Лекция 2. Архитектурные особенности современных компьютеров. Память.

Содержание

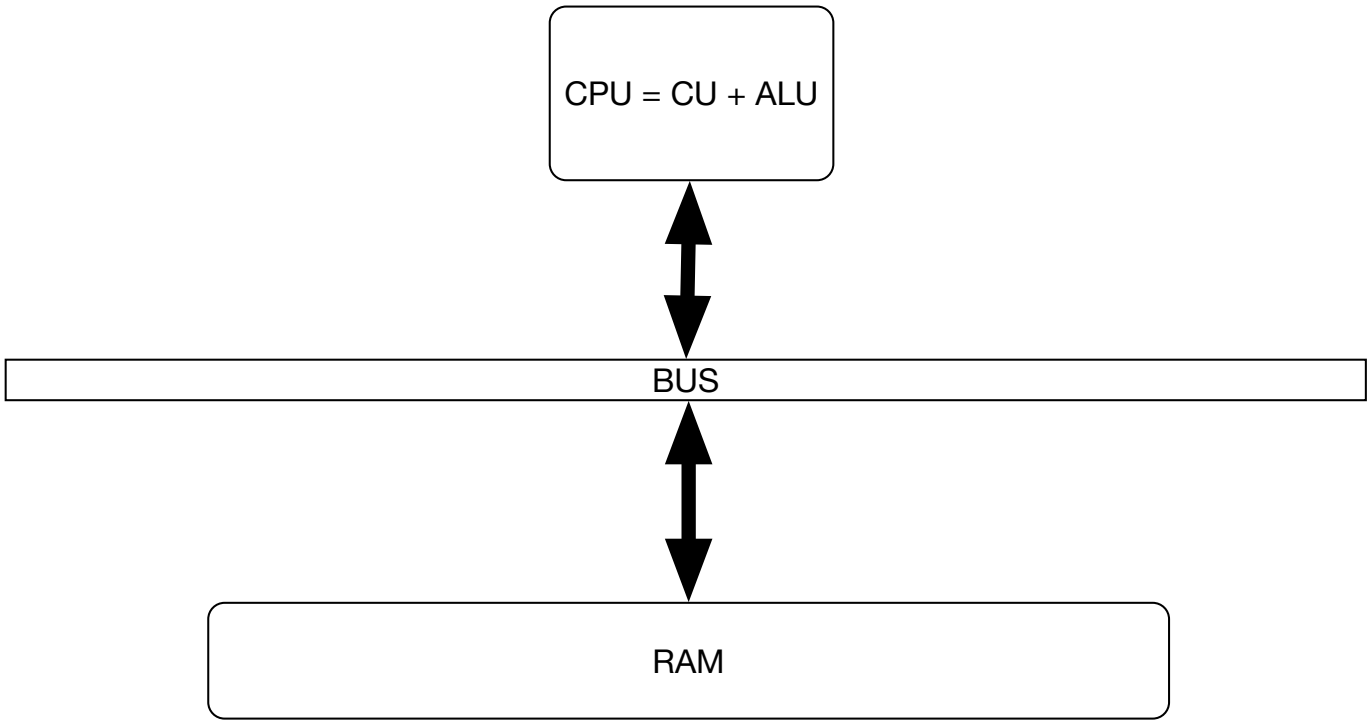
- Архитектуры Принстонская и Гарвардская.
- Архитектура современных ЭВМ с точки зрения программиста.
- Кэш-память и её влияние на производительность.
- Особенности программирования, учитывающего влияние кэша.
- Основные узкие места в вычислительных системах.

Архитектуры Принстонская и Гарвардская

- Первые программисты коммутировали программу на наборном поле проводами.
- Сейчас программа хранится в памяти.
- Два основных вида разделения памяти между программами и данными.
  - Общая память для хранения данных и программ — Принстонской или фон Неймана.
  - Раздельная память для данных и программ — Гарвардская.

Принстонская архитектура

- RAM — *память со случайным доступом*. Соединена с исполнителем, *процессором*, устройством связи (*шиной*). Процессор содержит *счётчик команд*. При исполнении команды из памяти извлекаются *операнды*, которые участвуют в операции, результат помещается в память.

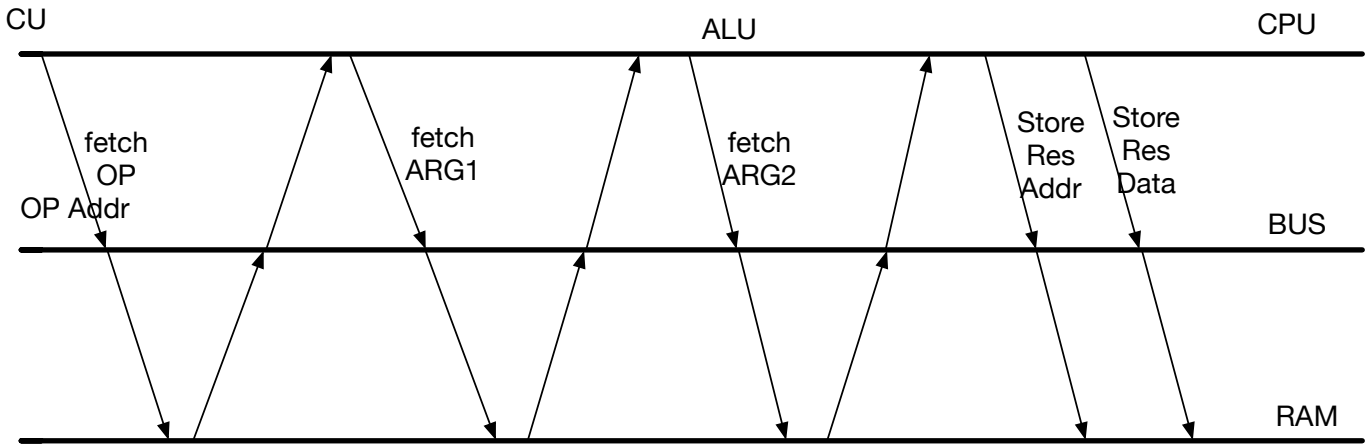


Принстонская архитектура

- Компоненты процессора:
  - АЛУ, ALU — исполняет операции арифметические и логические
  - УУ, CU — выбирает команду и операнды из памяти, определяет исполняемую команду, передаёт команду и операнды в АЛУ и записывает результат операции в RAM.

  Посредник — *шина*, BUS. Первые шины — набор проводников.

Принстонская архитектура: узкие места



Исполняется инструкция с двумя операндами.

1. На шину выставляется адрес инструкции.
2. Память отслеживает операцию и выставляет на шину значение по адресу.
3. CPU ждёт. Код команды получен. Команда декодирована. Запрос операнда 1. Ожидание операнда 1. Запрос операнда 2. Ожидание операнда 2.
4. Передача исполнения ALU.
5. Запрос очередной инструкции. ...

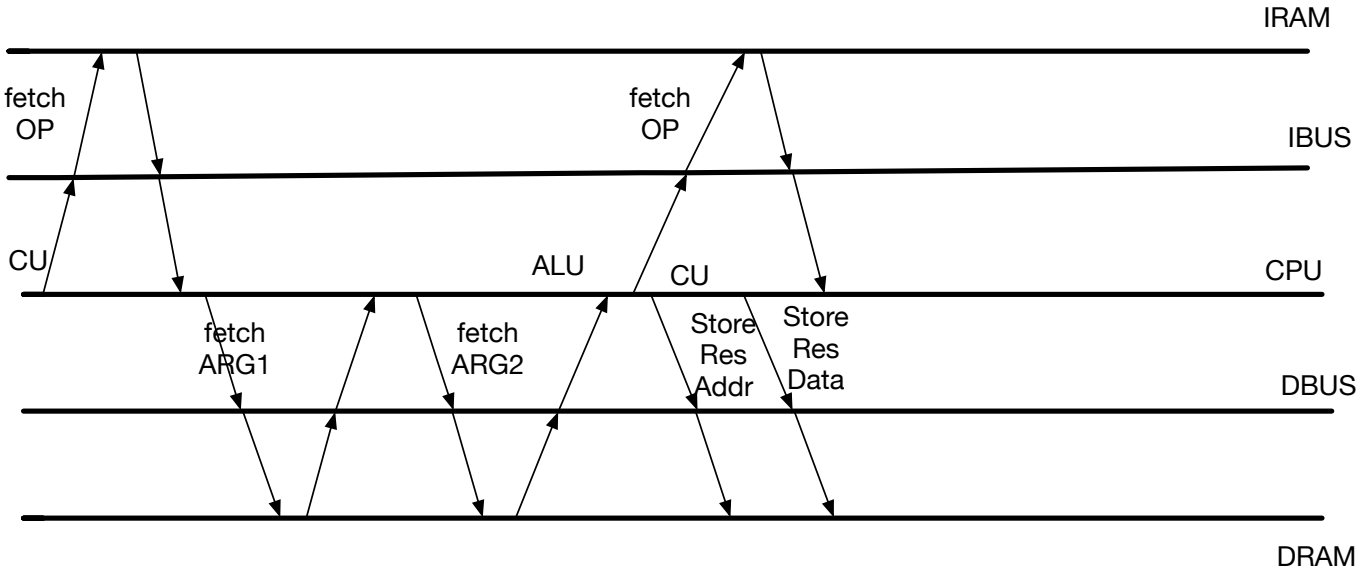
Три устройства и все работают строго последовательно.

Гарвардская архитектура

Память программ отличается от памяти данных.

- Адресные пространства памяти и организация памяти могут быть различными.
- Разрядности адреса и данных могут быть различными.
- Каждое адресное пространство обслуживается собственной шиной.
- Например, шина адреса программ 15 бит адрес 45-битных слов мультиплексируемая, а шина адреса данных 12 бит адреса 18-битных слов раздельная.

Гарвардская архитектура: узкие места



Исполняется инструкция с двумя операндами.

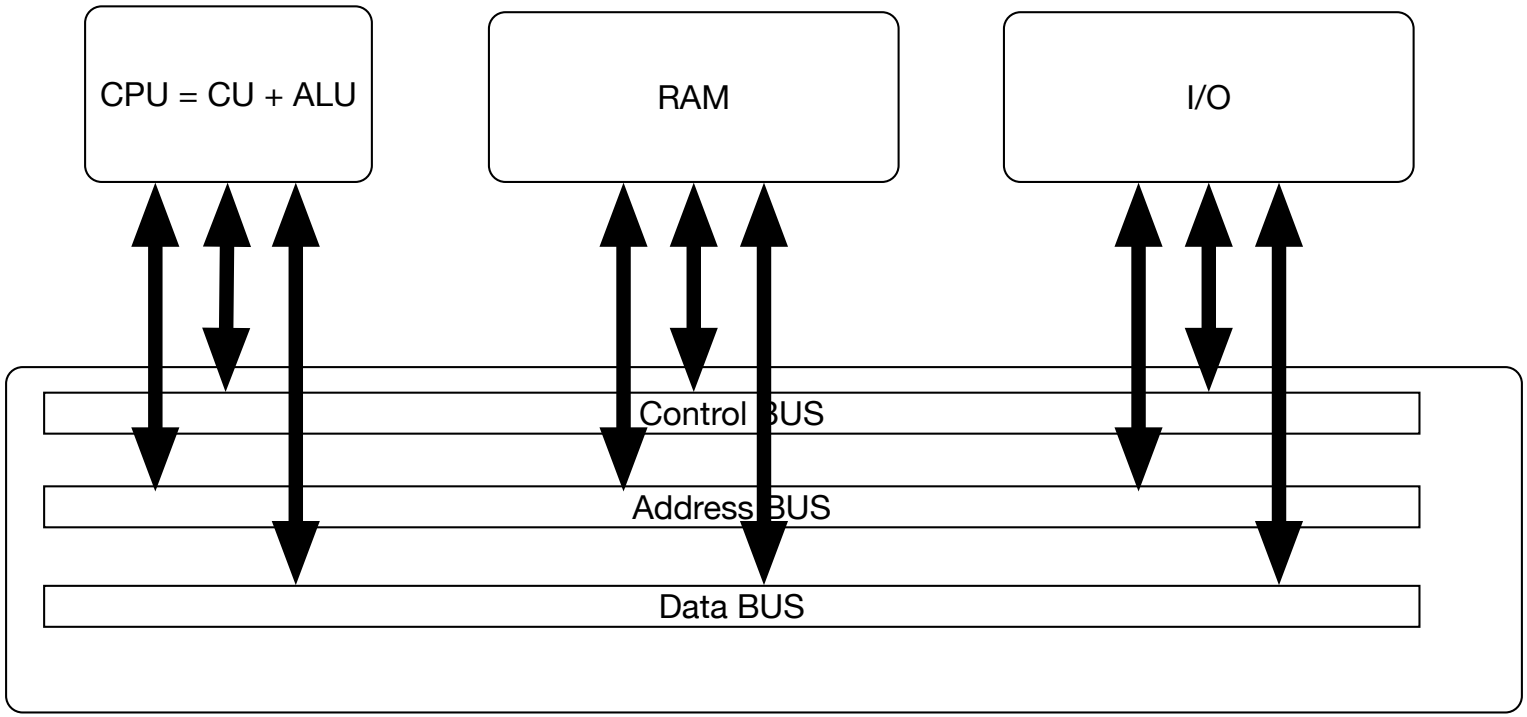
1. Выставляется адрес инструкции на шину адреса
2. Память выставляет значение по адресу на шину.
3. CPU ждёт. Код команды получен. Команда декодирована. Запрос операнда 1. Ожидание операнда 1. Запрос операнда 2. Ожидание операнда 2.
4. Передача исполнения ALU и параллельно запрос очередной инструкции. ...

Сравнение архитектур

- Гарвардская архитектура должна быть более производительной, более сложной в реализации и более дорогой в производстве.
- Бутылочное горлышко обеих архитектур — наличие шин.
- Вторая шина в Гарвардской архитектуре даёт ей несомненное преимущество с точки зрения производительности.

Современное состояние дел

- Для программистов интересна неразличимость данных и программ (Принстонская).
- Производители имитируют удобную архитектуру любым удобным образом.



Современное состояние дел

- Продуктивная мысль Гарвардской архитектуры: как можно большее количество операций исполнять параллельно.
- Добавим ещё одну шину адреса.
- Добавим шину ввода/вывода.
- Ввод/вывод отдадим периферийным процессорам — пока идёт ввод/вывод, CPU простаивает.
- Ввод/вывод с помощью CPU перенесём на мини- и микроЭВМ (70-е годы)
- Ввод/вывод только с помощью периферийных процессоров (сейчас)

Современное состояние дел

- Все современные вычислительные системы обладают дуализмом
  - являются реализацией модифицированной Принстонской архитектуры.
  - выглядят как чистая Принстонская архитектура для обычных программистов.
- Понимание особенностей современной архитектуры для продвинутых программистов необходимо для эффективного использования доступных ресурсов.

Архитектура современных ЭВМ с точки зрения программиста.

Архитектура современных ЭВМ

- CPU
- Cache
- RAM
- Chipset
- Memory controller
- Peripheral controller

Архитектура: CPU

Главный компонент ЭВМ.
Несколько определений из теории ОС:

- **Вычислительное ядро** — компонент центрального процессора, исполняющий машинные команды. Видится как отдельный процессор, исполняющий свой собственный поток инструкций.
- **Вычислительный поток** — абстракция виртуального процессора, исполняющая свой собственный поток инструкций в адресном пространстве, разделяемом с другими вычислительными потоками, исполняющимися в контексте одного процесса.
- **Процесс** — единица исполнения операционной системы, характеризующаяся собственным адресным пространством. Процесс состоит по крайней мере из одного вычислительного потока.

Архитектура: CPU

Каждый процессор имеет по крайней мере два режима работы:

- **системный**, в котором возможно исполнение любой машинной команды, включая команды ввода/вывода и изменения системных регистров процессора.
- **пользовательский**, в котором привилегированные команды приводят к аппаратному исключению.

- **Системный вызов** — запрос к временному переводу режима процессора в системный для исполнения особых заявок.
- **Переключение режимов** процессора происходит при каждом системном вызове. Это — дорогая операция (сотни тактов).

Архитектура: CPU: контекст процесса

Каждый процесс имеет два контекста

- **пользовательский**, который состоит из адресного пространства (виртуальной памяти)
- **системный**, включающий компоненты процесса, изменяемые только с помощью системных вызовов, такие, как таблицы страниц, таблицы открытых файлов, аутентификационная информация.

Если процесс не исполняет системных вызовов, он может изменять только зафиксированное множество ячеек в своём адресном пространстве.

Архитектура: CPU:контекст потока

Каждый из исполняющихся в адресном пространстве процесса потоков разделяет с процессом его пользовательский и системный контексты.

- **Контекст потока** — значение регистров процессора. Один из регистров контекста есть программный счётчик, другой — указатель стека.
- Указатель стека у каждого потока свой, но нельзя строго сказать: стек является частью контекста потока. Правильно: стек потока адресуется из контекста потока.
- В первом приближении для большинства применений можно рассматривать стек потока как часть контекста потока.

Архитектуры: CPU: переключение контекстов

Переключение потоков есть переключение регистров внутри процесса. Оно обычно инициируется *планировщиком* операционной системы $\rightarrow$ происходит переключение режимов процессора $\rightarrow$ это долго.

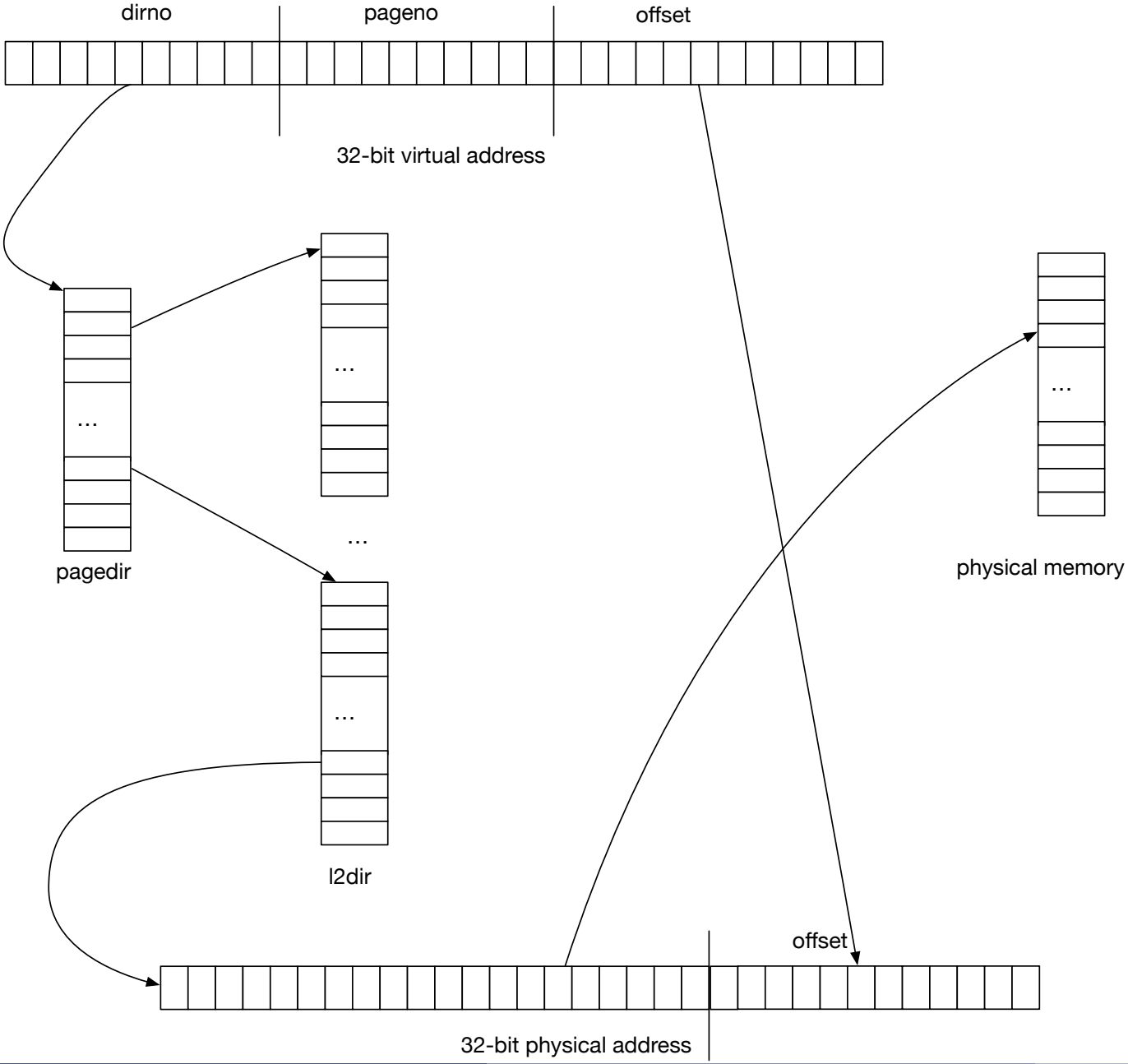
Расход времени на переключение потоков есть сумма времени переключения режима процессора и времени в течение которого процессор находится в программе планировщика, выбирая очередной готовый к выполнению процесс из очереди (и перемещая его из головы очереди в её хвост).

Это требует использования совместной структуры данных внутри ядра и соответствующих, не очень быстрых, машинных команд (это ещё увидим).

Архитектуры: CPU: переключение контекстов

Переключение процессов: планировщик обнаружил, что управление требуется передать потоку, который принадлежит неактивному процессу $\rightarrow$ помимо переключения потоков требуется переключить процессы.

Архитектуры: CPU: переключение процессов



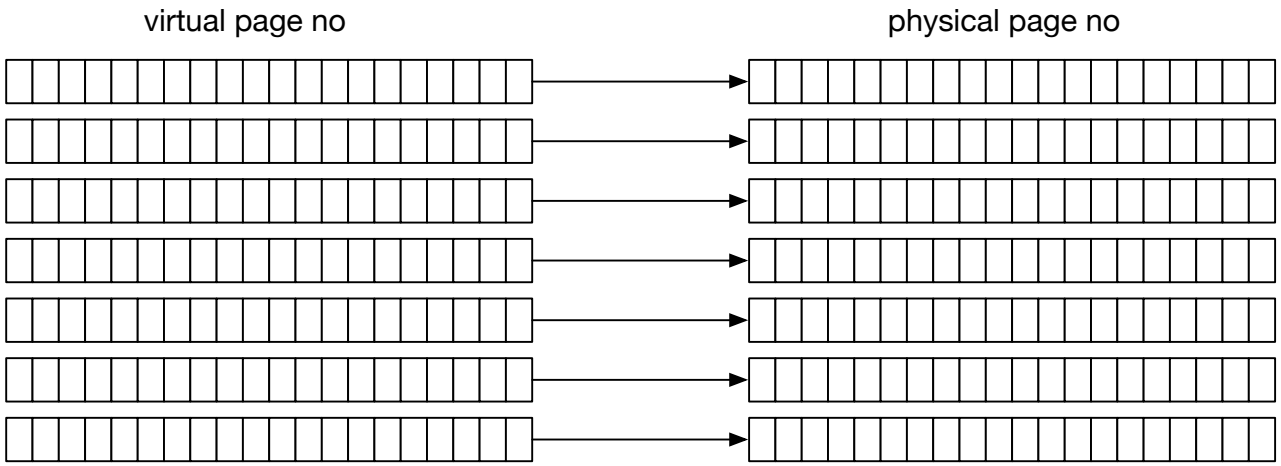
Архитектура: CPU: трансляция адресов

32-битный защищённый режим архитектуры x86. Для **каждого** адреса:

1. 32-битный адрес делится на три области
2. 10 бит используется для адресации номера директории. Всего возможно до 1024 директорий, каждая из которых имеет размер в 1024 адреса.
3. 10 бит используется для нахождения адреса внутри директории.
4. Из директории извлекается запись, содержащая 20-битный номер страницы физической памяти.
5. Младшие 12 бит логического адреса приписываются к получанному 20-битному значению, формируя 32-битный физический адрес.

Единичное обращение к памяти приводит к трём обращениям к памяти. Самый медленный из всех ресурсов оказывается ещё медленнее.

Архитектура: CPU: TLB



- Ассоциативный массив отображений.
- 20-битный номер логической страницы сравнивается с записями, при наличии в TLB обращения к таблицам нет.
- Очень хорошо при последовательном обращении к памяти.
- Неэффективно при истинно случайных обращения к памяти.

Архитектура: CPU: TLB и процессы

- При переключении процессов изменяется CR3.
- Все записи в TLB приходится уничтожать (машинная команда INVTLB).
- Активный процесс начинает с заполнения TLB, на что ещё уходит несколько сот тактов.
- При переключении потоков CR3 на меняется, TLB не очищается.

Архитектура: CPU: прерывания

- Многозадачная система не может позволить монополизирование процессора потоком.
- Если ядро одно и поток считает пи до миллионного знака, всё должно продолжать работать.
- В системе происходят аппаратные события. Приходят сетевые пакеты. Заканчиваются операции ввода/вывода на жёстких дисках. Идут часы. Для этого требуется процессор и он отбирается даже у самого тяжёлого процесса.
- Происходят прерывание по завершению ввода/вывода.
- Происходят прерывания по таймеру.

Архитектура: CPU: прерывания

- При любом прерывании управление передаётся ядру операционной системы.
- (Пока оставим в стороне DPC.)
- Прерывание приводит к передаче управления планировщику.
- Планировщик определяет наиболее приоритетный поток и активизирует его, возможно, переключая контекст потоков, возможно — процессов.
- Любой системный вызов приводит к активизации планировщика.

Архитектура: Chipset, BUS

- Данные между процессором и памятью передаются по шине.
- В модифицированной архитектуре фон Неймана понятие шины становится размытым и вместо него лучше использовать термин Interconnect — соединитель.
- Interconnect при плохом проектировании может с лёгкостью оказаться бутылочным горлышком всей вычислительной системы.
- Часто именно Interconnect, реализуемый системным набором логики, чипсетом, становился бутылочным горлышком.

Архитектура: Interconnect

- 90-t годы. `Intel`,`AMD`,`Cyrix`,`VIA`,`IBM` выпускают процессоры x86.
- Процессор требует материнской платы и оперативной памяти.
- Наборы системной логики (чипсеты) выпускают `Intel`,`SIS`,`ALI`,`UMC`,...
- Каждый производитель применяет оригинальное решение.
- Эффективно реализовать протокол обмена процессор $\leftrightarrow$ чипсет $\leftrightarrow$ оперативная память, как оказалось, не так просто.
- Разница в производительности всей системы достигала двух раз.
- Если `Interconnect` слаб, процессор простаивает.

Peripheral controller

- Для увеличения работы в единицу времени часть действий можно перенести на периферийные устройства.
- Сетевые адаптеры — первый опыт. `TCP offload`.
- Следующий шаг — обработка трёхмерных изображений. Z-buffer, затенение Гуро, Фонга, `mip-mapping`.
- Стандарт `OpenGL` поощрял конвейеризацию графических операций.
- Сейчас вычислительную мощность видеокарт некуда девать. `Cuda`, `OpenCL`.

Архитектура: Interconnect

- Ещё способ — добавлять внешние процессоры, взаимодействующие с основным через Interconnect.
- Intel Xeon Phi — до 72 4-CPU процессора, каждый с локальной (близкой) памятью. Все вместе используют общую (далёкую) память.
- Инструменты — OpenMP, OpenCL, MPI.

Архитектура: RAM

- В 2020 году массово применяются два вида памяти:
  - **статическая** — значение бита определяется состоянием ячейки, типично 6 или 8 транзисторов. Состояние устойчивое,регенерация не требуется, дорогая, быстрая.
  - **динамическая** — значение бита определяется зарядом конденсатора. Состояние неустойчивое, считывание значение разрушает заряд, требуется регенерация, дешёвая, медленная.

Архитектура: DRAM: принцип работы

- Пусть имеется запрос на 4 байта памяти.
- Пусть имеется модуль памяти с частотой 2400 MHz.
- Логически DRAM есть двумерный массив столбцов и строк.
- Для получения значения нужно подать на шину управляющие сигналы выбора номера строки ($RAS\sharp$).
- Значение сохраняется в буферах на основе статической памяти.
- В следующем цикле подаётся сигнал выбора номера столбца ($CAS\sharp$), вместе с ним сигнал направления передачи (чтение или запись).
- Каждая операция требует фиксированного количества тактов.
- Для нашего модуля это, например, 27-9-9-9 – общая латентность на разных этапах запроса данных. Общая задержка не меньше суммы (54) что даёт не менее 25 наносекунд.

Архитектура: DRAM

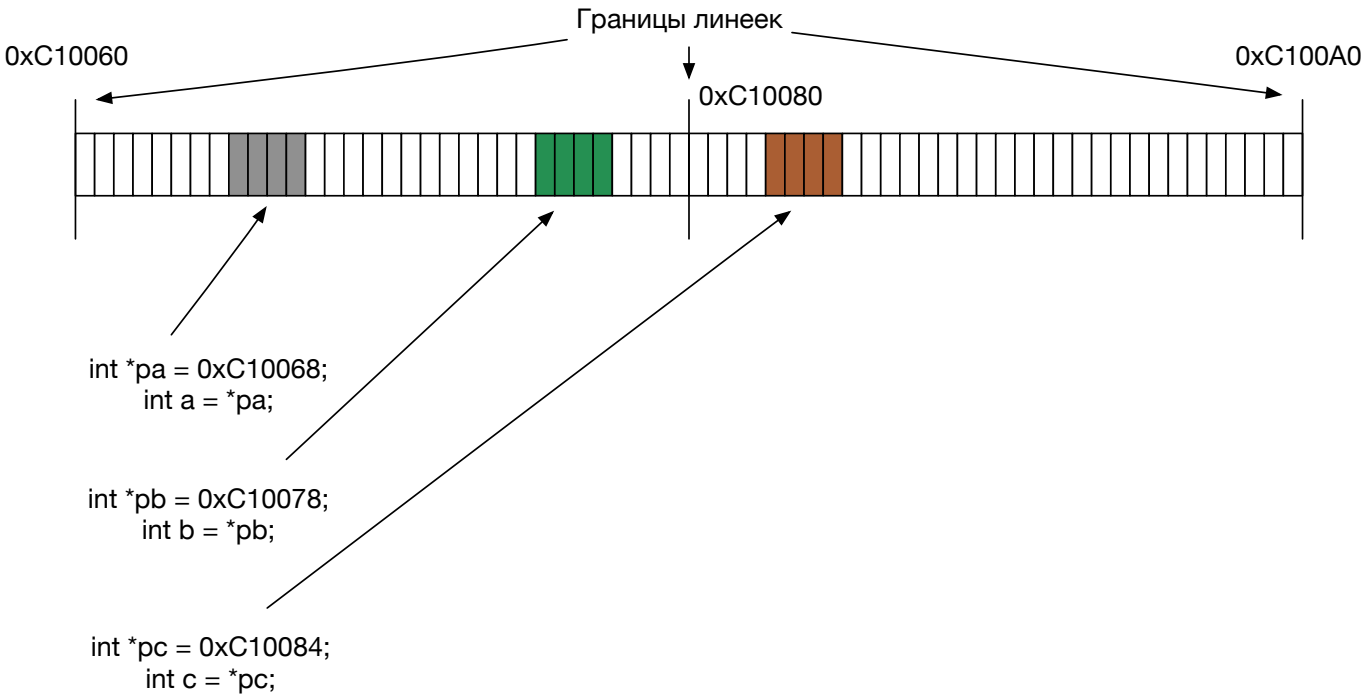
- Для ускорения работы адресные сигналы подаются на группу модулей.
- Передача из модулей происходит параллельно.
- Из памяти считывается не запрошенные 32 бита, а целая линейка из 256-512-1024 битов.
- Количество битов определяется шириной шины памяти.

Кэш-память и её влияние на производительность.

Кэширование: зачем нужно

- Память — узкое место современных вычислительных систем.
- Ширина шины памяти увеличилась от 8 до 128 битов, пропускная способность увеличилась.
- Латентность изменилась незначительно.
- Введение кэш-памяти помогает решить две основных проблемы:
  1. При запросе 32-х битов общая задержка определяется только латентностью. Следующие 32 бита достаются чуть быстрее (всё равно долго). Контроллер кэша запрашивает *линейку* в 32-64 байта. Эффективная латентность на байт уменьшается.
  2. Линейка хранится в кэше и при повторном обращении к тому же участку, физической передачи информации по `Interconnect` не происходит — нагрузка на `Interconnect` уменьшается.

Кэш: принципы



- При первичном обращении к a загружается линейка кэша размером 32 байта, большая латентность.
- Обращение к b (в линейке) не требует обращения к физической памяти
- Каждая линейка кэша принадлежит ровно одной странице виртуальной памяти $\rightarrow$ адрес уже в TLB.
- Первое обращение к c из другой линейки $\rightarrow$ кэш-промах.

Кэш: принципы

- Возможен третий способ увеличения производительности: при записи в кэш-память управление процессору передаётся немедленно, сама же операция записи произойдёт в удобное для контроллера время.
- Это позволяет комбинировать запросы на запись и сливать их в одну транзакцию, что ещё более увеличивает производительность.

Кэш: принципы

Общий принцип работы кэш-памяти:

- Вся память делится на линейки фиксированного размера. Адреса выровнены $\rightarrow$ младшие биты адреса линейки содержат нули (64 байта $\rightarrow$ 6 нулей).
- Первое чтения (*cache miss*) $\rightarrow$ линейка с данным адресом загружается в кэш. Невыровненное обращение к памяти to в кэш загружаются две линейки. После загрузки линейка *присутствует* в кэше.
- $\forall$ запрос к памяти $\rightarrow$ контроллер кэша проверяет наличие адреса в тегах кэша. Имеется(*cache hit*) $\rightarrow$ запрос исполняется из кэша.
- Если кэш заполнен, но требуется загрузка, возникает ситуация вытеснения (*evict*) какой-либо линейки. Обычно реализуется по LRU-подобному алгоритму.

Кэш: принципы

- Полностью ассоциативный кэш может назначать любому адресу памяти любую линейку. Дорого. TLB. Ассоциативность:
  - каждый адрес может отображаться не на все возможные линейки, а только на определённое количество линеек. Например, адрес 0xF0040 при размере кэш-памяти в 512 линеек мог бы отображаться только на 1-ю, 5-ю, ... $4n + 1$ линейки.
  - Каждая линейка может обслуживать не все адреса, а только содержащие определённый набор битов. Например, 1-я линейка могла бы содержать только адреса, остаток от деления начала которых на 2^8 равен 0x40.

Кэширование прозрачно для большинства программ. Знаем про кэширование $\rightarrow$ увеличиваем производительность. Иногда в 10 раз.

Кэш: эффективность использования

Эффективность кэша определяется несколькими факторами:

1. Отношением количества попаданий в кэш к общему числу запросов.

$$HitRatio = \frac{CacheHits}{CacheHits + CacheMisses}$$

2. Скоростью обработки.
   - T_R — среднее время доступа к памяти.
   - T_C — среднее время доступа к кэшу.
   - $H = \frac{CacheHits}{CacheHits + CacheMisses}$ — вероятность попадания в кэш.

   Тогда математическое ожидание времени доступа к элементу памяти будет равно

$$T_{eff} = T_C \times H + (T_R + T_C) \times (1 - H)$$

Кэш: эффективность использования

Пусть $T_C = 1$ и $T_R = 30$

H	T_{eff}
0.90	4
0.91	3.7
0.92	3.4
0.93	3.1
0.94	2.8
0.95	2.5
0.96	2.2
0.97	1.9
0.98	1.6
0.99	1.3
1.00	1.0

При 95% попадании в кэш эффективная производительность уменьшается в 2.5 раза!

Кэш: эффективность использования

Для максимизации попадания кода и данных в кэш, есть подходы:

1. обеспечить высокую локальность данных:
   - хорошие компиляторы, помещают переменные в регистры;
   - хорошие программисты стараются использовать алгоритмы, не носящиеся по памяти;
2. увеличить количество кэша:
   - к первыму уровню, самыму быстрому, но и самыму маленькому, добавим второй, третий и т. д. уровни, каждый из которых будет всё больше, но и всё медленнее.
   - L1: десятки килобайт, время обслуживания 2-3 такта
   - L2: сотни килобайт, время обслуживания 5-10 тактов, HitRatio=99%.
   - L3: сервера и многоядерные процессоры: десятки и сотни мегабайт.
   - Exclusive (`AMD`) и inclusive (`Intel`) стратегии обслуживания.

Спасибо за внимание.

Следующая тема — архитектурные особенности современных компьютеров. Исполнение инструкций.