команд) накладывает ограничение на микроархитектуру процессора, что затрудняет независимую разработку процессора и компилятора. Также это затрудняет бинарную совместимость.
- Широкие команды прекрасно справляются с "плоским кодом" (не использующим процедуры). Особенно это справедливо для математических расчётов. В тоже время вызовы процедур (и отчасти условные переходы) "рвут контекст распараллеливания", что значительно усложняет работу компилятору. Эффективный код требует большого количества оптимизаций и спекулятивных вычислений, что не всегда удается реализовать в рамках инструментария, а также требует аппаратной поддержки со стороны процессора.
- Высокопроизводительная работа VLIW процессора требует активного использования спекулятивных вычислений. К примеру: вынесение операций из тела "условного оператора" для их спекулятивного вычисления перед или во время условного перехода.

Дополнение про EPIC из конспекта Пенского:

Одним из вариантов развития VLIW процессоров является: Explicitly parallel instruction computing (EPIC).

> Инженеры HP и Intel предполагали, что архитектура микропроцессора EPIC (Explicitly Parallel Instruction Computing) («вычисление с явным параллелизмом машинных команд») на основе VLIW способна преодолеть врождённые ограничения RISC и поможет увеличивать производительность процессоров без увеличения тактовой частоты, выполняя всё больше и больше команд за такт, при этом планировщик инструкций, предсказание ветвлений и прочее удалялись из процессора и переносились в компилятор. В теории, это должно было освободить на микросхеме место для дополнительных модулей выполнения и увеличить параллельную производительность. Но на практике это не сработало. По-настоящему эффективных компиляторов не было создано. К тому же EPIC настолько радикально отличалась от других архитектур, что ARM и x86 можно считать братьями-близнецами по сравнению с ней.

Отличия от архитектуры фон Неймана.

Архитектура фон Неймана является последовательной архитектурой. То есть в чистой архитектуре фон Неймана процессор одномоментно может либо читать инструкцию, либо читать/записывать единицу данных из/в памяти. То и другое не может происходить одновременно, поскольку инструкции и данные используют одну и ту же системную шину.

В то время как VLIW является суперскалярной архитектурой, то есть имеет способность выполнения нескольких машинных инструкций за один такт процессора путем увеличения числа исполнительных устройств. Появление этой технологии привело к существенному увеличению производительности, в то же время существует определенный предел роста числа исполнительных устройств, при превышении которого производительность практически перестает расти, а исполнительные устройства простаивают.

Если информации оказалось недостаточно, то вот основные источники

https://studfile.net/preview/5837724/page:3/
https://www.osp.ru/os/2008/06/5340894
https://habr.com/ru/post/535926/
http://www.nsc.ru/win/elbib/data/show_page.dhtml?77+741

Ну и вики/конспект Пенского

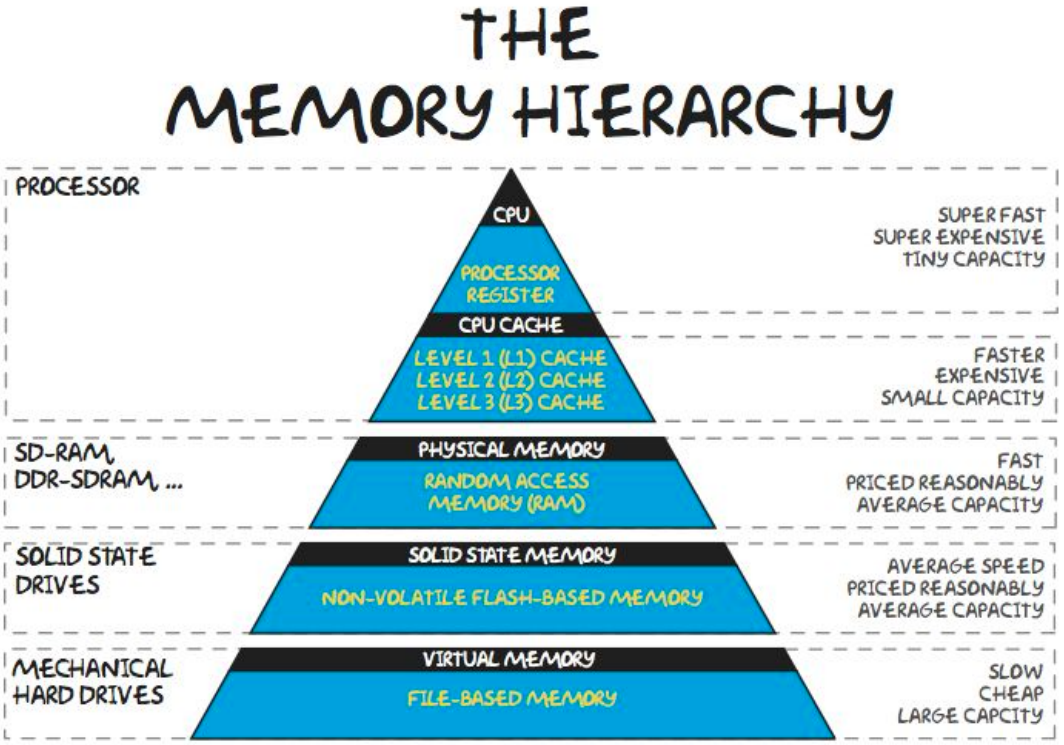
19.Иерархия памяти. Виды памяти. Особенности использования на практике. Устройство памяти с произвольным доступом.

Память в компьютерных системах решает две основные задач

- хранение исходных данных, необходимых для работы системы (программное обеспечение, настройки и начальные значения);
- хранение и обновление рабочих данных, к обработке которых можно свести функционирование компьютерной системы (видео поток, данные для расчетов, текстовый документ).

Существует огромное количество разных типов памяти компьютеров, для которых справедливо следующее правило: **с ростом скорости памяти растёт её стоимость (производства и размещения в нужном месте)**.

Иерархия памяти:



{#fig:memory-hierarchy}

Иерархия памяти показана на схеме выше. Чем выше -- тем быстрее. Чем уже -- тем меньше (по объему). На ней показаны:

- память размещённая в процессоре (по крайней мере сегодня):
  - регистры (работают на частоте процессора);
  - кеши (могут также работать на частоте процессора, кэш первого уровня имеет возможность доступа к данным за один такт);
- основная / физическая / оперативная память, как правило размещенная в непосредственной близости от процессора с высокоскоростным произвольным доступом (но намного более медленным, по сравнению с внутри-процессорной памятью);
- твердотельные накопители, энергонезависимая память позволяющая хранить большой объём данных и обеспечивает произвольный доступ к ней;

- механические накопители.

Виды памяти:

Виды памяти представлены в пирамиде памяти.

Но, память также подразделяется по типу доступа:

- с произвольным доступом, память может предоставить данные по любому адресу с одинаковой задержкой относительно прошлого запроса;
- с последовательным доступом, причем сюда попадает последовательный доступ в понимании потока данных (сетевой, ввод и т.п.), магнитное ленты, жёсткие диски и т.п.
- возможны и гибридные варианты: библиотека магнитных лент, где доступ к магнитной ленте реализуется произвольно, а доступ к данным в рамках ленты -- последовательно.

Особенности использования на практике:

С практической точки зрения последовательный доступ в подавляющем большинстве случаев говорит не о невозможности произвольного доступа, а очень высокой длительности. К примеру жёсткий диск, где доступ к данным завязан на физическое позиционирование головки над магнитным диском, а значит последовательное считывание данных может быть очень быстрым, в то время как перемещение и поиск будет долгими за счёт ожидания перемещения головки и поворота диска.

P.S: если говорить о видах памяти, представленных в пирамиде памяти, то особенностями использования на практике является соотношение стоимости/объема/скорости

Устройство памяти с произвольным доступом:

Произвольный доступ к данным памяти реализуется за счет особой организации хранилища данных. Как правило это массив ячеек памяти со структурой приведенной на схеме. Регулярность структуры упрощает её масштабирование и повышает плотность хранения данных. Крупные хранилища данных имеют более сложную структуру, что позволяет сократить объём линий данных.

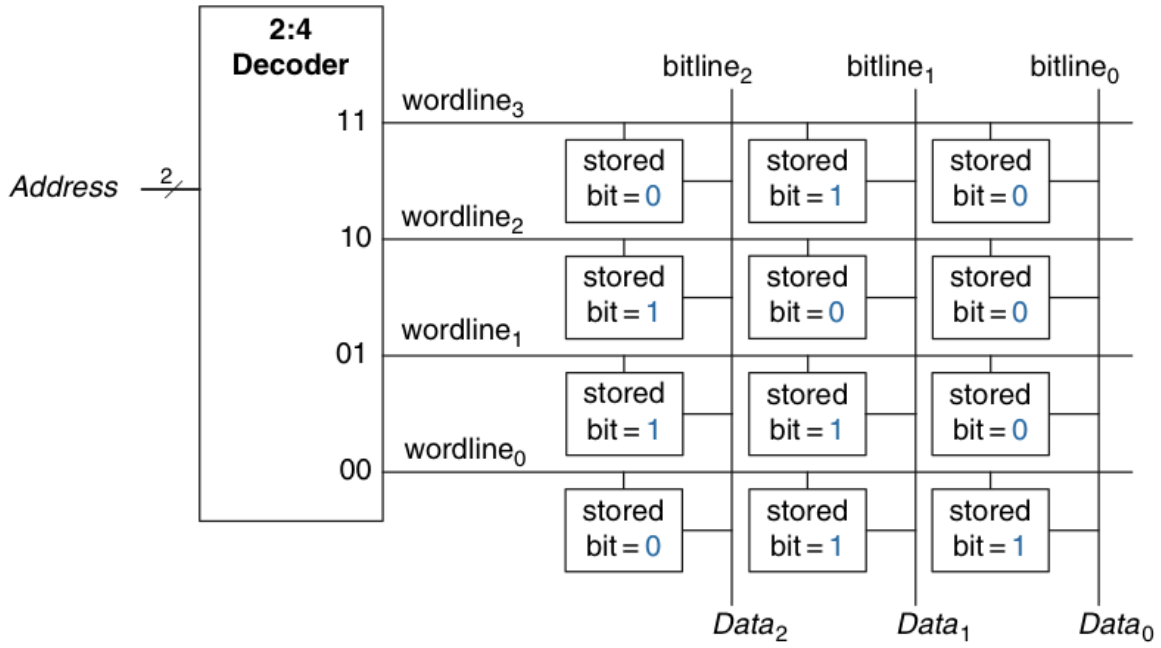


Figure 5.42 4 × 3 memory array

На схеме показан массив ячеек памяти $4x3$ (4 слова по 3 бита), где:

- Address -- адрес ячейки памяти шириной 2 бита.
- Decoder (дешифратор), транслирующий адрес из позиционной системы счисления в одноединичный код. На схеме показано значения адреса для активации соответствующей линии.
- $wordline_i$ -- линия активирующая взаимодействие с ячейками памяти хранящими одно машинное слово.
- $bitline_i$ -- линия, на которую выставляется / читается значение бита определённой позиции.
- stored bit -- собственно ячейка памяти.

Для ячеек памяти существует большое количество вариантов реализации. На схемах ниже обозначены основные из них.

Memory Type	Transistors per Bit Cell	Latency
flip-flop	~20	fast
SRAM	6	medium
DRAM	1	slow

1. **Read Only Memory (ROM)**
   Память только для чтения. Может реализовываться как путем физического размещения / неразмещения транзисторов (тогда реализуется в рамках

производства), так и путем однократного программирования, к примеру, за счет "пережигания перемычек" (см. PROM).

2. **Static Random Access Memory (SRAM)**

Статическая память с произвольным доступом. Особенность такой памяти -- хранение данных при помощи состояния группы транзисторов, что с одной стороны обеспечивает:

- быстрый доступ к данным на чтение и запись;
- долговременное хранение данных (значение будет храниться произвольное время, если не отключать питание с ячейки).

В тоже время для реализации требуется довольно большое количество транзисторов, а значит плотность ячеек памяти будет не очень высока.

3. **Dynamic Random Access Memory (DRAM)**

Динамическая память, главная особенность которой является то, что значение хранится не в виде "стабильного состояния группы транзисторов", а в конденсаторе. Как известно, будучи заряженным любой конденсатор имеет токи утечки (а значит сохраненное значение рано или поздно "утечёт") и не может несколько раз отдать записанное в него значение. Как следствие -- DRAM помимо непосредственно ячеек памяти нуждается в контроллере памяти, который будет осуществлять *регенерацию* хранимых в памяти данных во избежании их утери, а также корректный доступ к ним. Наличие такого контроллера требует дополнительной логики, а также снижает скорость доступа к данным, так как доступ к ним в процессе регенерации становится невозможным.

В то же время, данный тип памяти требует наличия всего одного транзистора и конденсатора, что радикально увеличивает плотность ячеек памяти по сравнению с SRAM.

20.Механизм кеширования в компьютерных системах. Основные свойства кеш памяти. Иерархия кеш памяти. Условия эффективной работы кеш памяти.

Механизм кеширования в компьютерных системах.

На примере процессора:

При доступе к данным **на чтение** процессор сперва запрашивает данных у кеша и в случае:

- **Если искомый тег не найден фиксируется *кеш промах*** *(cache miss)*, данные запрашиваются из основной памяти в кеш, после чего передаются в процессор. Причем запрос данных в кеш -- сложный многоэтапный процесс с непредсказуемой длительностью, так как:

    - вся память кеша уже занята (типовая ситуация), значит необходимо принять решение о том, какие данные будут вытеснены / замещены (evict);
    - доступ к основной памяти может быть заблокирован другим процессом или внутренними процессами памяти (если это DRAM).
- **Если искомый тег найден, то фиксируется *кеш попадание*** *(cache hit)* и данные из кеша передаются прямо в процессор.

Эффективность работы кешей характеризуются процентом обращений к кэшу, когда в нём найден результат, называется уровнем попаданий (hit rate), или коэффициентом попаданий в кэш.

Для выбора вытесняемой строки кэша используется эвристика, называемая политика замещения (англ. replacement policy). Основной **проблемой алгоритма является предсказание**, какая строка вероятнее всего не потребуется для последующих операций. Качественные предсказания сложны, и **аппаратные кэши используют простые правила**, такие, как **LRU** (least recently used). **Пометка** некоторых **областей памяти** как **не кэшируемых** (non cacheable) **улучшает производительность** за счёт запрета кэширования редко используемых данных. **Промахи для такой памяти не создают копии данных в кэше**.

При доступе к данным **на запись** процесс немного сложнее, так как подразумевает перенос внесенных изменений из кеша в основную память, реализация которого регулируется политикой записи. Выделяются следующие варианты:

- *Немедленная запись* (сквозная, write-through). Каждое изменение вызывает **синхронное обновление данных в основной памяти**, что делает доступ к памяти на запись медленным, причем зачастую медленнее чем если бы кеша не существовало (из-за лишнего шага). В тоже время, такой подход позволяет поддерживать основную память в наиболее актуальном состоянии.
- *Отложенная запись* (обратная запись, write-back). **Обновление происходит в случае вытеснения записи данных**, периодически или по запросу, что позволяет группировать записи в память, в том числе и игнорируя промежуточные обновления.
    - Для отслеживания модифицированных элементов данных записи кэша хранят признак модификации (изменённый или «грязный»).
    - Промах в кэше с отложенной записью может потребовать два обращения к основной памяти: первое для записи вытесняемых данных из кэша, второе для чтения необходимого элемента данных.
    - Обратная запись может приводить к неконсистентному состоянию кеша и основной памяти. Для самого процессора эта рассогласованность не заметна, но перед обращением к памяти другого ведущего системной шины (контроллера DMA, bus-master-устройства шины PCI) кэш должен быть записан в память принудительно.
- Гибридные варианты. **Кэш может быть со сквозной записью**, но для уменьшения количества транзакций на шине записи могут временно помещаться в очередь и объединяться друг с другом.

Основные свойства кеш памяти.

Кеш -- промежуточный буфер с быстрым доступом к нему, содержащий информацию, которая может быть запрошена с наибольшей вероятностью.

Кеш предназначен для ускорения доступа к памяти за счет:

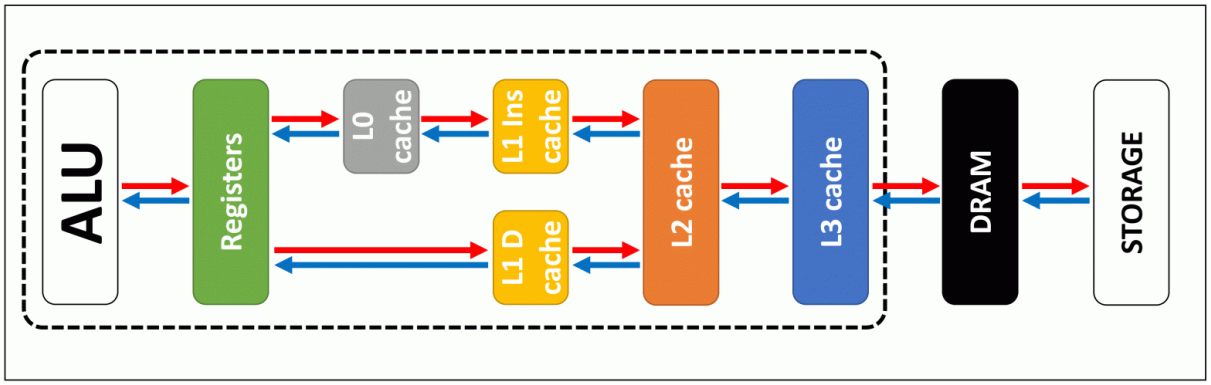
- **изменения режима доступа**;
- **использования более быстрой памяти для буфера** (кеша);
- **увеличение пропускной способности канала,** к примеру за счет изменения типа канала связи или машинного слова (к примеру, процессор -- кеш использует машинное слово процессора, кеш -- основная память может использовать ширину слова в 2 и более раз большую).

Дело в том что **кеш является не просто памятью,** но также **имеет в своём составе логику управления**, которая включает:

- **логика поиска кеш линии по тегу**;
- **логика вытеснения / замещения**;
- **логику предзагрузки** (prefetch) данных (анализ загруженных инструкций) и программ (загрузка следующей строки и заданной строки);
- **логика взаимодействия с основной памятью** (группировка операций записи, правила записи данных в память);
- **логику синхронизации разных уровней кешей**.

В общем случае справедливо правило: **чем больше данных может храниться в кеше, тем больше действий необходимо проделывать для работы с ними.** Действия могут проделываться либо в рамках логических схем (требуют площади для реализации), так и в рамках управленческих автоматов (требуют времени для работы). К примеру, если есть кеш с двумя линиями, мы можем проверить вхождение данных в кеш за один такт имея два компаратора для тега или за два такта имея один компаратор. Как следствие при увеличении размеров кеша мы должны либо увеличивать площадь управляющих схем (а значит и энергопотребление), либо длительность доступа к памяти.

Иерархия кеш памяти.



В современных процессорах выделяют следующие уровни кеширования:

- L0 -- наиболее **специализированный кеш** (обычно размером 256 байт) который далеко не всегда выделяется в рамках процессоров. Как правило он имеет узкую специализацию, к примеру: предназначен для **кеширования стека, кеширования целочисленных переменных, кеширование чисел с плавающей точкой** для FPU (floating-point unit) и т.п. Как правило кеш данного уровня обеспечивает доступ к данным за один такт, что обеспечивается его непосредственной интеграцией в процессор.

- L1 cache (level 1 cache), **самый быстрый кеш**. По сути, он **является неотъемлемой частью процессора**, поскольку расположен на одном с ним кристалле и входит в состав функциональных блоков. В современных процессорах обычно L1 **разделён на два кэша**: 1) кэш команд (инструкций) и 2) кэш данных (в соответствии с Гарвардской архитектурой). Большинство процессоров без L1 не могут функционировать. L1 работает на частоте процессора, и, в общем случае, обращение к нему может производиться каждый такт. Зачастую является возможным выполнять несколько операций чтения/записи одновременно.

    - Примечание: кэш команд может быть реализован на разном уровне, в частности для процессоров Pentium 4, отличающихся высокой частотой и сверхдлинным конвейером, кеширование производилось на уровне микроинструкций.
- L2 cache, который обычно, **как и L1, расположен на одном кристалле с процессором**. В ранних версиях процессоров L2 реализовывался в виде отдельного набора микросхем памяти на материнской плате. **Объём L2 от 128 кбайт до 1-12 Мбайт**. В современных многоядерных процессорах кэш второго уровня, находясь на том же кристалле, является памятью раздельного пользования -- при общем объёме кэша в n Мбайт на каждое ядро приходится по n/c Мбайта, где c -- количество ядер процессора.

- L3 **может достигать 24 Мбайт и более**, медленнее предыдущих кэшей, но всё равно **значительно быстрее, чем оперативная память**. В многопроцессорных системах находится в общем пользовании и предназначен для синхронизации данных различных L2, что особенно актуально для многоядерных систем

поддерживающих параллельное программирование.

- L4, на сегодня весьма **экзотический вариант кеша**, применение которого оправдано только для многопроцессорных высокопроизводительных серверов и мейнфреймов. Обычно он реализован отдельной микросхемой. Как отмечают специалисты, по видимому, кеш L4 будет реализовываться при помощи eDRAM (что позволит обеспечить большой объём хранимых данных) и будет ориентирован в первую очередь на повышение пропускной способности к основной памяти, чем на быстрый доступ к данным.

Виды кеш промахов:

- **Промах по чтению из кэша инструкций**. Обычно дает очень большую задержку, поскольку процессор не может продолжать исполнение программы (по крайней мере, текущего потока исполнения) и вынужден простаивать в ожидании загрузки инструкции из памяти.
- **Промах по чтению из кэша данных**. Обычно дает меньшую задержку, поскольку инструкции, не зависящие от запрошенных данных, могут продолжать исполняться, пока запрос обрабатывается в основной памяти. После получения данных из памяти можно продолжать исполнение зависимых инструкций.
- **Промах по записи в кэш данных**. Обычно дает наименьшую задержку, поскольку запись может быть поставлена в очередь и последующие инструкции практически не ограничены в своих возможностях. Процессор может продолжать свою работу, кроме случаев промаха по записи с полностью заполненной очередью.

Условия эффективной работы кеш памяти.

При этом, эффективность кешей сильно связана с локальностью данных (пространственной и временной). Локальность данных (Data Locality) – это близость данных к обрабатывающему их коду. Можно выделить два несколько парадоксальных недостатка кешей хорошо демонстрирующий сложность выбора их размера :

1. **Когда наборы данных очень большие**. Кэши просто плохо работают, когда наборы данных очень велики, имеют низкую временную или пространственную локальность.
2. **Когда кэши работают хорошо**. Когда кэши работают хорошо, локальность очень высока, то есть, по определению, большая часть кэша простаивает большую часть времени

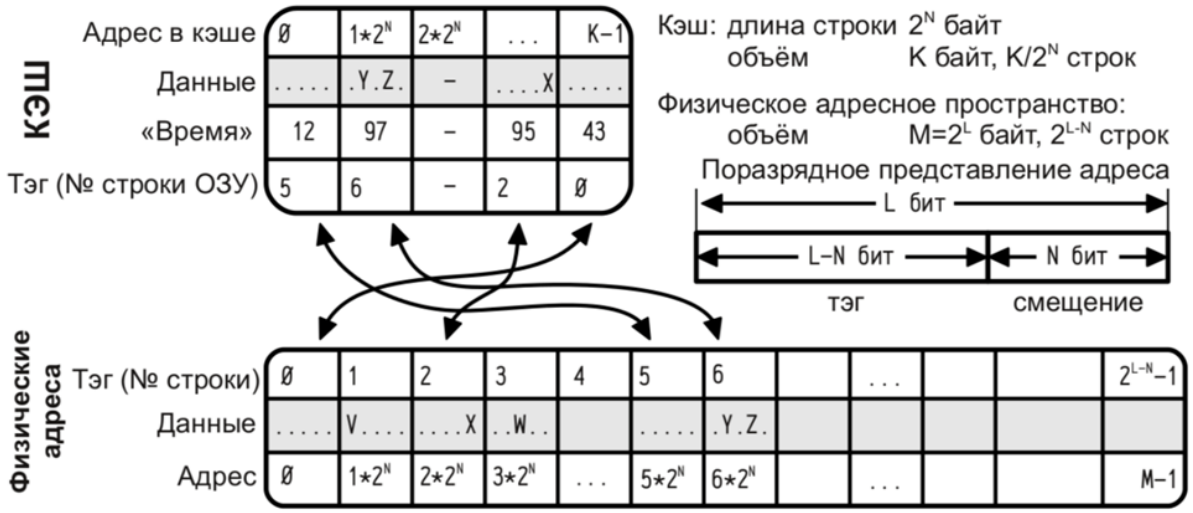
21. Ассоциативность кеш памяти. Детальное описание принципов работы кеш памяти с разным уровнем ассоциативности.

Ассоциативность кэша - характеризует то, как и какие области памяти могут быть отображены на кэш линии. Отображает её логическую сегментацию, которая вызвана тем, что последовательный перебор всех строк кэша в поисках необходимых данных потребовал бы десятков тактов и свёл бы на нет весь выигрыш от использования встроенной в ЦП памяти. Поэтому ячейки ОЗУ жёстко привязываются к строкам кэш-памяти (в каждой строке могут быть данные из фиксированного набора адресов), что значительно сокращает время поиска.

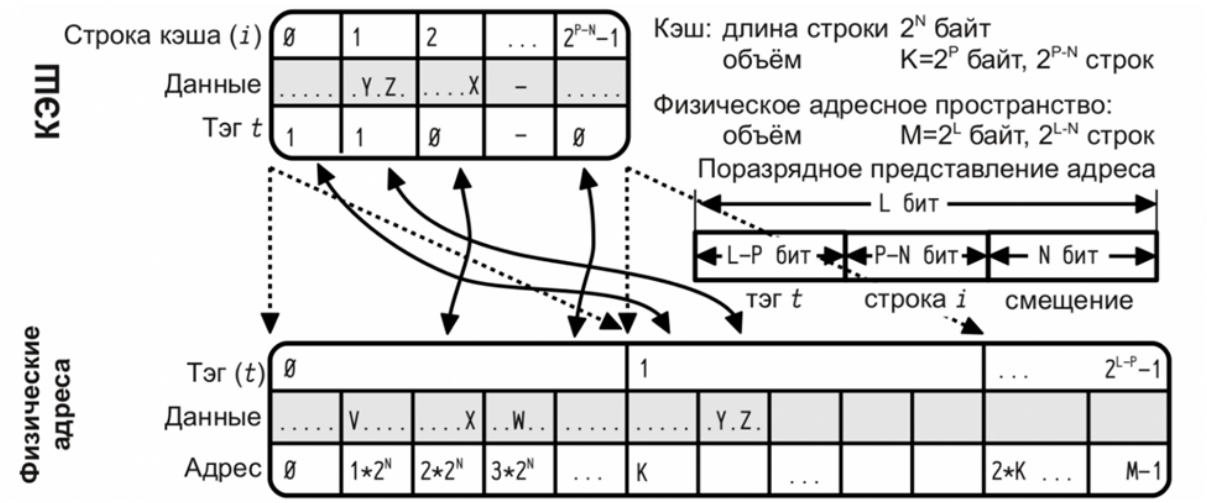
В пределе ТЭГ является адресом в оперативной памяти, но на практике такой подход не является эффективным. Первым делом адрес памяти целесообразно обрезать справа (младшие биты), что позволяет:

- уменьшить объём служебных данных кэша (кол-во кэш линий * кол-во обрезанных бит);
- обеспечить доступ к памяти с учётом ширины машинного слова (если внешняя память адресуется на уровне байт или 8 бит, то большая часть фактической работы с памятью рассчитывает на доступ по словам, что сегодня обычно 64 бита) или больше.

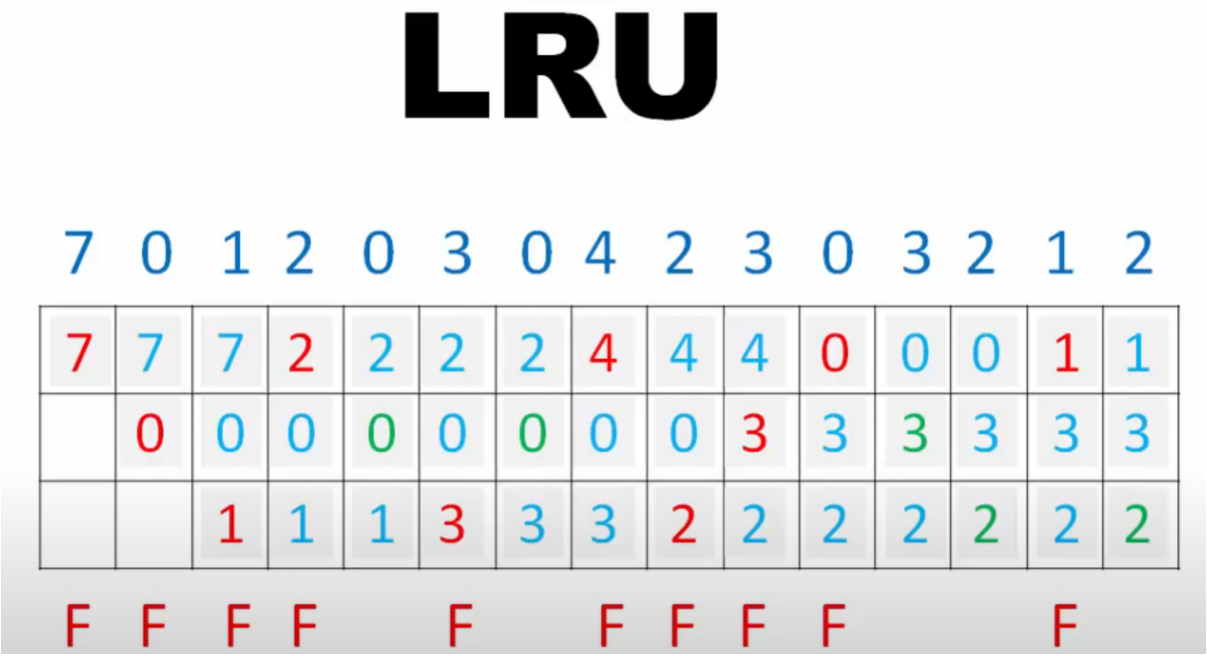
Полностью ассоциативный кэш (fully associative cache) - любая строка памяти ОЗУ может быть отображена в любую строку кэша. Такой способ позволяет реализовать наиболее эффективный кэш с точки зрения работы, но в то же время требует либо большого количества логики, либо длительного времени работы.

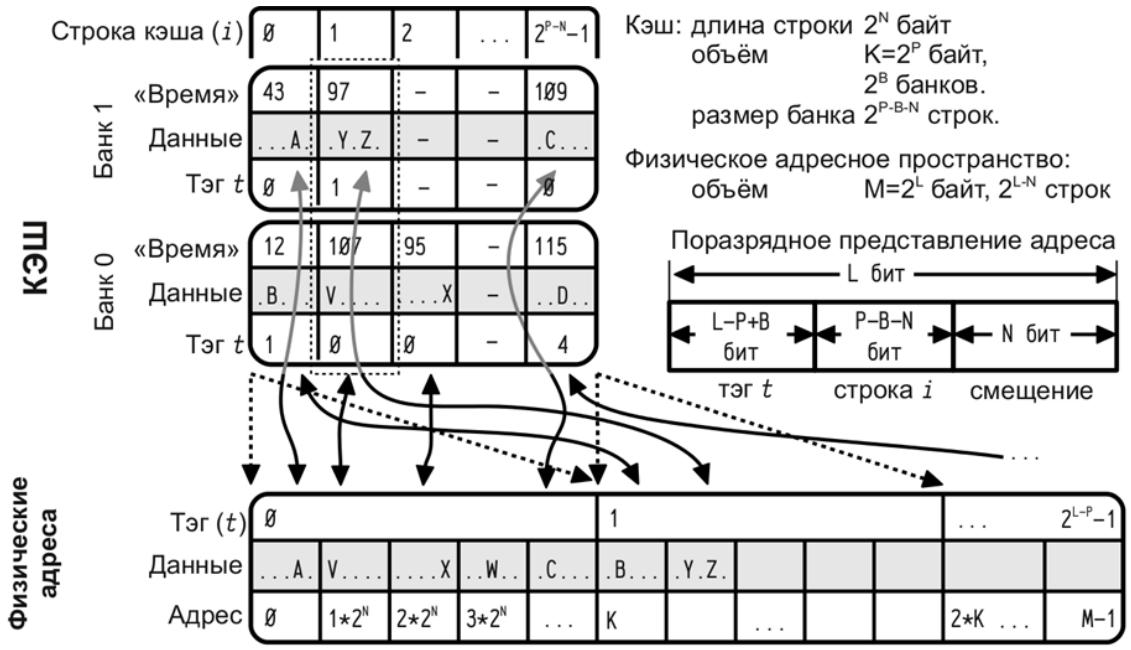


Кэш с прямым отображением (direct mapping cache) - данная строка ОЗУ может быть отображена в единственную строку кэша, но в каждую строку кэша может быть отображено много возможных строк ОЗУ.

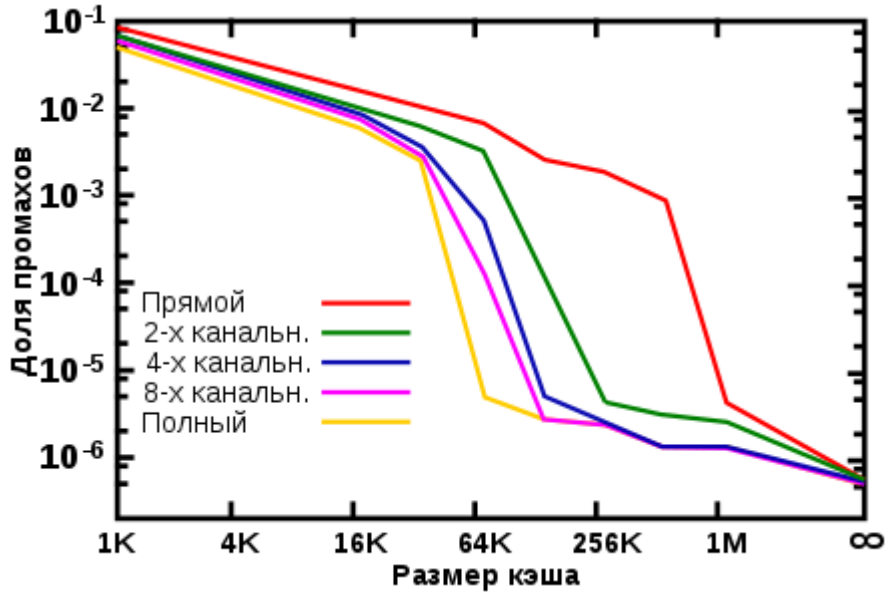


Множественно-ассоциативный кэш - кэш-память делится на несколько "банков", каждый из которых функционирует как *кэш с прямым отображением*[КЕД1] , таким образом строка памяти может быть отображена не в единственную возможную запись кэша (как было бы в случае прямого отображения), а столько раз, сколько доступно банков (в один из них); выбор банка осуществляется на основе алгоритма вытеснения (LRU – вспоминаем ОСИ).





При одинаковом объёме кэша схема с большей ассоциативностью будет наименее быстрой, но наиболее эффективной (после четырёх потоковой реализации прирост «удельной эффективности» на один поток растёт мало).



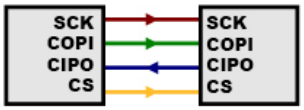
22. Поддержка операций ввода-вывода в фон Неймановских процессорах: система команд, механизм прерываний, механизм прямого доступа к памяти.

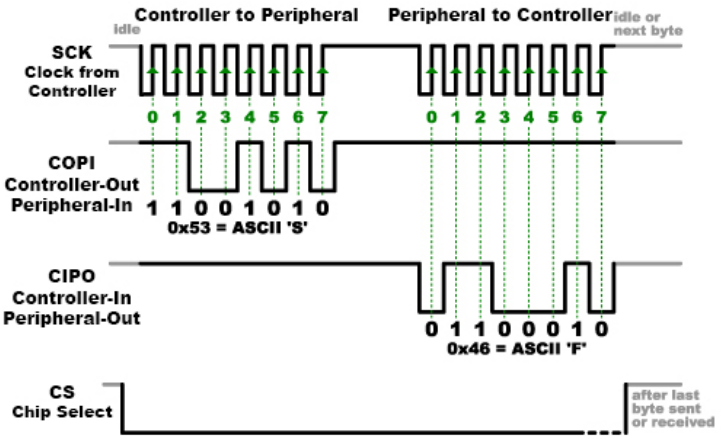
В данном контексте ввод-вывод это передача информации между процессором (CPU) и внешними устройствами (диски, мышки, экраны, GPIO и т.п.). Можно выделить три подхода:

1. ***Программно-управляемый ввод-вывод*** – все операции реализуются процессором, включая постоянное наблюдение за готовностью внешнего

устройства к передаче данных. Требует огромных временных затрат на взаимодействие с внешним устройством.
Процессор в цикле (polling) опрашивает состояние внешнего устройства и по наступлению необходимости начинает с ним взаимодействовать. К примеру наше устройство это Slave шины SPI реализованная через порты общего назначения (GPIO):

- регистрируем сигнал CS (Chip Select) который говорит о том что вот-вот начнётся передача данных, а значит опрос нужно начать производить чаще;
- регистрируем изменение сигнала SCLK (сигнал синхронизации)
- и далее, в зависимости от настроек SPI, фиксируем биты на линию данных (для ввода, COPI) в память побитно, а также выставляем наши данные на линию (для вывода, CIPO);
- завершаем работу с интерфейсом по установлению сигнала CS.





вот смотришь сюда, сразу ТКС вспоминается

Простейший пример: получение информации от кнопки (включая фильтрацию дребезга).

2. ***Ввод-вывод по прерыванию***. Снимает с процессора задачу постоянного наблюдения за внешним устройством и позволяет это реализовать по внешнему событию, к примеру готовность устройства к передаче данных или наступление заданного события (срабатывание таймера, нажатие на клавишу и т.п.). Непосредственно ввод-вывод реализуется по прежнему процессором.

   Получив прерывание, процессор незамедлительно прерывает текущий поток управления и передает управление по вектору прерывания, сохраняя состояние прерванного потока, а после завершения, возвращается к нему, восстанавливая состояние исходного.

В зависимости от источника возникновения сигнала прерывания делятся на:

- асинхронные, или внешние (аппаратные) – события от ВУ, могут произойти в произвольный момент относительно внутренней работы процессора
- синхронные, или внутренние – события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода, например, деление на ноль или переполнение стека.
- программные (частный случай внутреннего прерывания) – инициируются исполнением специальной инструкции в коде программы. Как правило, используются для обращения к функциям встроенного программного обеспечения (firmware), драйверов и операционной системы.

По принятию решения о необходимости обработки прерывания прерывания делятся на:

- Маскируемые прерывания (разработчику доступны регистры, конфигурация которых позволяет их игнорировать). К примеру: отключение прерываний в рамках выполнения критической секции кода.
- Немаскируемые прерывания (которые нельзя проигнорировать). К примеру: ошибка доступа к основной памяти или обработки сторожевого таймера (WatchDog таймер).

Приоритезация прерываний:

- относительные прерывания (может быть обработано немного позднее);
- абсолютное (необходимо прервать текущее прерывание и перейти к обработке нового немедленно).

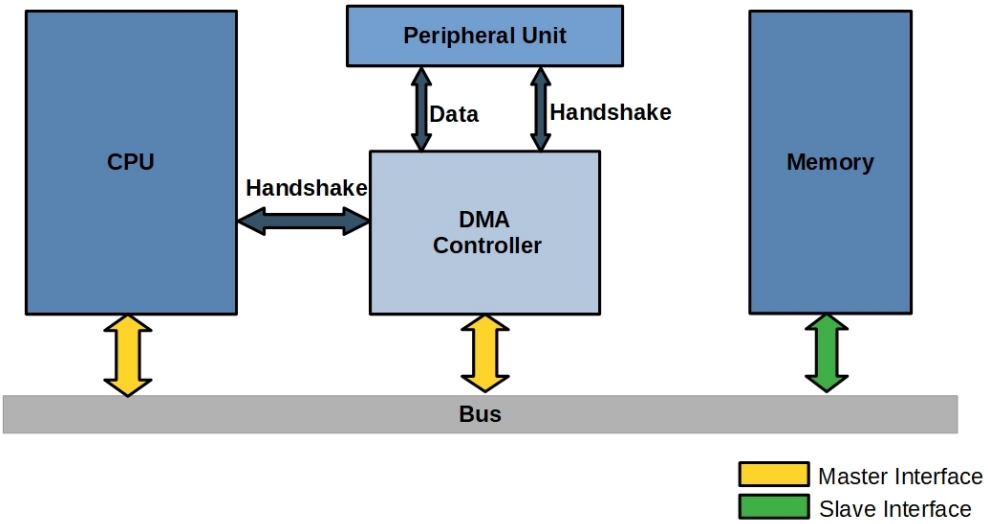
Прерывания могут срабатывать по разным видам событий, а именно:

- по фронту (о котором сигнализирует переход уровня сигнала на линии прерывания)
- по уровню сигнала
- по сообщению (message signaled), реализуется очередь сообщений о наступлении прерываний);
- по дверному звонку (doorbell), сигнализируется наступление прерывания, в то время как информация о нём сохраняется в условленном месте.

Данные задачи обычно решаются на аппаратном уровне (контроллер прерываний).

3. ***Channel I/O и прямой доступ к памяти*** (Direct Memory Access -- DMA). В случае, если необходимо передать большой объём данных, использование для этого процессора является нецелесообразным по причине его откровенной избыточности для этих задач. В таком случае могут применяться процессоры ввода-вывода и контроллеры прямого доступа к памяти, которые позволяют процессору задекларировать необходимость передачи данных (откуда, куда, сколько), после чего её сможет реализовать специализированное устройство, которое собственно и обеспечит передачу данных и уведомит процессор о

результатах через систему прерываний.



Выделяют два основных принципа работы:

- Third-party, где работа с DMA управляется полностью процессором, и любая передача данных должна инициализироваться им.
- Bus mastering, где работа с DMA управляется в том числе и со стороны устройств ввода-вывода, что позволяет инициализировать передачу данных без участия процессора.

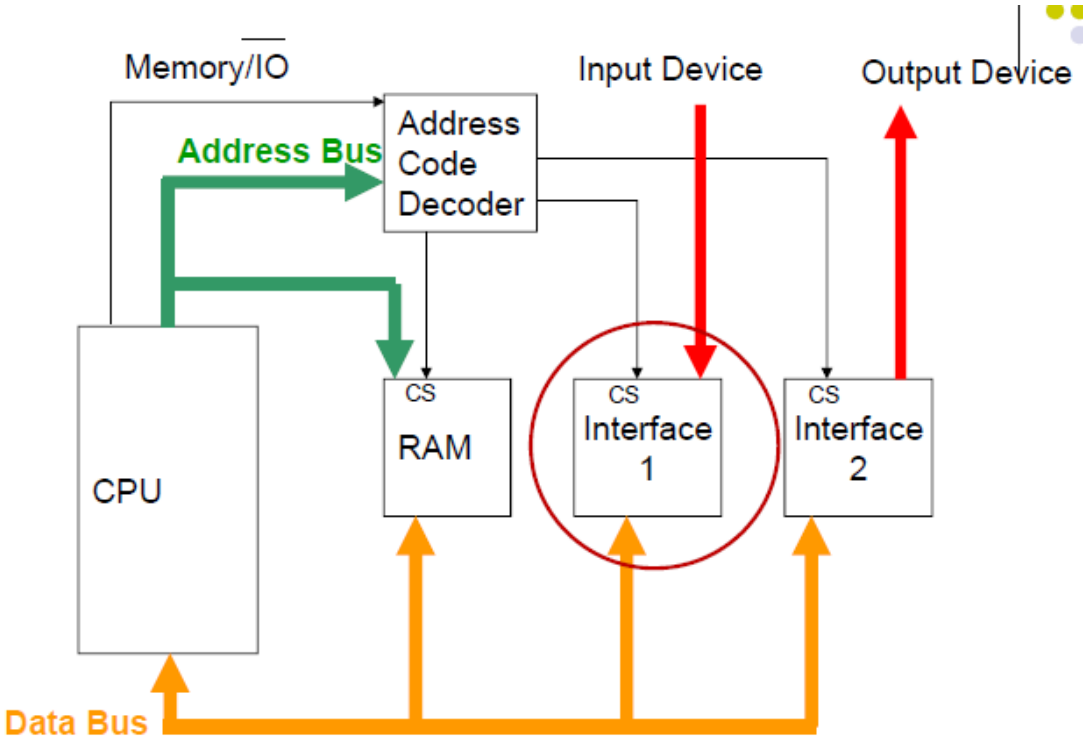
Так как в подавляющем большинстве случаев одновременная работа контроллера прямого доступа с памятью и процессора невозможна, режимы функционирования DMA по отношению к памяти:

- Пакетный режим (Burst Mode). Передача данных осуществляется единой операцией, которая не может быть прервана процессором.
- Циклический режим (Cycle stealing mode). Работа с памятью для процессора и устройств ввода-вывода имеет циклическую природу, где для каждого вида операций выделяется свой временной слот. Это позволяет избежать непредсказуемых задержек в работе процессора, но может снижать общую производительность системы.
- Прозрачный режим (Transparent Mode). Передача данных осуществляется в те моменты времени, когда процессор не взаимодействует с памятью.

Другим вариантом реализации ввода-вывода минуя центральный процессор являются каналы ввода-вывода или процессоры ввода вывода. Данные механизмы были широко распространены на заре вычислительной техники, а именно в мейнфреймах и позволяли процессору определить относительно полноценную программу для взаимодействия с внешним устройством ввода-вывода, где система команд адаптирована под задачи ввода-вывода (типичный пример: автоматическая конвертация формата хранения данных). Данный подход является более общим по сравнению с DMA, который следует рассматривать частным случаем.

Процессор может взаимодействовать с **вводом-выводом через память и через порты**.

При ***вводе-выводе через память*** часть адресного пространства резервируется для устройств ввода вывода, поэтому адресное пространство используется как для доступа к основной памяти, так и для доступа к устройствам ввода-вывода.



Ключевые **достоинства** данного подхода:

- Упрощение внутреннего устройства процессора.
- Вы можете работать с данными устройств ввода-вывода как с обычными данными
- Удобнее работа с функциями, оперирующими большими объёмами данных (например, чтение-запись на диск, в видеоадаптер). Нет необходимости "двойного переноса".
- Адресное пространство памяти, как правило, огромно. А значит возможность столкнуться с ограничениями специального адресного пространства для устройств ввода-вывода значительно ниже.

К **недостаткам** можно отнести:

- использование единой шины для ввода-вывода и доступа к памяти;
- неоднородность памяти, сложная конфигурация системы (включая инструментарий);
- конфликты с системой кеширования и параллелизмом уровня инструкций, так как процессор не может рассчитывать что память это память, к примеру:
    - запись данных в кеш не гарантирует, что они будут отправлены в устройство ввода-вывода;
    - последовательность записи данных в несколько командных регистров внешнего устройства может иметь принципиальное значений (см. барьеры памяти).

- устройства ввода-вывода должны работать с полным адресом (если 64 бита, то всеми 64 битами), чего не требуется с точки зрения ввода-вывода;
- возможно замедление работы системы в целом, так как ввод-вывод зачастую медленнее чем доступ к памяти.

При ***вводе-выводе через порты*** адресация устройств ввода-вывода производится независимо от адресации памяти, но для этого требуется наличие специально выделенных для этого инструкций.

Достоинства:

- Объем логики управления соответствует адресному пространству устройств ввода/вывода.
- На уровне ассемблера чётко видно, где реализуется доступ к памяти, а где к внешним устройствам.
- Адресное пространство однородно и не имеет "запретных мест".
- В целом система проще, так как ввод-вывод не приходится интегрировать в систему кешей, а операции ввода/вывода могут обрабатываться особым образом на уровне конвейеров и спекулятивных вычислений.

Недостатки:

- Усложнение системы команд и процессора, реализующую его.
- Данные ввода-вывода становятся данными второго сорта.

23. Операции ввода-вывода с точки зрения параллелизма уровня задач. Механизм прерываний. Как реализуется параллелизм?

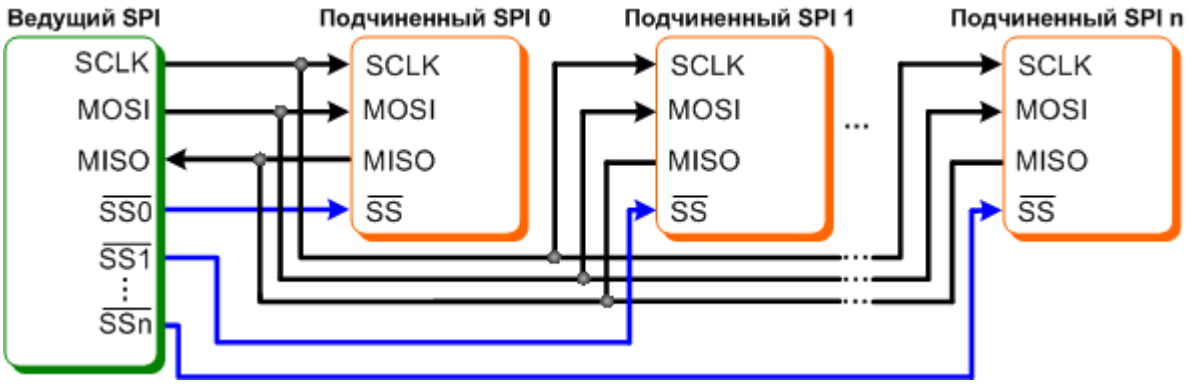
- Операции ввода-вывода с точки зрения параллелизма уровня задач.

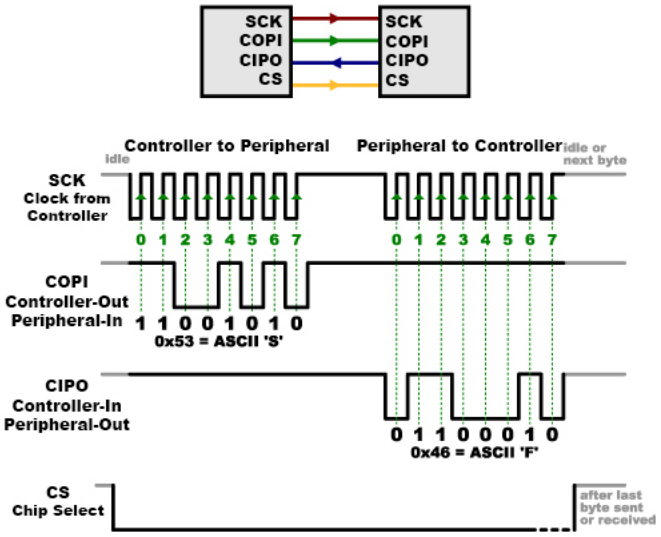
Программно-управляемый ввод-вывод

Работа с вводом-выводом при программном управлении реализуется следующим образом: процессор в цикле (polling) опрашивает состояние внешнего устройства и по наступлению необходимости начинает с ним взаимодействовать. К примеру наше устройство это Slave шины SPI реализованная через порты общего назначения (GPIO):

- регистрируем сигнал CS (Chip Select) который говорит о том что вот-вот начнётся передача данных, а значит опрос нужно начать производить чаще;
- регистрируем изменение сигнала SCLK (сигнал синхронизации), причем частота опроса должна быть как 2 раза выше частоты передачи данных (в теории по теореме Котельникова) или больше (на практике), но и не в порядки (регистрация электрического дребезга);
- и далее, в зависимости от настроек SPI, фиксируем биты на линию данных (Master Out Slave In, MOSI или Controller Out Peripheral In, COPI) в память побитно на положительный или отрицательный фронт SCLK, а также выставляем наши данные на линию (Master In Slave Out, MISO или Controller In Peripheral Out, CIPO);
- завершаем работу с интерфейсом по установлению сигнала CS.

Примечание: почему сигнал CS устанавливает по нулевому значению? Вероятнее всего это связано с традицией использования в качестве сигнала CS сигнала RESET периферийного устройства, то есть переход CS в отрицательной состояние приводит к его включению.





Как можно видеть, это требует огромного количества работы в реальном времени со стороны процессора, в тоже время реальные системы имеют множество интерфейсов ввода-вывода, а значит количество работы будет увеличиваться кратно им. Код при этом может выглядеть следующим образом или требует приоритизация устройств на уровне аппаратуры:

```
if (device[0].flag)
    device[0].service();
else if (device[1].flag)
    device[1].service();
```

Таким образом процессор не только начинает тратить свои ресурсы на обеспечение ввода-вывода (включая высокие накладные расходы и необходимость делать это "параллельно" основной работе), но и принципиально ограничивает скорость передачи данных (см. высокоскоростные интерфейсы со скоростью передачи информации десятков гигабит).

- Механизм прерываний.

В основе архитектуры фон Неймана лежат принципы последовательного исполнения команд и возможности условного перехода. Именно они позволили сделать программирование данного типа процессоров относительно простым и гибким процессом.

Обратной стороной данного эффекта стала невозможность изменить поток управления извне его, процессор будет "переваривать" последовательность команд без перерыва столько, сколько посчитает необходимым.

К сожалению, данное поведение не всегда является удобным, так как:

- не позволяет эффективно работать с внешними событиями и устройствами ввода-вывода (мы вынуждены выполнять данные операции по опросу (polling), а не по появлению новой информации);
- не позволяет обрабатывать нештатные ситуации иначе, чем при помощи кодов ошибок или флагов (что накладывает высокие требования на дисциплину разработки программного обеспечения);
- не позволяет независимо исполнять несколько потоков команд (вытесняющая многозадачность).

Решает эту проблему такой механизм как "прерывание" (interrupt), который позволяет сигнализировать процессору о том, что текущий поток управления должен:

- быть незамедлительно прерван,
- его состояние сохранено,
- а управление передано по указанному вектору прерывания (в зависимости от источника прерывания может быть необходимым вызвать разные его реализации).
- По завершению выполнения обработчика прерывания управление должно быть возвращено исходному потоку управления с восстановлением его состояния.

В зависимости от источника возникновения сигнала прерывания делятся на:

- асинхронные, или внешние (аппаратные) -- события, которые исходят от внешних аппаратных устройств (например, периферийных устройств) и могут произойти в любой произвольный момент относительно внутренней работы процессора: сигнал от таймера, сетевой карты или дискового накопителя, нажатие клавиш клавиатуры, движение мыши.
- синхронные, или внутренние -- события в самом процессоре как результат нарушения каких-то условий при исполнении машинного кода: деление на ноль или переполнение стека, обращение к недопустимым адресам памяти или недопустимый код операции;
- программные (частный случай внутреннего прерывания) -- инициируются исполнением специальной инструкции в коде программы. Программные прерывания, как правило, используются для обращения к функциям встроенного программного обеспечения (firmware), драйверов и операционной системы.

Обработка прерывания с точки зрения процессора довольно сложный процесс, который включает в себя:

- Принятие решения о необходимости обработки прерывания. Прерывания делятся на:
    - Маскируемые прерывания, для которых разработчику доступны регистры, конфигурация которых позволяет их игнорировать. К примеру: отключение прерываний в рамках выполнения критической секции кода.
    - Немаскируемые прерывания, которые нельзя проигнорировать. К примеру: ошибка доступа к основной памяти или обработки сторожевого таймера (WatchDog таймер).

- Приоритезация прерываний. В случае если во время обработки прерывания приходит другое, то они делятся на:
    - относительные прерывания (может быть обработано немного позднее);
    - абсолютное (необходимо прервать текущее прерывание и перейти к обработке нового немедленно).
- Прерывания могут срабатывать по разным видам событий, а именно:
    - по фронту (положительному -- изменение сигнала с 0 на 1 и отрицательному -- наоборот);
    - по уровню сигнала, в таком случае после обработки прерывания необходимо "сбросить" прерывание во избежании его циклической обработки. Позволяют объединить множество сигналов прерываний через логические или (что порождает неидентифицируемые (Spurious interrupts) прерывания, что конечно плохо, но может быть целесообразно);
    - по сообщению (message signaled), реализуется очередь сообщений о наступлении прерываний);
    - по дверному звонку (doorbell), сигнализируется наступление прерывания, в то время как информация о нём сохраняется в условленном месте.

Данные задачи обычно решаются на аппаратном уровне (контроллер прерываний). Возможно и "потеря прерываний", если количество заявок на прерывание от внешних устройств больше, чем возможно обработать. Чтобы минимизировать такое стечение обстоятельств, обработчик прерываний должен быть по возможности компактным и быстрым, поэтому часто применяется практика отложенных вызовов: вместо того чтобы выполнить сложное взаимодействие с внешним устройством в рамках обработки прерывания происходит фиксация отложенной процедуры, которая в свою очередь может быть выполнена в обычном режиме функционирования.

Примеры обработки ввода вывода с использованием системы прерываний. Задача: передача данных интерфейса SPI. Обратите внимание:

- Необходимо прерывание от двух сигналов: CS и SCLK. Причём второе прерывание необходимо маскировать, если CS положительный (см. схему подключения).
- Возможно обойтись и одним прерыванием, полностью реализовав обработку передачи данных при её начале, но с высокой вероятностью это будет нецелесообразно.
- Целесообразным выглядит следующий подход: внутри прерывания работа производится с буфером (и чтение, и запись), при передаче данных (CS отрицательный) доступ в буфер запрещён. По завершению передачи данных основной поток команд получает доступ к полученным в буфере данным в режиме опроса или по прерыванию.

Оба описанных выше метода работы с портами ввода-вывода требуют непосредственной обработки со стороны процессора, а значит данные будут проходить через процессор, что не всегда является целесообразным использованием ресурсов.

- Как реализуется параллелизм?

Повышение скорости вычислений может достигаться за счет ускорения исполнения операций / преобразований данных, которое в свою очередь может достигаться за счёт **закона масштабирования Деннарда** -- уменьшая размеры транзистора и повышая тактовую частоту процессора, мы можем легко повышать его производительность. К сожалению, как и закон Мура, данный закон имеет свою области применимости и срок годности, так как наталкивается на:

- Ограничения современных технологий (сложность и стоимость производства).
- Физические ограничения на возможности уменьшения транзисторов (размеры атомов и эффекты связанные с уменьшением транзисторов).

Другой источник роста скорости вычислений лежит в том, чтобы заставить как можно больше операций / преобразований данных происходить в один момент времени или **параллелизм.**

Параллелизм можно рассматривать с двух точек зрения:

1. Сверху вниз, когда прикладные функции и задачи описываются языковыми средствами описания вычислительного процесса и используют набор "атомарных" операций (в данном случае скорее не требующих разбиения, чем неразделимых) и определяют какие из них могут быть выполнены параллельно (на основании модели вычислений языка, а не той информации которую мы можем вывести из описания конкретного алгоритма).
2. Снизу вверх, организация физических процессов в вычислительный процесс, который в свою очередь реализует прикладной процесс. И чем больше физических процессов мы сможем провернуть "параллельно" для одного процесса, тем лучше.

24. Параллелизм уровня битов в рамках комбинационных схем и фон Неймановских процессоров.

Параллелизм уровня битов (Bit-level Parallelism) достигается за счет увеличения “ширины” комбинационных схем, то есть за счет увеличения количества входных сигналов схемы.

В контексте фон Неймановского процессора, это ширина машинного слова – фрагмент данных фиксированного размера, обрабатываемый как единое целое с помощью набора команд или аппаратного обеспечения процессора

Пример оптимизации за счёт параллелизма уровня битов для сложения бинарных чисел:

- машинное слово: 8 бит, данные: 16 бит, в таком случае сложение производится несколько шагов:

    1. сложение младших битов, фиксация бита переполнения и сохранение результата в память;
    2. сложение старших битов, добавление бита переполнения (если бы установлен) и сохранение результата в память;
- машинное слово: 16 бит, данные: 16 бит, сложение осуществляется в рамках большой комбинационной схемы за один такт (примечание: комбинационная схема в 16 бит будет работать на меньшей частоте чем в 8 бит, но в данном случае это разница несопоставима с необходимостью нескольких тактов).

Параллелизм уровня битов имеет серьёзные ограничения так как “простые типы данных” не имеет практического смысла наращивать (к примеру, int64 более чем достаточно для подавляющего числа задач, а значит переход на int128 приведёт к тому, что старшие биты будут в основном “греть воздух”, а в тех редких случаях где они действительно нужны не факт, что их будет достаточно).

Использовать же широкое машинное слово для составных данных также затруднено в случае процессоров общего назначения из-за их многообразия (составные данные зависят от прикладной задачи) и необходимости их поддержки на уровне системы команд (см. CISC vs RISC), без которой это теряет смысл.

25. Параллелизм уровня инструкций. Принцип конвейерного и суперскалярного исполнения инструкций.

Параллелизм уровня инструкций

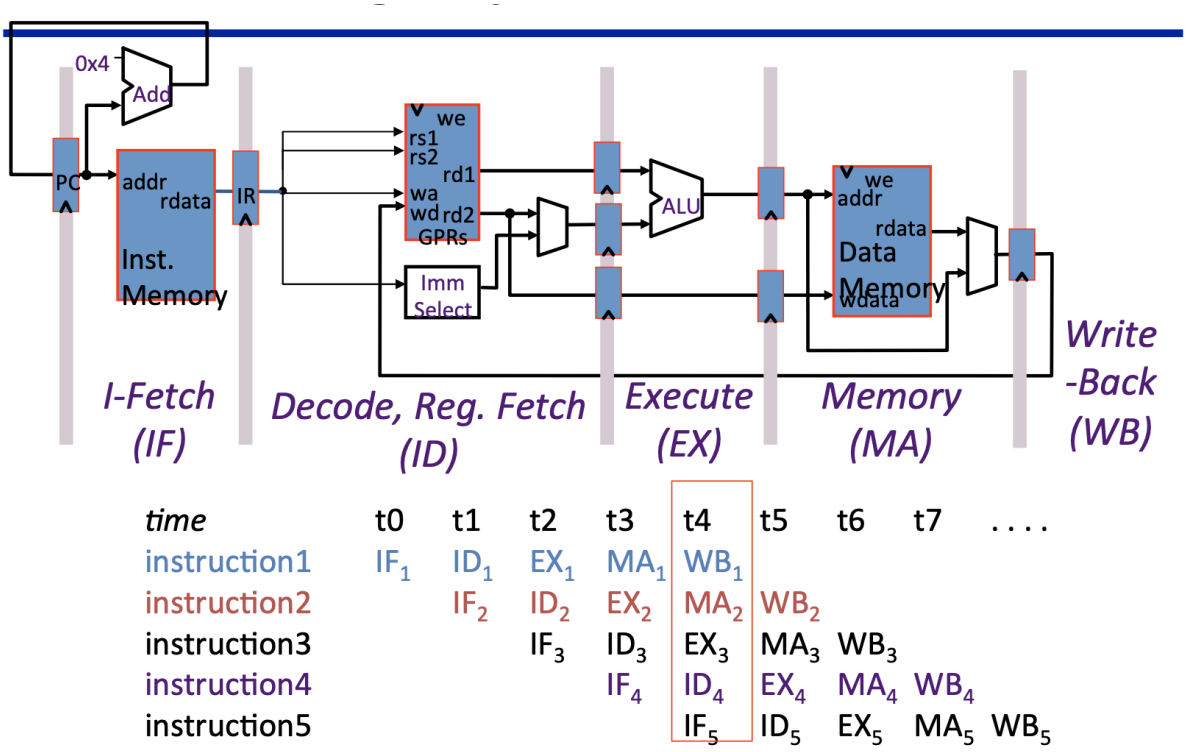
Одним из ключевых элементом архитектуры процессора фон Неймана является представление алгоритма в виде последовательности команд, что говорит о том, что инструкции не могут быть выполнены одновременно. На самом деле это не так и параллелизм может быть найден за счёт:

- выполнение команды -- длительный и сложный процесс, значит он может быть разделён на этапы и запущен на конвейере;
- выполнение основной задачи команды (к примеру, деления) может требовать большого количества времени, а значит в процессоре будут присутствовать простаивающие элементы, которые можно загрузить другими задачами командами;
- изменение порядка выполнения команд, зачастую, не изменяет их результата.

Конвейеризированное исполнение команд

Данный тип параллелизма аналогичен изобретению Генри Форда, который предложил реорганизовать производство с цехового принципа (каждый цех выполняет полный цикл по отношению к заготовке) на конвейерный (каждый пост конвейера выполняет работы параллельно над своей заготовкой). Конвейерная организация работ позволяет за счёт реорганизации рабочего пространства и увеличения количества рабочих (пропорционально количеству постов, но не путём простого умножения) повысить производительность. С точки зрения цифровой схемотехники, данный способ

эксплуатирует возможность параллельной и независимой работы комбинационных схем разделённых регистрами, сдвиг же конвейера реализуется по глобальному тактовому сигналу.



time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

Преимущества: повышение производительности и уровня утилизации вычислительных ресурсов.

Потенциальные недостатки конвейера:

- снижение скорости исполнения отдельной команды (за счёт плохой балансировки стадий и дополнительных регистров);
- не все операции могут пройти стадию конвейера за один машинный цикл (загрузка ресурсов);
- необходимость разрешения конфликтов (загрузка ресурсов);
- непредсказуемое время исполнения;
- противоречие с фон Неймановской архитектурой (принцип единой памяти);
- уязвимости связанные с доступом через "косвенные каналы".

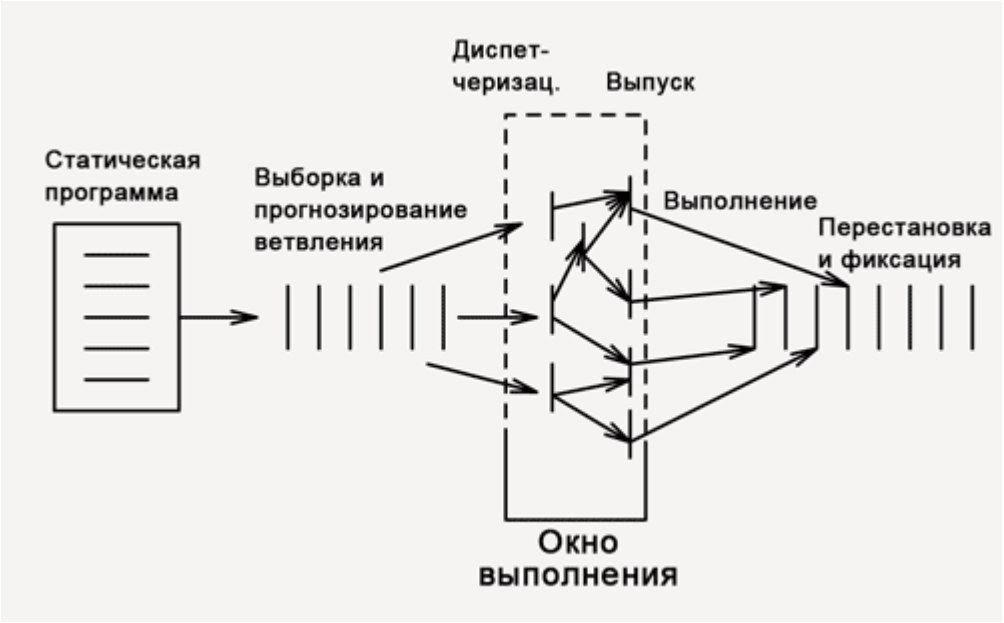
Краткое содержание: Каждая команда так или иначе состоит из нескольких основных этапов. Так как каждый из таких этапов выполняет определенную логику, работает с определенными регистрами, разные этапы конвейера можно выполнить параллельно. К примеру, когда процессор ищет в памяти аргумент для инструкции, АЛУ свободен, в этот момента на нем можно произвести операцию. Поэтому внутри процессора команды отрабатываются на так называемом конвейере. Стадии конвейера связаны между собой, а также выровнены по времени исполнения.

Суперскалярные архитектуры

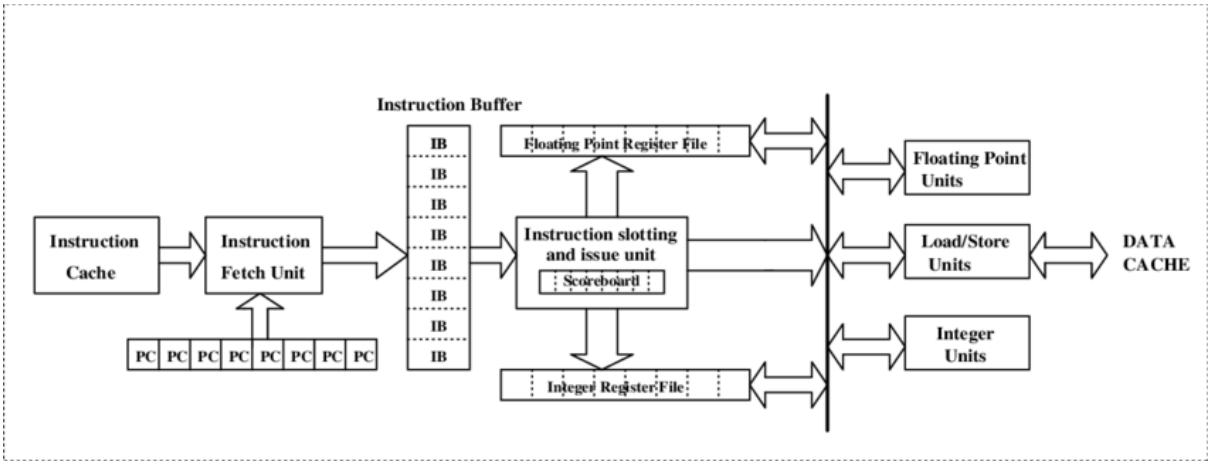
Термин "суперскалярный" произошёл от понятия "скалярной величины" -- величины, которая может быть представлена числом (целочисленным или с плавающей точкой). Отсюда:

- скалярная операция -- операция над числом (если один операнд) или числами (несколько операндов);
- векторная операция -- операция, для которой в качестве операнда и результата могут выступать массивы, за счёт чего нет необходимости на каждый набор увеличивать счётчик, декодировать команду и т.п.

В тоже время крупный процессор может иметь большое количество устройств обработки (АЛУ), в том числе и разных типов. Подход, позволяющий одновременно исполнять несколько скалярных и/или векторных операций называется суперскалярным и показан на схеме.



В нормальном состоянии (в том числе и конвейеризированный) процессор выполняет каждую команду одну за другой, что приводит к неоптимальной загрузке устройств обработки (АЛУ). Её можно улучшить за счёт параллельного исполнения операций если есть доступные блоки обработки и команды не зависят друг от друга по данным. Для этого необходимо в процессе работы "на лету" находить такие ситуации, для чего формируются специальные узлы процессора. Визуализация приведена на рисунке.



Важной особенностью суперскалярных процессоров является то, что с точки зрения системы команд они идентичны простым процессорам и вся диспетчеризация происходит без ведома пользователя.

Преимущества: повышение производительности и уровня утилизации вычислительных ресурсов.

Потенциальные недостатки конвейера:

26. Конвейеризированное исполнение команд. Стадии конвейера. Виды конфликтов и их примеры.

Конвейеризированное исполнение команд

Конвейеризировнное исполнение команд - способ организации вычислений, используемый в современных процессорах и контроллерах с целью повышения их производительности (увеличения числа инструкций, выполняемых в единицу времени).

Для организации конвейерной работы процессора необходимо:

- Выделить стадии выполнения команд;
- Организовать внутренние структуры процессора таким образом, чтобы:
    - У процессора был входной (поступают команды) и выходной конец (готовые команды "покидают" конвейер процессора);
    - Структура процессора должна соответствовать стадиям выполнения команд, каждый сегмент должен быть связан с последующим через регистровый интерфейс, причем длительность комбинационных схем каждого сегмента должна быть по возможности одинаковой;
    - Все сегменты процессора должны управляться от одного тактового сигнала;

- Настроить управление конвейером таким образом, чтобы загружать в него команды каждый такт;
- Разрешить конфликты и противоречия, вызванные частично параллельным исполнением команд.

Стадии конвейера

Рассмотрим вариант с классическим конвейером RISC процессора. Он имеет пятиступенчатый конвейер.

1. Instruction Fetch (Извлечение инструкций). Процессор считывает инструкцию по адресу в памяти, значение которого присутствует в счетчике программ.
2. Instruction Decode (Декодирование инструкций). Команда декодируется и осуществляет доступ к регистровому файлу для получения значений из регистров, используемых в инструкции.
3. Instruction Execute (Выполнение инструкции). На этом этапе выполняются операции. Причём операции данного этапа можно разделить на:
    - Register-Register Operation (Single-cycle latency): Сложение, вычитание, сравнение и логические операции. На этапе выполнения два аргумента были переданы в простой АЛУ, который сгенерировал результат к концу этапа выполнения.
    - Memory Reference (Two-cycle latency): Получение данных из памяти и их загрузка в регистры с учетом формирования физического адреса в памяти выполняемого АЛУ.
    - Multi-cycle Instructions (Many cycle latency): Целочисленное умножение и деление и все операции с плавающей запятой. Могут реализовываться в рамках сопроцессора, сохраняющего результат в отдельные регистры, чтобы освободить конвейер. Длительность выполнения команды зависит от конкретных данных.
4. Memory Access (Доступ к памяти). На этом этапе операнды памяти считываются и записываются в виде/в память, присутствующую в инструкции.
5. Write Back (Написать ответ). На этом этапе вычисленное/полученное значение записывается обратно в регистр, присутствующий в инструкциях.

time	t0	t1	t2	t3	t4	t5	t6	t7	. . . .
instruction1	IF_1	ID_1	EX_1	MA_1	WB_1				
instruction2		IF_2	ID_2	EX_2	MA_2	WB_2			
instruction3			IF_3	ID_3	EX_3	MA_3	WB_3		
instruction4				IF_4	ID_4	EX_4	MA_4	WB_4	
instruction5					IF_5	ID_5	EX_5	MA_5	WB_5

Количество стадий конвейера может варьироваться в широких пределах. Есть как простые варианты процессоров PIC (2 стадии), так и более сложные (Pentium 4, 20-31).

Виды конфликтов и их примеры

Очевидно, что если бы конвейеризация вычислений реализовывалась без каких-либо трудностей, тогда это было бы универсальным решением, позволяющим повышать производительность за счёт повышения частоты (и энергопотребления). Основную сложность для реализации конвейерных вычислений представляют возникающие при его работе конфликты между стадиями конвейера. Выделяют три группы конфликтов:

1. Структурные конфликты

2. Конфликты по данным
3. Конфликты по управлению

Структурный конфликт
Структурные конфликты возникают из-за конфликтов ресурсов, когда аппаратура не может поддерживать все возможные комбинации одновременно выполняемых команд. Если какая-то комбинация команд не может быть поддержана, то говорят, что процессор имеет структурный конфликт.

Inst. / Tick	1	2	3	4	5
I1	IF(Mem)	ID	EX	! Mem	
I2		IF(Mem)	ID	EX	
I3			IF(Mem)	ID	EX
I4				! IF(Mem)	ID

восклицательным знаком отмечен конфликт.

Тут показано, что во время выполнения инструкций I4 идет обращение к памяти в то время, когда инструкция I1 достигла обращения к памяти для взятия операндов.
Решение данной проблемы - приостановить конвейер на один такт (вставка пузырька)

Конфликт по данным
Конфликты по данным возникают, когда зависимость команды от результатов предыдущей проявляется при совмещении команд в конвейере.

Есть 3 типа конфликтов по данным: RAW (read after write), WAR (write after read), WAW (write after write).
Рассмотрим две инструкции i1 и i2 , причем i1 стоит перед i2 в программном порядке.

RAW:
i1. **R2** <- R5 + R3
i2. R4 <- **R2** + R3
Первая инструкция вычисляет значение, которое будет сохранено в регистре R2 , а вторая собирается использовать это значение для вычисления результата для регистра R4 . Однако в конвейере , когда операнды выбираются для второй операции, результаты первой еще не сохранены, и, следовательно, возникает зависимость данных.

WAR:
i1. R4 <- R1 + **R5**
i2. **R5** <- R1 + R2
(i2 пытается записать место назначения до того, как оно будет прочитано i1)
Опасность записи после чтения (WAR) представляет собой проблему с параллельным выполнением.
i2 может завершиться раньше , чем i1 (т. е. при одновременном выполнении),

необходимо гарантировать, что результат регистра R5 не будет сохранен до того , как i1 получит возможность получить операнды.

WAW:
i1. **R2** <- R4 + R7
i2. **R2** <- R1 + R3
(i2 пытается записать операнд до того, как он будет записан i1) В среде параллельного выполнения может возникнуть опасность записи после записи .

Механизмы разрешения конфликтов:

- вставка пузыря в конвейер - это пропуск нескольких тактов процессора;
- исполнения не по порядку, out-of-order -- изменение порядка выполнения команд с учётом конвейерного исполнения на уровне компиляции или процессора (см. далее, суперскалярные архитектуры);
- проброс операндов между стадиями процессора -- для выполнения операций без задержек данные могут быть переданы напрямую между стадиями конвейера, не дожидаясь сохранения в регистровый файл или память;
- переименования регистров -- в случае если зависимость по данным ложная (WAW который может быть оптимизирован), промежуточный результат одной из записей может быть переназначен на другой регистр на лету.

Конфликты по управлению
При выполнении условного перехода, может измениться PC (счетчик команд) и если не принять нужных мер то произойдет остановка конвейера на много тактов, пока не будет вычислено условие перехода (переход определится).
Сброс конвейера -- крайне дорогостоящая операция, частое наступление которой может значительно сократить эффект от использования конвейера, особенно если в нём много стадий.

Решение этой проблемы лежит в области предсказания будущих переходов.
Для безусловных -- ранее извлечение информации об условном переходе и проброс её на стадию выборки инструкции. Для условных -- спекулятивные вычисления, которые выполняются на основании наиболее вероятного перехода.

Преимущества и недостатки конвейера

Преимущества**:** повышение производительности и уровня утилизации вычислительных ресурсов.

Потенциальные недостатки конвейера:

- снижение скорости исполнения отдельной команды (за счёт плохой балансировки стадий и дополнительных регистров);
- не все операции могут пройти стадию конвейера за один машинный цикл (загрузка ресурсов);
- необходимость разрешения конфликтов (загрузка ресурсов);
- непредсказуемое время исполнения;
- противоречие с фон Неймановской архитектурой (принцип единой памяти);
- уязвимости связанные с доступом через "косвенные каналы".

Множество схем включают в себя конвейеры в 7, 10 или даже 20 уровней (как, например, в процессоре Pentium 4). Поздние ядра Pentium 4 с кодовыми именами Prescott и Cedar Mill (и их Pentium D-производные) имеют 31-уровневый конвейер.

Еще оно используется в суперкомпьютерах, и там программы специально пишутся так, чтобы как можно реже использовать условные операторы, поэтому очень длинные конвейеры весьма позитивно скажутся на общей скорости вычислений, так как они проектируются так, чтобы уменьшить CPI (количество тактов на инструкцию).

27. Параллелизм уровня задач. Закон Амдала. Способы реализации. Механизмы синхронизации.

Параллелизм уровня задач подразумевает параллельное выполнения нескольких потоков команд, которые определены разработчиком программного обеспечения. Другими словами: он не является прозрачным для программиста, а значит не может быть рассмотрен без учета средств программирования.

Центральное место занимает вопрос о том, как происходит переключение между потоками команд, влияние этого на разработку ПО и синхронизацию потоков исполнения.

Кооперативная многозадачность (Cooperative multitasking)

> Тип многозадачности, при котором следующая задача выполняется только после того, как текущая задача явно объявит себя готовой отдать процессорное время другим задачам.
>
> При простом переключении активная программа получает все процессорное время, а фоновые приложения полностью замораживаются. При кооперативной многозадачности приложение может захватить фактически столько процессорного времени, сколько оно считает нужным. Все приложения делят процессорное время, периодически передавая управление следующей задаче.

Реализация кооперативной многозадачности может осуществляться разными способами:

- В рамках операционной системы: предоставляется набор специализированных системных вызовов, которые позволяют сообщить ОС о том, что поток команд может быть приостановлен.
- На уровне виртуальных машин и/или компилятора. Такие конструкции языков программирования как yield, async/await и т.п.
- В ручном режиме, когда многозадачность реализуется программистом самостоятельно в рамках его приложения, что может быть реализовано или имитацией многопоточности (пример будет ниже), или решениями построенными вокруг callback-ов и event-loop-а.

Может использоваться для:

- Решения проблем связанных с медленным вводом-выводом. Примером этого является работа вычислительной системы в пакетном режиме. Пакетный режим подразумевает, что программы знают когда им остановиться и отдать управление в другой процесс и как правило это приурочено к задачам ввода-вывода.
- Для оптимизации систем требующих частого переключения между задачами или большого количества потоков команд. Использование кооперативной многозадачности в данном случае позволяет значительно сократить затраты на хранение и смену контекстов. Как правило такой режим многозадачности реализуется на уровне виртуальной машины и/или приложения/сервиса. Пример: nginx и apache, node.js (Event-loop).
- В простых встроенных системах, где нет возможности использовать ОС или это видится не целесообразным. В таком случае многопоточность может реализовываться "в ручном режиме" (ниже приведён пример).

Преимущества кооперативной многозадачности:

- отсутствие необходимости защищать все разделяемые структуры данных объектами типа критических секций и мьютексов, что упрощает программирование и скорость вычислений (примечание: сохраняются классические сложности работы с изменяемым состоянием);
- контроль за доступом к ресурсам процессора со стороны разработчика.

Недостатки:

- Небезопасность и непредсказуемость отдельных потоков команд. В случае если процесс не отдаёт управление в течении долгого времени это может быть обусловлено как "особенностями реализации", так и "ошибкой программирования". В любом случае это вносит непредсказуемость в поведение компьютера в целом.
- Высокая сложность реализации интерактивных приложений и балансировки ресурсов между большим количеством потоков команд, особенно в случае если программы разработаны независимо друг от друга.

Вытесняющая многозадачность (Preemptive multitasking)

Вид многозадачности, в котором операционная система сама передает управление от одной выполняемой программы другой в случае завершения операций ввода-вывода, возникновения событий в аппаратуре компьютера, истечения таймеров и квантов времени, или же поступлений тех или иных сигналов от одной программы к другой. В этом виде многозадачности процессор может быть переключен с исполнения одной программы на исполнение другой без всякого пожелания первой программы и буквально между любыми двумя инструкциями в её коде. Распределение процессорного времени осуществляется планировщиком процессов. К тому же каждой задаче может быть назначен пользователем или самой операционной системой определенный приоритет, что обеспечивает гибкое управление распределением процессорного времени между задачами (например, можно снизить приоритет ресурсоемкой программе, снизив тем

самым скорость её работы, но повысив производительность фоновых процессов). Этот вид многозадачности обеспечивает более быстрый отклик на действия пользователя.

Ключевые элементы для реализации данного метода многозадачности:

- поддержка со стороны процессора механизмами сохранения состояния процессора в память с целью последующего его возобновления;
- поддержка механизмов прерывания, позволяющих осуществить переключение процессов независимо от их желания, а также защиты отдельных операций или их последовательностей от прерывания;
- поддержка со стороны операционной системы, а именно: планировщик, который должен определять когда один поток команд будет прерван и какой поток команд его заместит.

Основным преимуществом вытесняющей многозадачности является относительная защищенность процессов и ОС друг от друга с точки зрения захвата ресурсов процессора. Также, вытесняющая многозадачность позволяет реализовать системы работающие более интерактивно, чем в случае кооперативной (реакция на внешнее событие может наступить практически мгновенно: вызов прерывания, информирование планировщика, переключение на связанный с событием процесс).

Ключевым недостатком данного вида многозадачности является как раз её главная особенность:

- Непредсказуемость момента переключения процесса, что требует решения проблемы синхронизации процессов. На сегодня, базовым инструментом синхронизации процессов является модель потоков (Thread) и блокировок, обладающая серьезными принципиальными ограничениями. Подробнее: The Problem with Threads.
- В "наивной реализации": незащищенность задач друг от друга с точки зрения данных. В этой области сегодня есть огромное количество механизмов защиты процессов друг от друга.
- Высокая "тяжесть" и "негибкость" работы с несколькими потоками команд на уровне процессора, на решение которой направлены зелёные или легкие потоки реализуемые на уровне виртуальных машин. К примеру:
    - язык программирования Go и go-рутины, переключение между которыми идёт за счёт искусственно вставленных в машинный код точек переключения контекстов (многозадачность является кооперативной, но её реализация скрыта от разработчика).
    - язык программирования Erlang и виртуальная машина Beam, в которой интерпретатор байт-кода имеет возможность "отсчитать" требуемое количество инструкций для выполнения (подробнее: Erlang Scheduler: what does it do?).

Закон Амдала: *ускорение программы с помощью параллельных вычислений на нескольких процессорах ограничено размером последовательной части программы*. Например, если можно распараллелить 95% программы, то теоретически

максимальное ускорение составит максимум 20×, невзирая на то, сколько процессоров используется.

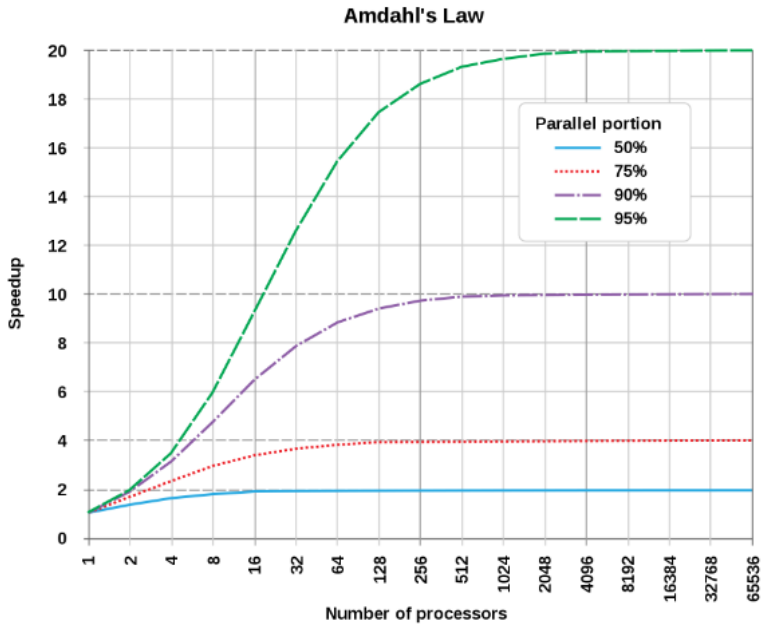


Figure 18: Amdahl's law

28. Проблема детерминизма в современном параллельном программировании. Примеры. Механизмы обеспечения детерминизма.

Используя последовательные программы, выполнение кода идет по счастливому пути предсказуемости и детерминизма. И наоборот, многопоточное программирование требует приверженности и усилий для достижения правильности.

Детерминизм является основополагающим требованием для создания программного обеспечения, поскольку часто предполагается, что компьютерные программы возвращают одинаковые результаты от одного запуска к другому. Но это свойство трудно решить параллельно. Внешние обстоятельства, такие как планировщик операционной системы или когерентность кеша, могут влиять на время выполнения и, следовательно, порядок доступа для двух или более потоков и изменять одну и ту же ячейку памяти. Этот вариант времени может повлиять на результат программы.

Пример:

Два потока находятся в состоянии гонки.

Пусть есть x = 1.

Также есть Поток 1 и Поток 2.

Первый поток: x + 5

Второй поток: x * 3

Запускаем программу..

// В зависимости от того, какой поток первым займет переменную, такой результат и получится. Если не использовать специальные средства, результат может отличаться от запуска к запуску программы.

// Дополнительно стоит помнить, что запуск одних и тех же программ на системах разных архитектур также может привести к разному результату.

Механизмы обеспечения:

1. С программными способами мы встречались ранее: запретить потокам кэшировать переменную, создавать очередь из потоков и ТД.

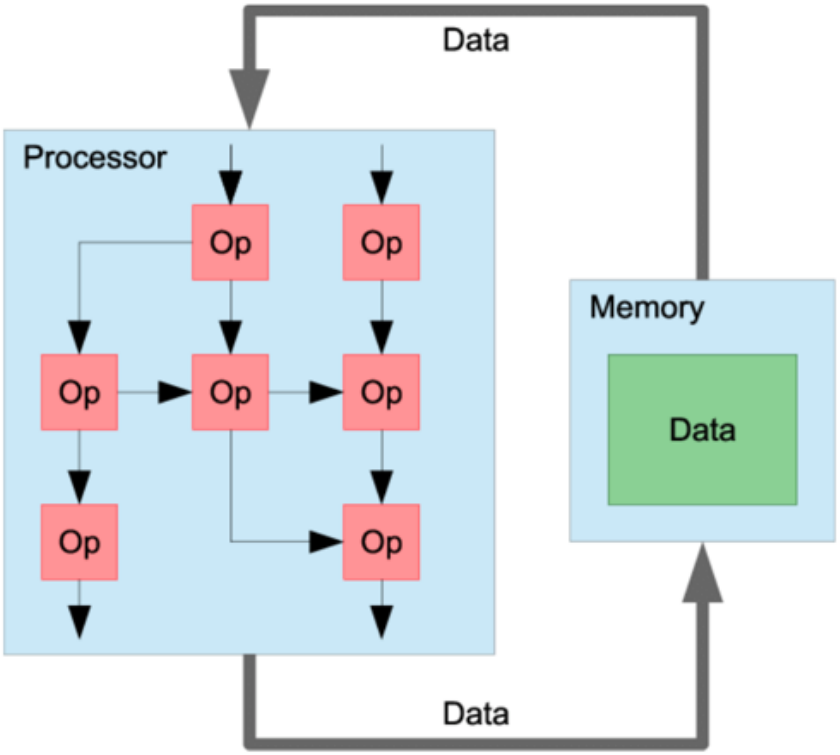
2. Использование детерминированных алгоритмов

В лекциях вроде как указывается, что есть и аппаратные механизмы:

1. Потоковые машины - эффективная реализация параллельных вычислений, реализуются на графах и сетях Петри.

- Особенностью **потоковых машин** является то, что последовательность вычислений задается не последовательностью команд, что характерно для неймановской архитектуры, а по мере готовности данных для выполнения команд. Данные загружаются в операционное устройство, если оно свободно, и для определенной команды имеются все необходимые данные.

- Принципиальным отличием от неймановских машин является то, что команды выполняются не в порядке их следования, а по готовности операндов. Как только на входе операционного блока появляются данные, соответствующая команда блока захватывает эти данные и выполняется соответствующая операция. Машины, основанные на модели с потоком данных, называются потоковыми: нет побочного эффекта и эффективные параллельные вычисления.



2. Пытались использовать архитектуру компа для языков высокого уровня

(пример. picoJava)

3. Flynn’s taxonomy

- **Системы с одной инструкцией и едиными данными (SISD) –**

Вычислительная система SISD-это однопроцессорная машина, которая способна выполнять одну команду, работая с одним потоком данных. В SISD машинные инструкции обрабатываются последовательно, и компьютеры, использующие эту модель, обычно называются последовательными компьютерами. Большинство обычных компьютеров имеют архитектуру SISD. Все инструкции и данные, подлежащие обработке, должны храниться в основной памяти.

- **Системы с одной инструкцией и несколькими данными (SIMD) –**

Система SIMD-это многопроцессорная машина, способная выполнять одну и ту же инструкцию на всех процессорах, но работающая с разными потоками данных. Машины, основанные на модели SIMD, хорошо подходят для научных вычислений, поскольку они включают в себя множество векторных и матричных операций. Чтобы информация могла быть передана всем элементам обработки (PEs), организованные элементы данных векторов могут быть разделены на

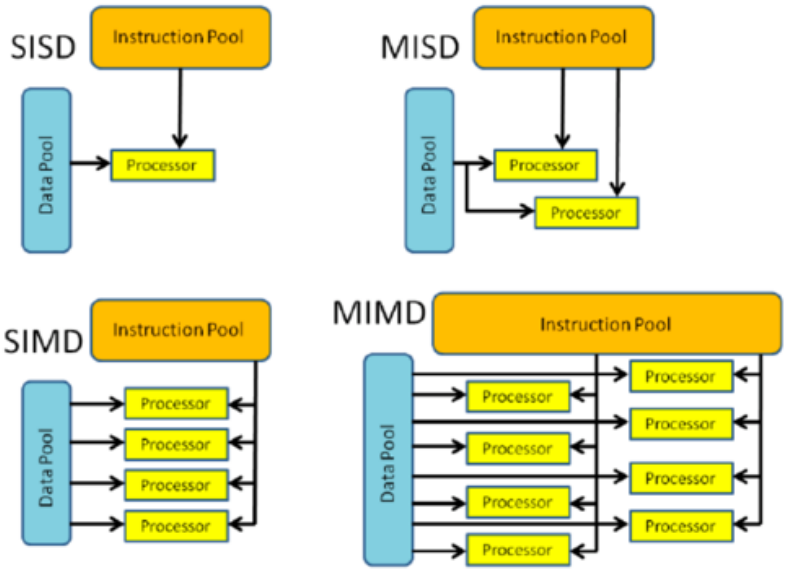
несколько наборов(N-наборов для N систем PE), и каждый PE может обрабатывать один набор данных.

- **Системы с несколькими инструкциями и едиными данными (MISD) –**

Вычислительная система MISD-это многопроцессорная машина, способная выполнять различные инструкции на разных PE, но все они работают с одним и тем же набором данных .

- **Системы с несколькими инструкциями и несколькими данными (MIMD) –**

Система MIMD-это многопроцессорная машина, которая способна выполнять несколько инструкций для нескольких наборов данных. Каждый PE в модели MIMD имеет отдельные потоки команд и данных; поэтому машины, построенные с использованием этой модели, способны работать с любым видом приложений. В отличие от машин SIMD и MISD, PE в машинах MIMD работают асинхронно.



4. Reconfigurable computing

5. Детерменированные по времени процессоры

   - специализированные процессоры (эффективный параллелизм под задачу, эффективная работа с кешом, управление точностью вычислений, более эффективное программирование за счет ручной оптимизации под процессор). Связка DSL+ спец процессор

Связка процессов разработки процессоров и инструментальных средств для разработки чипов.

1. наследники фон Неймановская архитектуры, среди которых:

– tagged architecture (когда попытались учесть на низком уровне типы данных);

29. Классификация Флинна. SIMT архитектура. Области применения и особенности функционирования. Отличия от архитектуры фон Неймана.

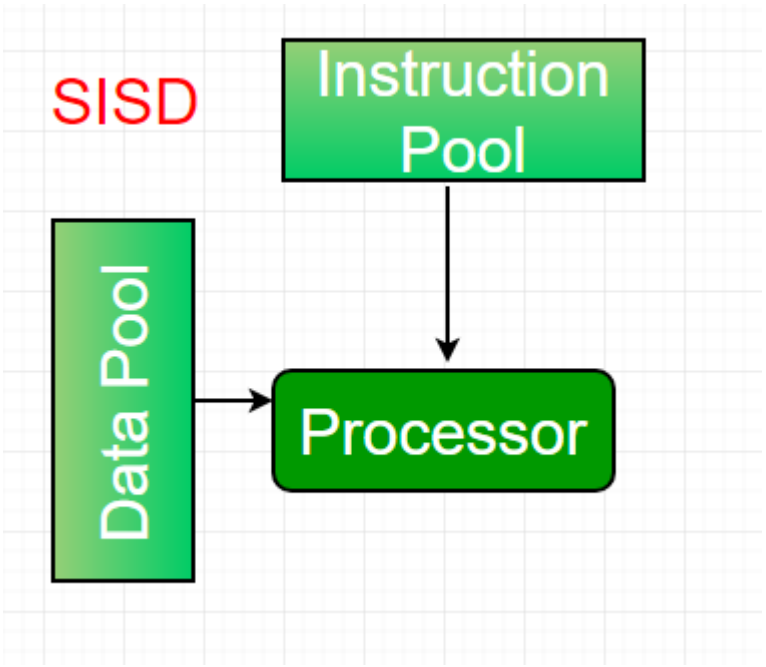
1. Классификация Флинна

Классификация Флинна базируется на понятии **“потока”**, под которым понимается последовательность элементов, команд или данных, обрабатываемая процессором. На основе числа потоков команд и потоков данных Флинн выделяет **четыре класса архитектур: SISD,MISD,SIMD,MIMD.**

SISD (single instruction stream / single data stream)

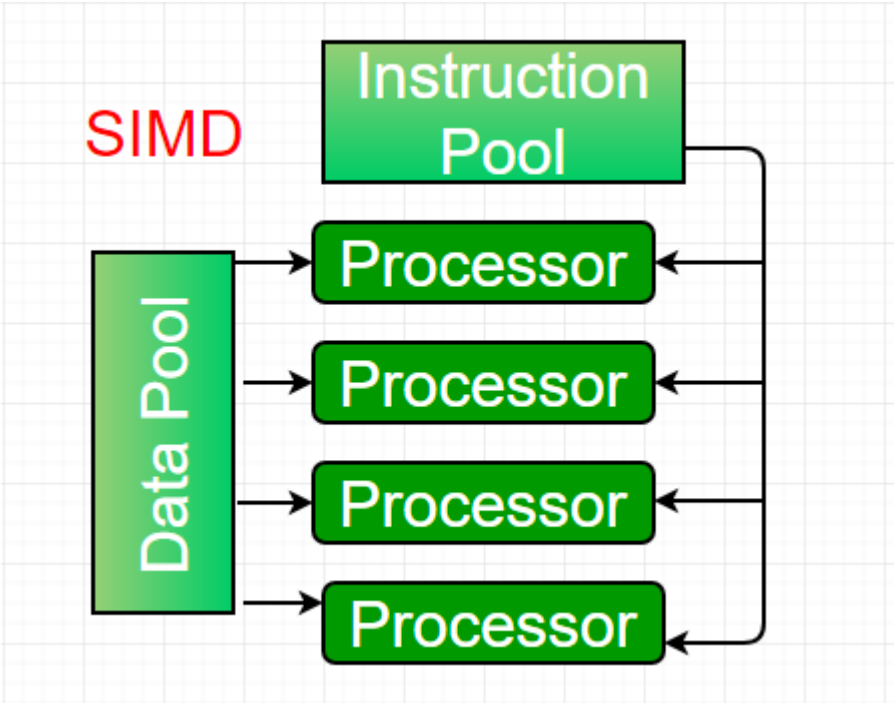
Вычислительная система SISD представляет собой однопроцессорную машину, способную выполнять одну инструкцию и работать с одним потоком данных. В SISD машинные инструкции обрабатываются последовательно, и компьютеры, использующие эту модель, обычно называют последовательными компьютерами. Большинство обычных компьютеров имеют SISD-архитектуру. Все инструкции и данные, подлежащие обработке, должны храниться в основной памяти.
Скорость процессорного элемента в модели SISD ограничена (зависит) от скорости, с которой компьютер может передавать информацию внутри себя. Доминирующими представителями систем SISD являются рабочие станции IBM PC.

классическая архитектура фон Неймана



SIMD (single instruction stream / multiple data stream)

Система SIMD представляет собой многопроцессорную машину, способную выполнять одну и ту же инструкцию на всех ЦП, но работать с разными потоками данных. Машины, основанные на модели SIMD, хорошо подходят для научных вычислений, поскольку они включают множество векторных и матричных операций. Чтобы информация могла быть передана всем элементам обработки (PE), организованные элементы данных векторов могут быть разделены на несколько наборов (N-наборов для систем N PE), и каждый PE может обрабатывать один набор данных. Доминирующим представителем систем SIMD является машина векторной обработки Cray.



MISD (multiple instruction stream / single data stream)

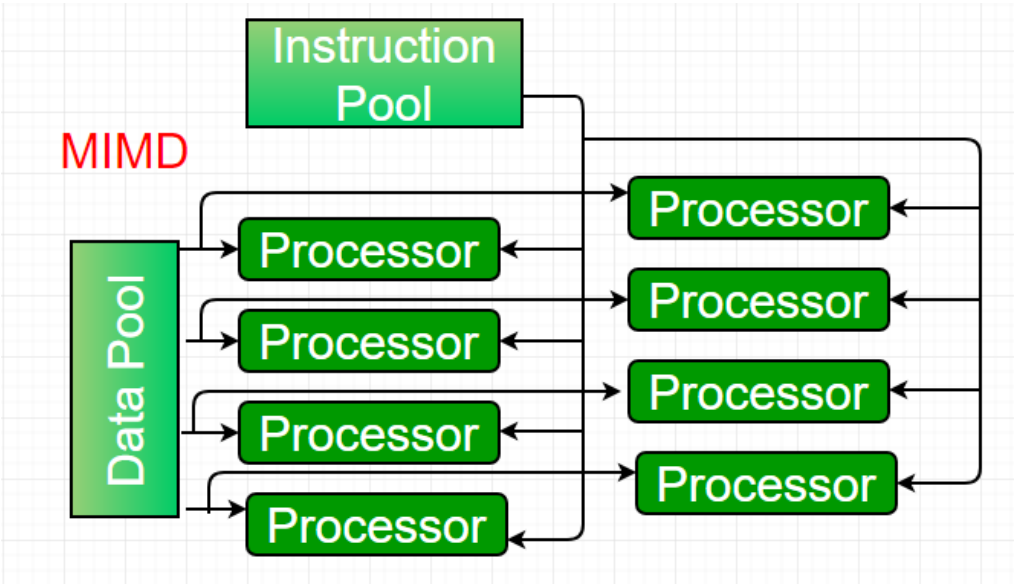
множественный поток команд и одиночный поток данных. Определение подразумевает наличие в архитектуре многих процессоров, обрабатывающих один и тот же поток данных. Однако ни Флинн, ни другие специалисты в области архитектуры компьютеров до сих пор не смогли представить убедительный пример реально существующей вычислительной системы, построенной на данном принципе. Ряд исследователей относят конвейерные машины к данному классу, однако это не нашло окончательного признания в научном сообществе. Будем считать, что пока данный класс пуст.

MIMD (multiple instruction stream / multiple data stream)

Система MIMD представляет собой многопроцессорную машину, которая способна выполнять несколько инструкций для нескольких наборов данных. Каждый **PE (элементы обработки)** в модели MIMD имеет отдельные потоки инструкций и данных; поэтому машины, построенные с использованием этой модели, подходят для любого применения. В отличие от машин SIMD и MISD, PE в машинах MIMD работают асинхронно
Машины MIMD в целом подразделяются на **MIMD с общей памятью** и **MIMD с распределенной памятью** в зависимости от того, как PE связаны с основной памятью.

В модели MIMD с общей памятью все PE подключены к одной глобальной памяти, и все они имеют к ней доступ. Связь между PE в этой модели происходит через разделяемую память. Модификация данных, хранящихся в глобальной памяти одним PE, видна всем остальным PE. Доминирующими репрезентативными системами MIMD с общей памятью являются машины Silicon Graphics и Sun/IBM SMP (Symmetric Multi-Processing).
В распределенной памяти MIMD машины все PE имеют локальную память. Связь между PE в этой модели осуществляется через сеть взаимосвязи (межпроцессный канал связи или IPC). Сеть, соединяющая PE, может быть сконфигурирована в виде дерева, сетки или в соответствии с требованиями.

Архитектуру MIMD с общей памятью легче программировать, но она менее устойчива к сбоям и ее сложнее расширять по сравнению с моделью MIMD с распределенной памятью. Сбои в MIMD с общей памятью влияют на всю систему, тогда как в распределенной модели это не так, в которой каждый из PE может быть легко изолирован. Более того, MIMD-архитектуры с общей памятью с меньшей вероятностью масштабируются, поскольку добавление большего количества PE приводит к конфликтам памяти. Архитектура MIMD с распределенной памятью превосходит другие существующие модели.



2. SIMT архитектура

SIMT (Single instruction, multiple threads) – модель выполнения, используемая в параллельных вычислениях, в которой сочетаются модель SIMD (single instruction multiple data) и многопоточность. Модель SIMT была реализована на нескольких графических процессорах и актуальна для вычислений общего назначения на них.

Так как время доступа оперативной памяти велико, возникла идея скрыть эту задержку, возникающую при доступе к памяти. Скрытие этой задержки – функция планирования с нулевыми издержками, которая реализуется современными графическими процессорами. SIMT предназначен для ограничения накладных расходов на выборку инструкций. Процессор перегружен вычислительными задачами и может быстро переключаться между задачами, когда в противном случае ему пришлось бы ждать памяти.

SIMT обычно используется в суперскалярных процессорах для реализации SIMD. Таким образом, технически каждое ядро является скалярным по своей природе, но оно по-прежнему работает аналогично модели SIMD, используя несколько потоков для выполнения одной и той же задачи с различными наборами данных.

Каждый раз, когда графическому процессору необходимо выполнить определенную инструкцию, данные и инструкции извлекаются из памяти, а затем декодируются и выполняются. В этом случае все наборы данных (до определенного предела), которым требуется одна и та же инструкция для выполнения, предварительно выбираются и выполняются одновременно с использованием различных потоков, доступных процессору.

SIMT была представлена Nvidia в микроархитектуре GPU Tesla с чипом G80. ATI Technologies , теперь AMD , немного позже, 14 мая 2007 г., выпустила конкурирующий продукт - R600 на базе TeraScale 1 Чип GPU

3. vs фон Нейман

Архитектура фон Неймана обладает тем недостатком, что она последовательная. Какой бы огромный массив данных ни требовалось обработать, каждый его байт должен будет пройти через центральный процессор, даже если над всеми байтами требуется провести одну и ту же операцию (узкое место архитектуры)
Для преодоления этого недостатка предлагались и предлагаются архитектуры процессоров, которые называются параллельными. Параллельные процессоры используются в суперкомпьютерах.

30. Изоляция потоков команд. Банки памяти, сегментная память, виртуальная память. Роль в развитии компьютерных систем. Применение на практике. Достоинства и недостатки.

1. Изоляция потоков команд

Для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить код и данные друг о друге, важнейшей задачей ОС является изоляция одного процесса от другого. Для этого операционная система обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

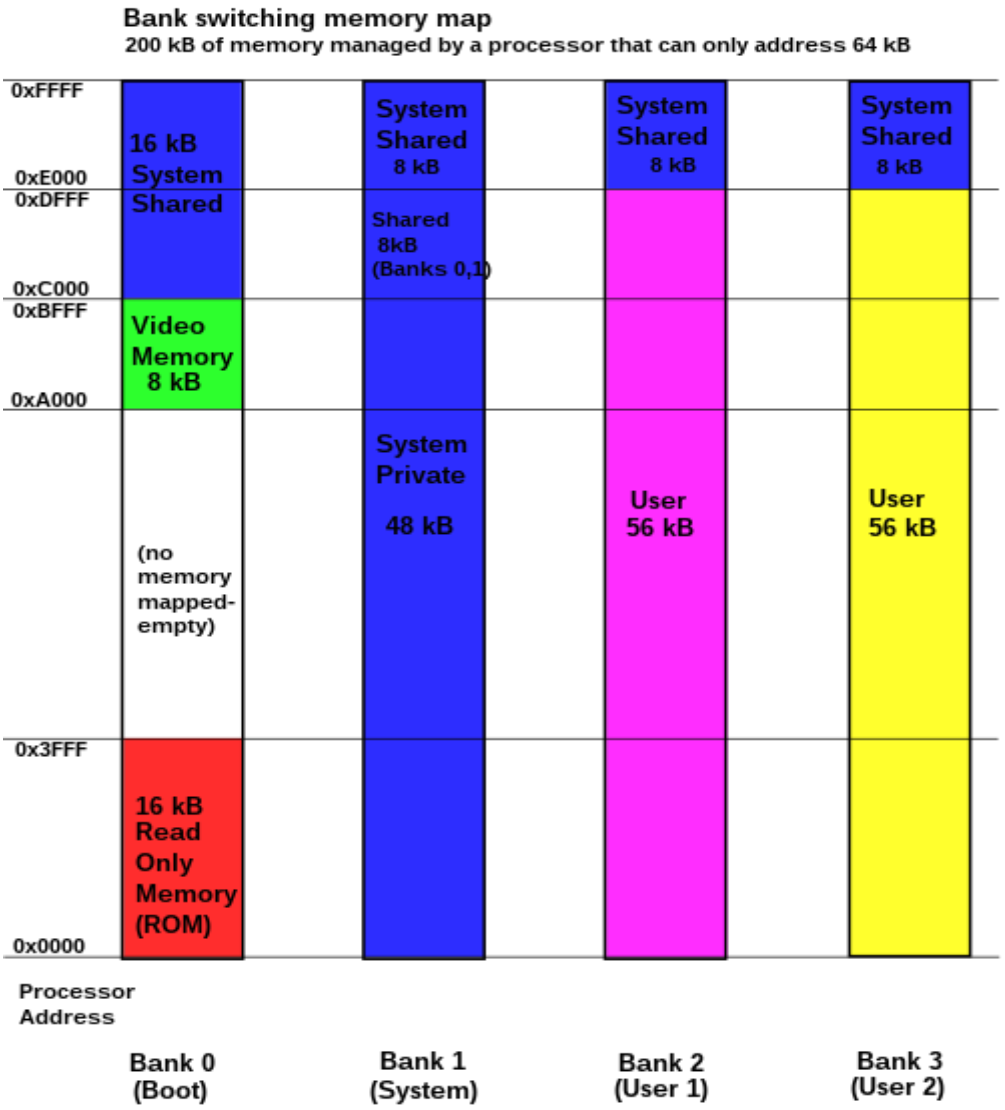
В ОС, где существуют процессы и потоки, процесс рассматривается как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Этот важнейший ресурс распределяется операционной системой между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд. Переход от выполнения одного потока к другому осуществляется в результате планирования текущего потока, и потока, которому следует предоставить возможность выполняться, называется планированием. Планирование потоков осуществляется на основе информации, хранящейся в показателях процессов и потоков. При планировании принимается во внимание приоритет потоков, время их ожидания в очереди, накопленное время выполнения интенсивность обращения к вводу-выводу и другие факторы.

2. Банки памяти, сегментная память, виртуальная память.

Банки памяти используются, когда адресное пространство процессора мало, а приложение требует. При этом стоимостные и электротехнические ограничения позволяют нам установить в систему гораздо больше памяти, чем процессор может адресовать.

Не является инструментом изоляции потоков команд, но может выполнять его роль. Оригинально использовался для того чтобы адресовать больший объём памяти чем это позволяла шина адреса. К примеру: у вас есть 8 бит адресного пространства. Вы можете его расширить следующим образом: вывести из процессора дополнительные 2 бита, которые будут поданы в микросхему памяти в качестве старших битов. Таким образом, их значение будет определять с каким именно банком памяти будет работать процессор.

Другой вариант использования: разделение банков памяти по младшим битам адреса. К примеру: по чётным и нечётным адресам. Может применяться для работы с более широкими машинными словами процессора.



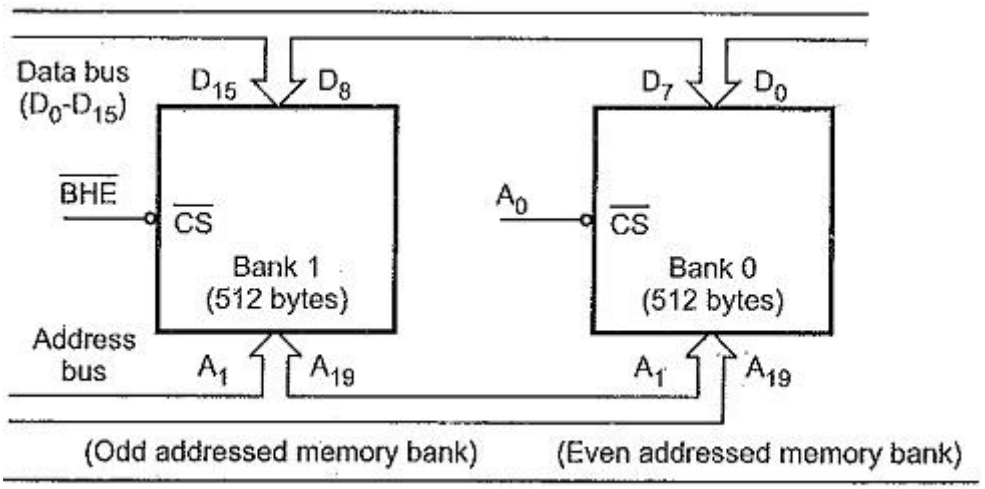
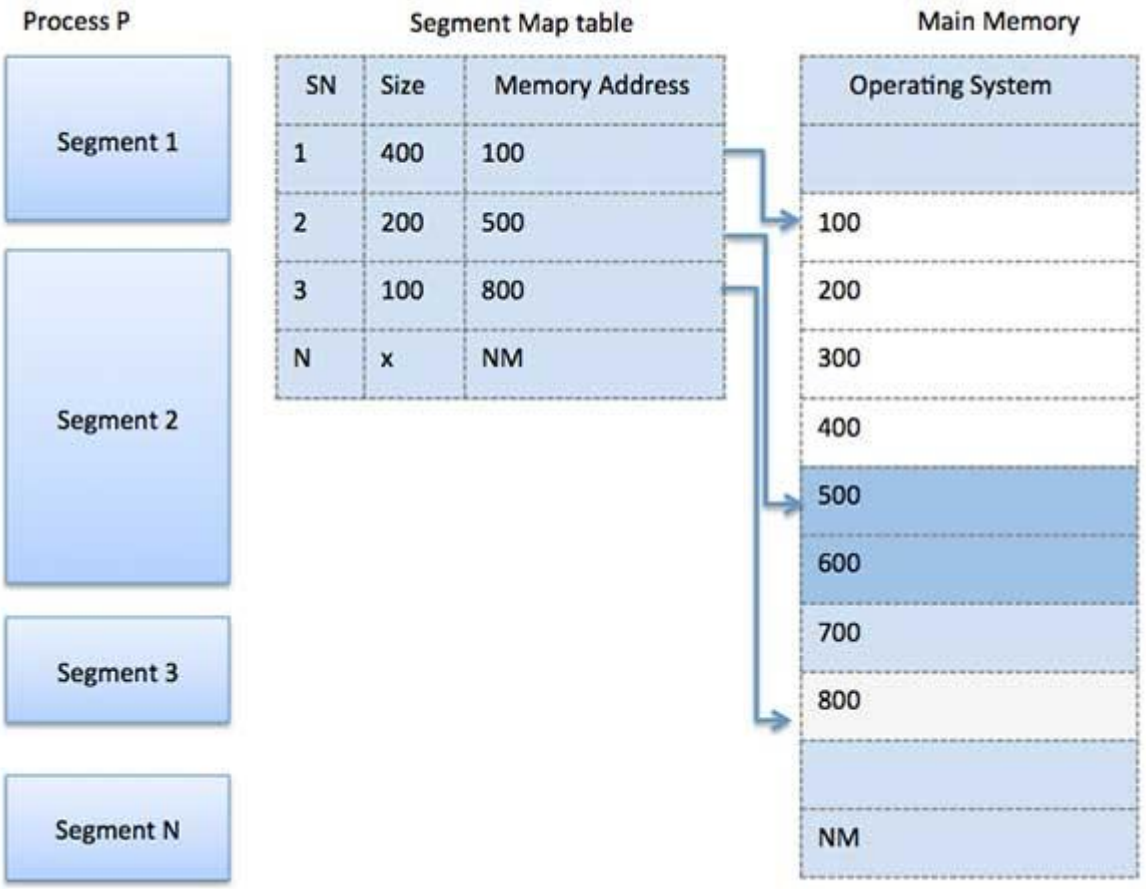


Fig. 10.11 Memory interfacing

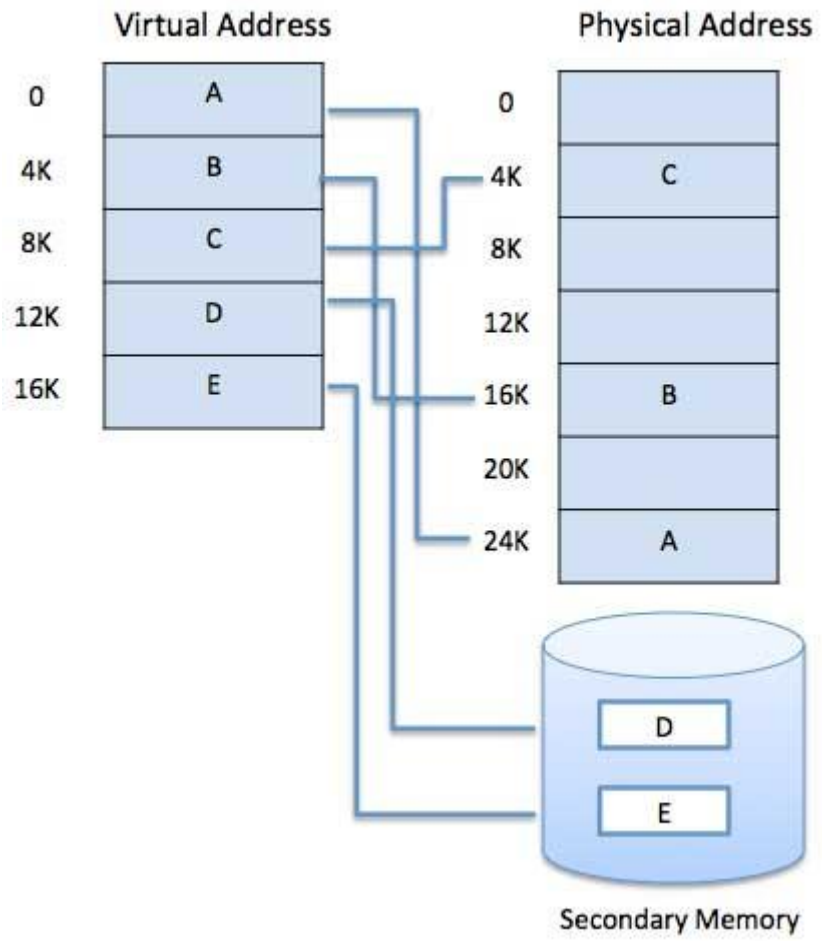
Сегментная адресация памяти – схема логической адресации памяти компьютера в архитектуре x86. Линейный адрес конкретной ячейки памяти, который в некоторых режимах работы процессора будет совпадать с физическим адресом, делится на две части: сегмент и смещение. Сегментом называется условно выделенная область адресного пространства определённого размера, а смещением — адрес ячейки памяти относительно начала сегмента. Базой сегмента называется линейный адрес (адрес относительно всего объёма памяти), который указывает на начало сегмента в адресном пространстве. В результате получается сегментный (логический) адрес, который соответствует линейному адресу база сегмента+смещение и который выставляется процессором на шину адреса.

Назначение:

- Независимая адресация внутри потоков команд относительно сегментов.
- Управление правами доступа к данным в сегменте (rwx, привилегированный режим).
- Независимая адресация и изоляция программных модулей друг от друга.
- Предоставление одного сегмента памяти разным потокам команд для взаимодействия.



Виртуальная память в некотором смысле является развитием и одновременно упрощением сегментной памяти. Основное отличие от сегментной памяти заключается в том, что с точки зрения прикладного программного обеспечения она прозрачна, а также адресное пространство виртуальной памяти (32 бита для шины адреса - 4 Гб) не должно соответствовать адресному пространству физической памяти (может быть меньше, 512 Мб), в тоже время использование физической памяти не обязательно требует использования физической памяти (работа по принципу кеша).



https://habr.com/ru/company/embox/blog/256191/

3. Достоинства и недостатки

Несомненно, применение виртуальной памяти в современных машинах это важнейшее достижение как в компьютерной технологии, так и в максимизации удобств создаваемых для пользователя. Но, как известно, каждая система имеет свои преимущества и недостатки. Предлагаю проанализировать суть применения ВП; её слабости и критерии эффективности, её плюсы и минусы.

Во-первых одним из преимуществ ВП с СО является достаточно большой объём прямо адресуемой памяти. Действительно объем памяти может исчисляться сотнями мегабайт (и даже гигабайтами). Размер виртуальной памяти целиком зависит от объёма накопителя на [жестком] магнитном диске. Созданный SWAP файл размещается на диске и эмулирует оперативную память. При этом пользователь не задумывается о том куда будет помещен “кусок” его программы с которой он только что отработал. Таким образом, ещё одним преимуществом ВП с СО является то, что программы пользователя могут размещаться в

любых свободных страницах. И наконец, одним из важнейших преимуществ ВП с СО (то, ради чего, собственно и была изобретена виртуальная память) повышение уровня мультипрограммной работы. Как было сказано выше, эта цель была одной из самых главных. С организацией ВП с СО пользователь получил реальную возможность загружать в память большее количество программ для того чтобы машина обрабатывала программы сразу (в действительности процессор устанавливает приоритет для каждой программы, находящейся в памяти, и далее в соответствии с приоритетом выделяет определённое количество времени на реализацию каждой программы или команды). Сам процессор постоянно “занят” каждый машинный такт выполняет определенную программу. Метод организации виртуальной памяти со страничной организацией значительно повысил эффективность работы с машиной.

У каждого гениального изобретения к сожалению есть свои недостатки. Таковые есть и у ВП с СО. Попытаемся проанализировать их. Основным недостатком виртуальной памяти пожалуй является то количество времени, которое машина тратит на обращение к внешней памяти. Извлечь необходимую информацию из ячеек оперативной памяти не представляет особого труда и больших затрат времени. Совсем иначе обстоит дело с диском: для того чтобы найти необходимую информацию, нужно сначала “раскрутить” диск, потом найти необходимую дорожку, в дорожке найти сектор, кластер, далее считать побитовую информацию в ОП. Все это требует времени и, порой если при методе случайного удаления страниц *, процессору понадобятся сразу несколько страниц, хранящихся во внешней памяти, большого времени. К сожалению, этот недостаток принадлежит к виду “неисправимых”. И если другие недостатки, рассмотренные ниже еще можно каким-то образом устранить (например путем расширения технических средств и т. д.), то данный недостаток не может быть устранен никоим образом, так как понятие виртуальной памяти ассоциируется с применением внешней памяти (магнитного диска).

Следующий недостаток скорее относится к вопросу о технической характеристике компьютера: наличие сверхоперативной памяти (СОП). Как было сказано выше, СОП.

Методы свопирования страниц имеют большую ёмкость и достаточно высокое быстродействие. СОП используется для хранения управляющей информации, служебных кодов, а также информации к которой осуществляется наиболее частое обращение в процессе выполнения программы. Этот недостаток в работе с ВП к счастью можно ликвидировать. Что касается технической характеристики есть ли в микросхемах оперативной памяти дополнительные интегральные схемы, которые являются запоминающими устройствами СОП? Если есть, то проблема с СОП решена, а если нет..? Тогда, благодаря достижениям в области компьютерной технологии, могут использоваться драйверы, резервирующие маленькую область ОП для имитирования СОП(стандартные операционные процедуры). Итак, что касается этого недостатка, то, мне кажется, что он не настолько серьёзен, чтобы о нём беспокоится. И, наконец третьим недостатком является внутренняя фрагментация страниц.

31. Уровневая организация компьютерной системы. Примеры. Что такое уровень компьютерной системы и что он определяет? Системные свойства компьютерных систем.

Уровень компьютерной системы -- совокупность процессора, вычислительных процессов и их моделей. Уровни определяются вне зависимости от конкретной стадии жизненного цикла вычислительной системы и могут включать в себя как архитектурные представления, так и модели сформированные в процессе отладки. Состав уровня компьютерной системы: Модель вычислительного процесса (модель) -- описание вычислительного процесса на определённом уровне ВС. К примеру: программа на С, конфигурация ПЛИС, схема электрическая принципиальная,

архитектурная спецификация, журнал. В качестве модели может выступать спецификация любого размера, ограничение -- единая модель вычислений.Вычислитель (процесс) -- целостный элемент ВС, определяющий возможные варианты развития вычислительного процесса или процессов в рамках отношения актуализации. Является частным случаем вычислительного механизма.Вычислительный процесс (процесс, ВП) -- последовательность смены состояний вычислителя, соответствующая указанной модели (если применимо).

Основная идея -- сформировать новый уровень организации вычислительного процесса, имеющий новые свойства: изменчивость, чувство безопасности, удобство задачи вычислительного процесса.
Организация архитектур машин представляется как ряд уровней, каждый из которых надстраивается над нижележащим уровнем. Это сделано не просто так, с помощью многоуровневой архитектуры мы можем абстрагироваться от реализации и сложности нижнего уровня, тем, самым облегчить процесс проектирования, и уменьшить вероятность ошибок. Абстрагирование является ключевым моментом во всей архитектуре компьютера.
Организация компьютера состоит из 6 уровней: цифровой логический, микроархитектура, архитектура набора команд (ISA), операционная система, язык ассемблера, прикладной, конечно, если не считать физический уровень, который находится ниже цифрового логического уровня

Составляющие компьютерной системы, как информационной, могут выполнять 5 основных функций (одну или несколько сразу):

1. получение информации из внешних источников;

2. выдача информации;

3. хранение информации;

4. передача информации;

5. обработка информации

Тут инфа не из конспекта, лучше ориентироваться на конспект

32. Явление дезагрегация. Место явления в развитии компьютерных систем. Тенденции и перспективы. Принцип развития иерархических систем Седова.

Процесс дезагрегации в компьютерных системах – это изменение рынка, представляющее из себя фрагментацию, разделение на обособленные экосистемы вокруг вычислительной платформы. По сути: с развитием компьютерных технологий на рынок выходит все больше разработчиков, что в свою очередь усложняет процесс системной интеграции продуктов. Получается, что компании стараются отделиться от

всей остальной массы, (как говорилось выше) создают свои экосистемы и концентрируют внимания на своих продуктах.

Принцип развития иерархических систем Седова: в сложной иерархически организованной системе рост разнообразия на верхнем уровне обеспечивается ограничением разнообразия на предыдущих уровнях, и наоборот, рост разнообразия на нижнем уровне разрушает верхний уровень организации (то есть, система как таковая гибнет).

Фактически, данный принцип описывает процесс дезагрегации.