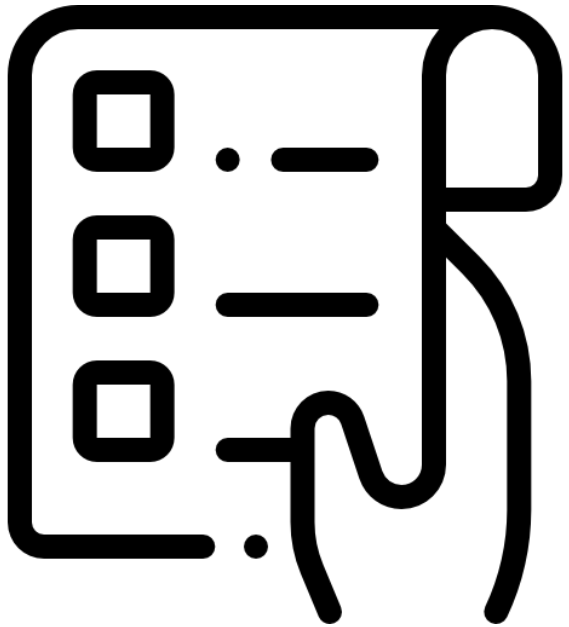
Белорусско-Российский университет
Кафедра «Программное обеспечение информационных технологий»

ЭВМ, периферийные устройства и контроллеры

Тема: Архитектурные особенности организации ЭВМ различных классов

Кутузов Виктор Владимирович

Республика Беларусь, Могилев, 2025



Содержание лекции

Содержание лекции

Тема: Архитектурные особенности организации ЭВМ различных классов

1. Рекомендуемые материалы по теме
2. Архитектура и организация компьютера
3. Классические архитектуры ЭВМ
4. Архитектура Джона фон Неймана
5. Гарвардская архитектура
6. Архитектура фон Неймана vs Гарвардская архитектура
7. Классификация Архитектур ЭВМ.
8. - по организации памяти и кода
9. - по модели параллелизма (классификация Флинна и далее)
10. - по способу организации процессора (микроархитектура)

Содержание лекции

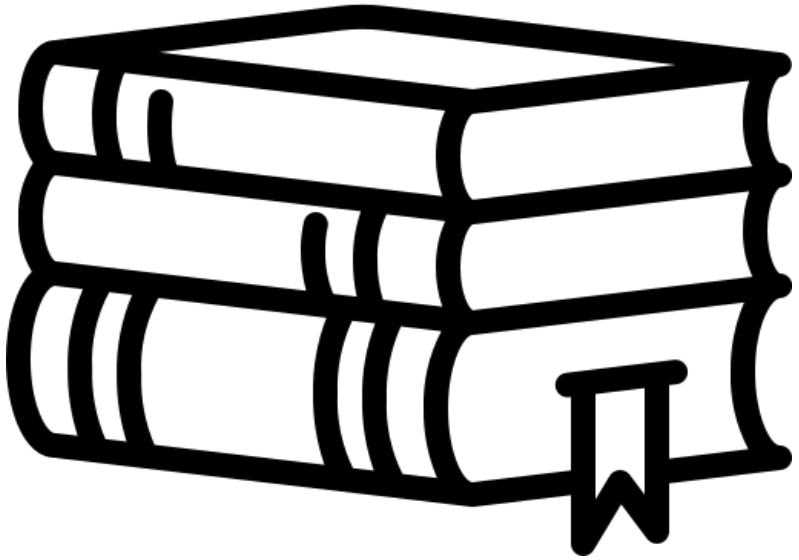
Тема: Архитектурные особенности организации ЭВМ различных классов

11. - по типу набора инструкций (ISA)
12. -- RISC \ CISC архитектуры
13. - по организации данных и вычислений
14. - по памяти и иерархии хранения
15. - по вводу-выводу и шинам
16. - по организации многопроцессорности
17. - по способу адресации и доступу к памяти
18. - по безопасности и изоляции
19. - по устойчивости и надёжности
20. - исторические и учебные архитектуры

Содержание лекции

Тема: Архитектурные особенности организации ЭВМ различных классов

21. - по специализированным областям применения
22. Межкристальные и упаковочные технологии
23. Энергетическая модель и управление питанием
24. Примеры современных архитектур и их применение



Рекомендуемые материалы по теме

Литература

Орлов С. А., Цилькер Б. Я. **Организация ЭВМ и систем**: Учебник для вузов. 2-е изд. – СПб.: Питер, 2011. – 688 с. https://rutracker.org/forum/viewtopic.php?t=5310884

Сара Л. Харрис, Дэвид Харрис **Цифровая схемотехника и архитектура компьютера: RISC-V** / пер. с англ. В. С. Яценкова, А. Ю. Романова; под ред. А. Ю. Романова. – М.: ДМК Пресс, 2021. – 810 с. https://rutracker.org/forum/viewtopic.php?t=6204850

Архитектура и организация компьютера

Архитектура и организация компьютера

- Под архитектурой компьютера в общем обычно понимается набор типов данных, операций и характеристик каждого отдельно взятого уровня ПК.
- Она связана с теми аспектами, что видимы программирующему пользователю соответствующего уровня.
- Например, ресурсы памяти – это аспект архитектуры. А технология реализации запоминающих устройств, физические принципы и механизмы их работы, хотя и являются базовыми, но к архитектуре компьютера в этом смысле не относятся.
- Но при анализе понятия архитектура ПК нужно иметь в виду, что на сегодня разработка компьютера – это не независимая разработка «железа» (hardware) и программного обеспечения (software) по отдельности, но их разработка в едином комплексе.
- Понятие архитектуры актуально в контексте взаимодействия аппаратных и программных компонентов.

Стандартный цикл работы компьютера

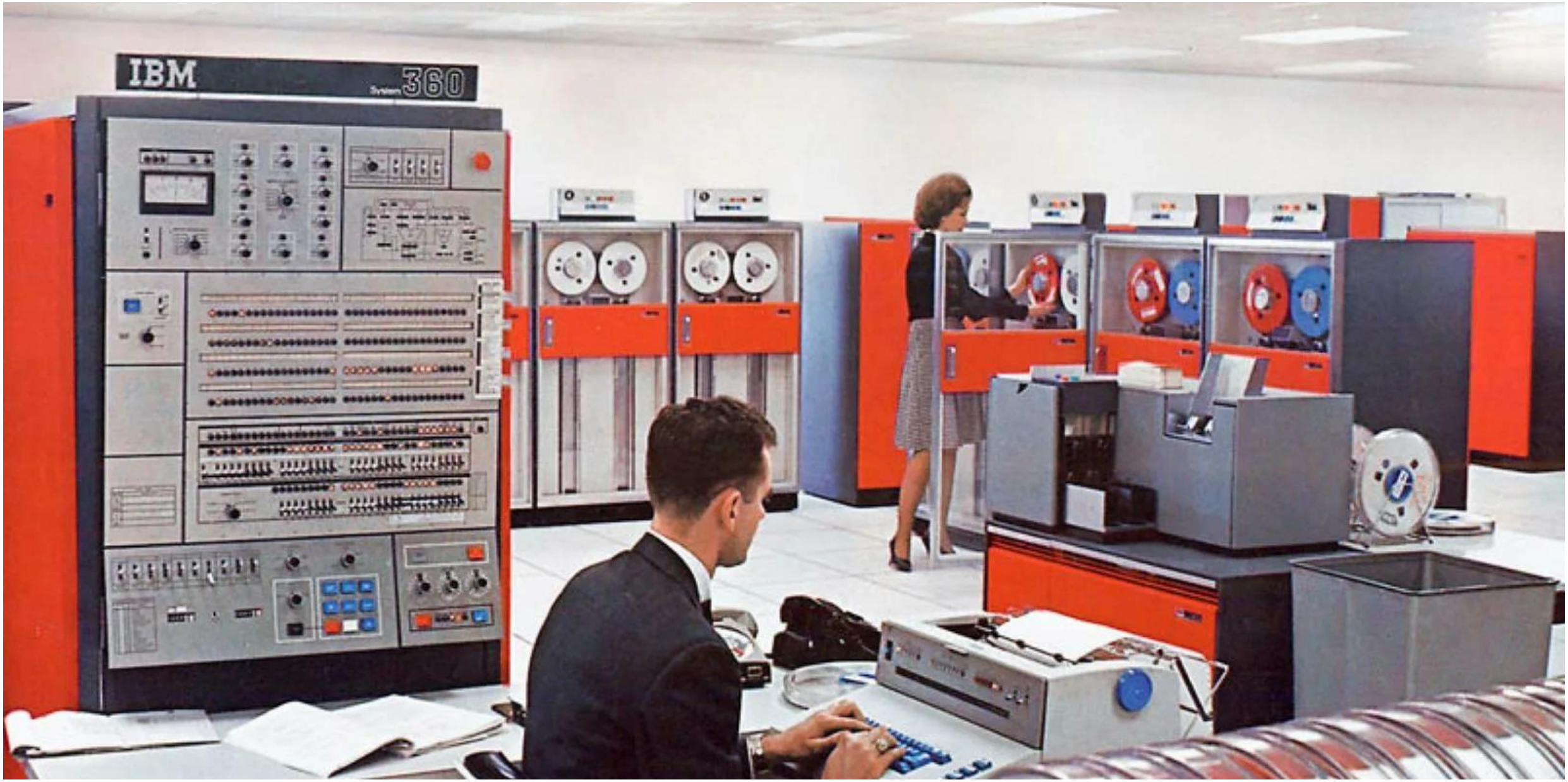
- Можно выделить условный стандартный цикл работы компьютера:
  - 1. выборка инструкции из основной памяти;
  - 2. выборка операндов из оперативной памяти и/или регистров;
  - 3. выполнение инструкции;
  - 4. фиксация результата – запись в память, регистр, вывод.

Архитектура ПК

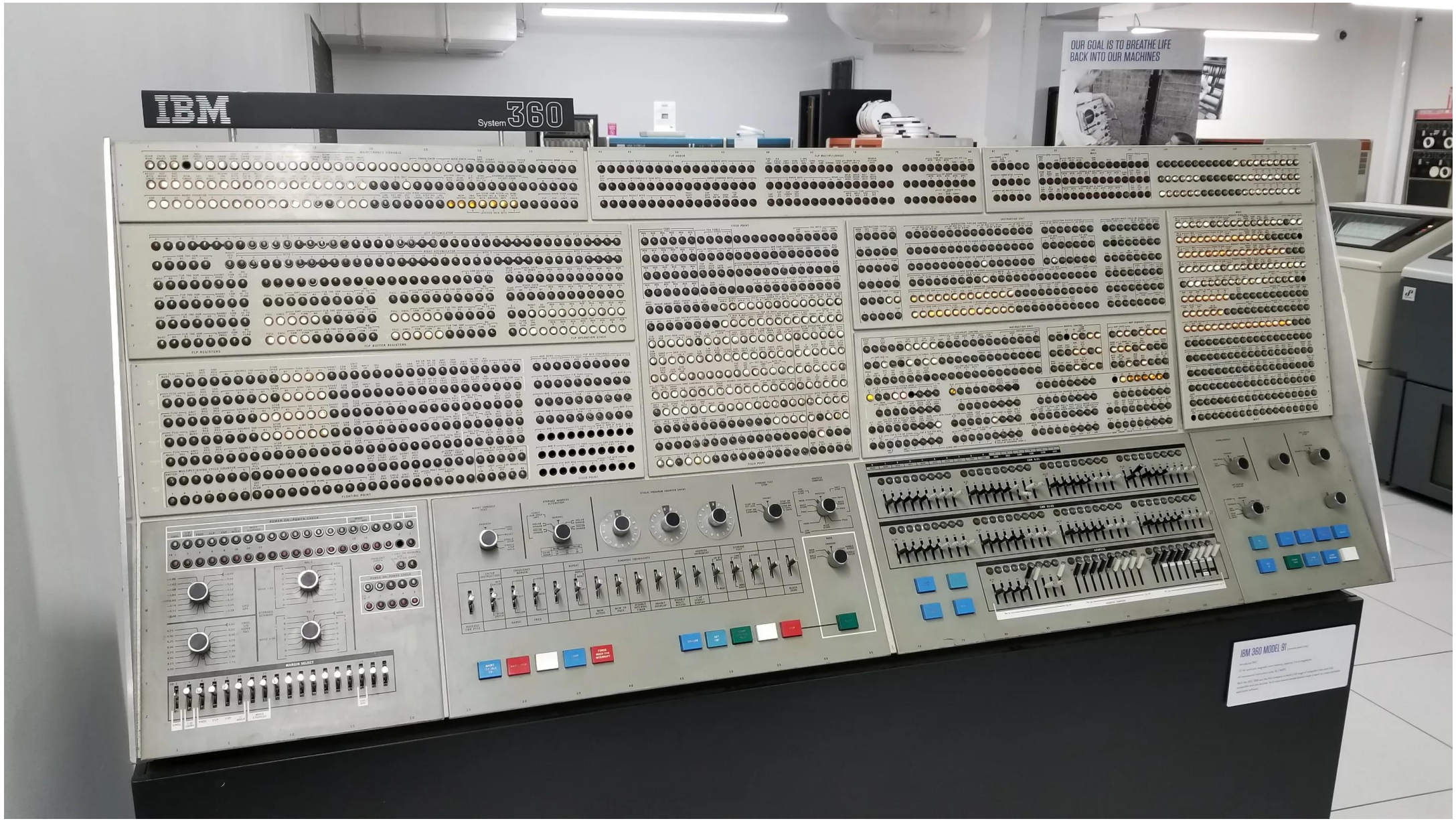
- На сегодня существует ряд трактовок и определений понятия архитектуры.
- **Термин «архитектура» применительно к компьютерам впервые был использован корпорацией IBM в середине 60-х годов при разработке семейства ПК IBM 360.**

- **Под термином «архитектура» часто понимается** представление возможностей компьютера с точки зрения пользователя, разрабатывающего программу на машинно-ориентированном языке.
- **Соответственно, архитектура отражает** те стороны структуры и функционирования компьютера, что существенны и видимы для программиста в контексте разрабатываемых им программ.

IBM System/360



IBM System/360



Архитектура IBM System/360

- **IBM System/360** – это семейство мейнфреймов, представленное IBM в 1964 году.
- Оно стало революционным в истории вычислительной техники, поскольку впервые предложило единую архитектуру для различных по производительности и назначению компьютеров.
- **Проблема, которую оно решало:**
  - **До System/360 IBM выпускала несовместимые линейки компьютеров** (1400-я серия для бизнеса, 7000-я – для науки).
  - Клиенты вынуждены были полностью переписывать ПО при переходе на новую модель.
- **Революционная идея:**
- **IBM System/360 ввела понятие единой архитектуры для всей линейки машин** – от недорогих моделей для учётных задач до суперкомпьютеров. Это означало:
  - Одно и то же ПО работало на всех моделях (с разной скоростью),
  - Аппаратное расширение без замены ПО,
  - Стандартизация интерфейсов и периферии.
- **Название "360": Обозначало "полный круг" – архитектура покрывала все возможные сценарии использования.**

Архитектура ПК. Термины

- **Архитектура** – совокупность характеристик компьютера, существенных с точки зрения пользователя. Можно видеть, что ключевой момент здесь – способ использования компьютера.

- **Архитектура** – совокупность общих принципов организации аппаратно-программных средств и их характеристик, определяющая функциональные возможности вычислителя при решении соответствующих классов задач.

- **Архитектура** – логическое построение вычислителя, как оно представляется программисту, разрабатывающему программу на машинно-ориентированном языке.

- **Архитектура компьютера** – это модель компьютерной системы, воплощённая в её компонентах, их взаимодействии между собой и окружением, включающая также принципы её проектирования и развития. Аспекты реализации (например, технология, применяемая при реализации памяти) не являются частью архитектуры

Архитектура ПК

- **Архитектура связана с аспектами, которые видны программисту.**
  - **Например**, сведения о том, сколько памяти можно использовать при написании программы – часть архитектуры, а аспекты разработки (например, какая технология используется при создании памяти) не является частью архитектуры.
- Изучение того, как разрабатываются те части компьютерной системы, которые видны программистам, называется изучением компьютерной архитектуры.
- Термины компьютерная архитектура и компьютерная организация означают, в сущности, одно и тоже…”.
- **Архитектура может трактоваться** также как раскрытие способов взаимодействия центрального процессора (ЦП) с памятью, периферийными устройствами, окружением и пользователем (интерфейсный подход).

Архитектура ПК

- **Альтернативный подход** основан на противопоставлении понятий «архитектура» и «организация».
- В рамках этого подхода, **архитектура** определяет возможности компьютера, а **организация** – реализацию этих возможностей в конкретных моделях.
- Правомочность такого подхода видна при сравнении моделей, принадлежащих одному семейству – как правило, эти модели обладают одной архитектурой, но разной структурной организацией.

Архитектура ПК

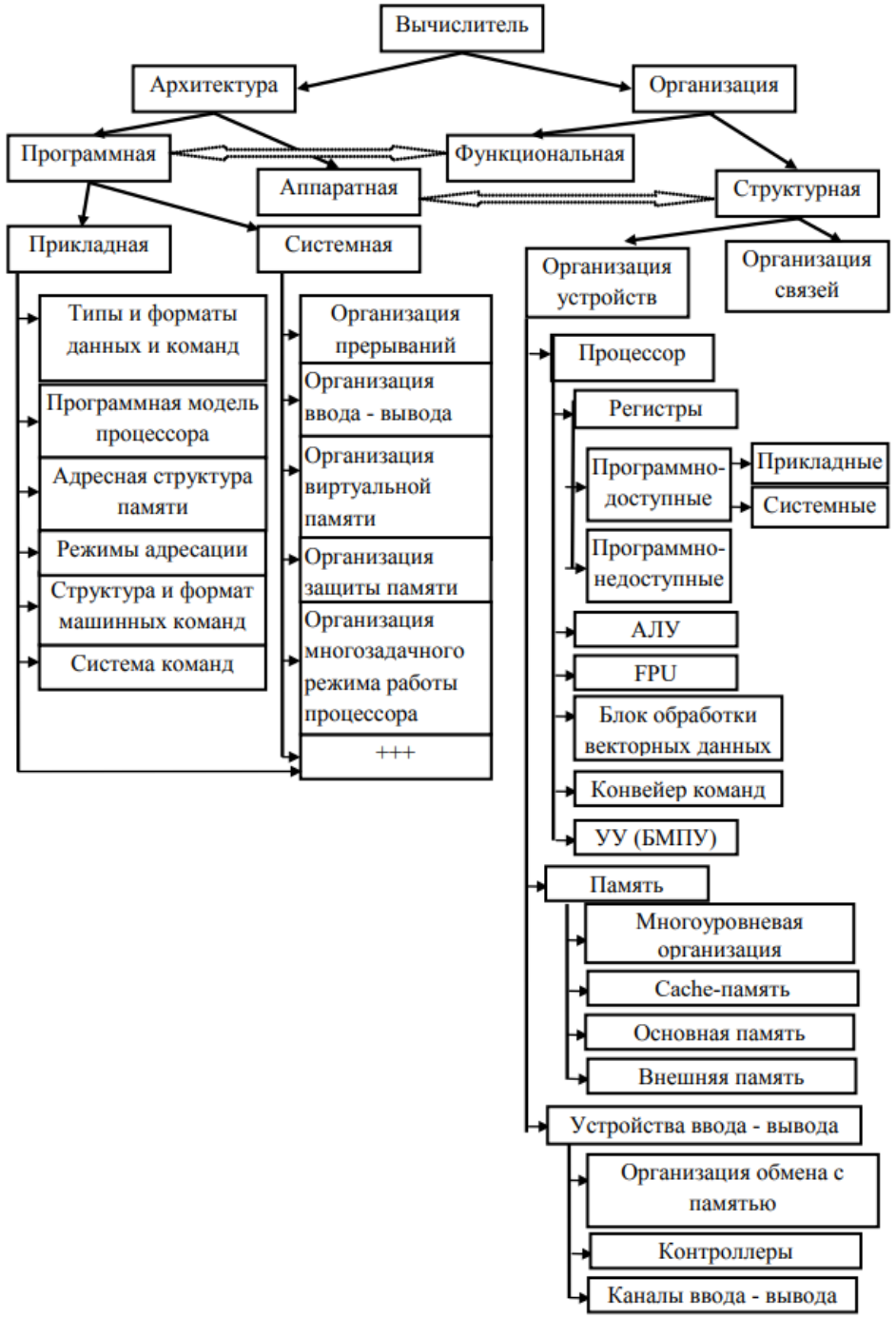
- **При описании компьютерных систем принято различать их структурную организацию и архитектуру.** Хотя точное определение этим понятиям дать довольно трудно, среди специалистов существует общепринятое мнение о смысле этих понятий и различий между ними.
- **Термин архитектура компьютерной системы (компьютера)** относится к тем характеристикам системы, которые доступны извне, то есть со стороны программы или, с другой точки зрения, оказывает непосредственное влияние на логику выполнения программ.
- **Под термином структурная организация компьютерной системы** подразумевается совокупность операционных блоков (устройств) и их взаимосвязей, обеспечивающих реализацию спецификаций, заданных архитектурой компьютера.
- В число характеристик архитектуры входят набор машинных команд, формат разрядной сетки для представления данных разных типов, механизм обращения к средствам ввода/вывода и метод адресации памяти.
- **Характеристики структурной организации** включают скрытые от программиста детали аппаратной реализации системы: управляющие сигналы, аппаратный интерфейс между компьютером и периферийным оборудованием, технологию функционирования памяти”.

Архитектура ПК

- **Два уровня представления архитектуры компьютера:**
  - **программная архитектура** включает в себя аспекты, видимые программистам и, соответственно, программам
  - **аппаратная архитектура** включает аспекты, невидимые для программиста.

- В этом смысле понятие аппаратной архитектуры и структурной организации компьютера можно рассматривать как синонимы.
- В связи с делением программистов на прикладных и системных, программную архитектуру также можно разделить на 2 вида: прикладную и системную.

Основные элементы архитектуры и структурной организации компьютеров



Архитектура ЭВМ

- **Нет одной «единой и окончательной» номенклатуры архитектур ЭВМ.**
- Существуют: термины относятся к разным уровням – от организации памяти и исполнения до параллелизма, ISA и I/O.

- Укрупненно **можно классифицировать архитектуры ЭВМ**, следующим образом:
  1. **По организации памяти и кода**
  2. **По модели параллелизма (классификация Флинна и далее)**
  3. **По способу организации процессора (микроархитектура)**
  4. **По типу набора инструкций (ISA)**
  5. **По организации данных и вычислений**
  6. **По памяти и иерархии хранения**
  7. **По вводу-выводу и шинам**
  8. **По организации многопроцессорности**
  9. **По способу адресации и доступу к памяти**
  10. **По безопасности и изоляции**
  11. **По устойчивости и надёжности**
  12. **Исторические и учебные архитектуры**
  13. **По специализированным областям применения**

Архитектуры процессоров

Классификация архитектур процессоров:

- **По модели исполнения (микроархитектура)**
  - Неконвейерные (single-cycle, multi-cycle), Конвейерные (in-order), Суперскалярные in-order, Out-of-Order (OoO) суперскалярные, Спекулятивное исполнение, предсказание переходов, VLIW/EPIC, CGRA (coarse-grained reconfigurable), Асинхронные/безтактные, GALS, SMT/Hyper-Threading, CMT
- **По параллелизму и формату данных**
  - SISD, SIMD (векторные расширения), SIMT (GPU-подобная модель на процессоре/ускорителе), Матричные/систолические блоки (матрица MAC)
- **По типу ISA (логическая архитектура)**
  - RISC, CISC, VLIW/EPIC, DSP-ориентированные, Стековые/виртуальные, Capability-based
- **По организации ядер и гетерогенности**
  - Однородные многоядерные (CMP), Гетерогенные (big.LITTLE, P/E), Гибрид CPU+специализированные блоки, Многопоточные специализированные
- **По организации памяти и когерентности на уровне процессора/SoC**
  - Классическая фон-неймановская с кэшами, Модифицированная гарвардская, Scratchpad-ориентированные (DSP/RT), Когерентные NoC/CHI в многокристальных CPU
- **По энергетической организации**
  - Статические низкопотребляющие (always-on, MCU), DVFS/Power Gating/Power Islands, Энергетически пропорциональные с гетерогенностью
- **По надежности и реальному времени**
  - Lockstep/TMR ядра, Реальное время (RT)
- **По уровню интеграции и упаковке**
  - Монолитные многокристальные на кристалле (монолит), Чиплетные CPU, 2.5D/3D с HBM/Stacked SRAM
- **По назначению (доменная специализация)**
  - Общего назначения (GP), Встраиваемые/MCU/low-power, DSP/сигнальные/коммуникационные, HPC/серверные, AI/NPU/DSA
- **По модели согласования памяти (memory consistency) для программирования**
  - Сильные: TSO (x86), Слабые/релаксированные: ARM/RISC-V (с барьерами), POWER, Гетерогенные с SVM/UVM

Архитектура ЭВМ VS Архитектура процессора

- **Что такое архитектура ЭВМ?**

- **Это совокупность принципов организации вычислительной системы целиком**: модель программирования (ISA/ABI), память и её иерархия, ввод-вывод, параллелизм, адресное пространство, способы взаимо-действия компонентов, требования к надежности/ безопасности.
- Иначе: что видит и чем пользуется программист и системный инженер при работе с системой в целом.

- **Что такое архитектура процессора?**

- **Это организация и принципы построения самого процессорного ядра (или набора ядер):** исполнение команд, конвейер, суперскалярность, OoO, регистровый файл, кэш L1, механизмы предсказания ветвлений, декодер, микрокод и т. п.
- Часто отделяют ISA (логический интерфейс команд) от микроархитектуры (конкретная реализация процессора).
- В разговорной речи «архитектура процессора» иногда означает либо ISA (ARM, x86, RISC-V и т.д.), либо микроархитектуру (Zen, Golden Cove и др.).

Архитектура ЭВМ VS Архитектура процессора

- **Чем схожи**
  - **Обе описывают абстракции и ограничения**, определяющие, какие программы и с какими свойствами можно выполнять.
  - **Оба уровня влияют на производительность, энергоэффективность, стоимость и надежность системы**.

- **Чем различаются**
  - **Объем:** архитектура ЭВМ шире – включает память, шины/фабрики, I/O, многопроцессорность, безопасность, виртуализацию, взаимодействие с ОС. Архитектура процессора – один компонент этой системы.
  - **Видимость:** архитектура ЭВМ определяет системную модель (например, NUMA, когерентность, устройство прерываний); архитектура процессора – как отдельное ядро исполняет инструкции (pipeline, ROB, ALU и др.).
  - **Инвариантность:** одну и ту же архитектуру ЭВМ можно построить на разных процессорных архитектурах, и наоборот: один ISA может быть встроен в разные ЭВМ (SMP, NUMA, SoC с разным I/O).

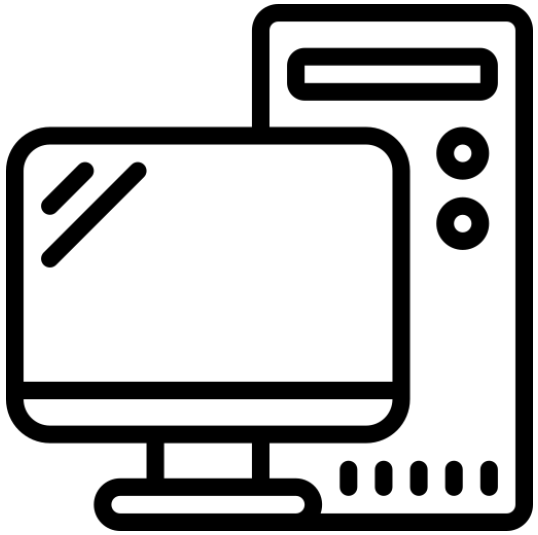
Архитектура ЭВМ VS Архитектура процессора

- **Архитектура ЭВМ** – системная организация всего компьютера (модель памяти, I/O, параллелизм, шины, когерентность).

- **Архитектура процессора** – организация исполнения инструкций внутри ядра(ядер) и близкой памяти, иногда также означает ISA.

- **Они связаны, но уровни разные: процессор – часть ЭВМ**; выбор архитектуры процессора определяет свойства системы, но не исчерпывает её.

Рассмотрим детальнее отдельные виды архитектур ЭВМ (классические архитектуры: архитектура фон Неймана и Гарвардская архитектура, архитектуры процессора: CISC и RISC)**, укрупненно пройдемся по классификации архитектур и отдельно разберем в них некоторые архитектуры**.

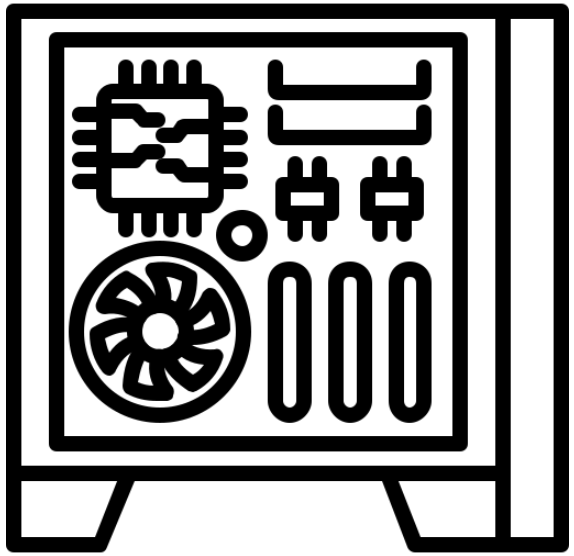


Классические архитектуры ЭВМ

Классические архитектуры ЭВМ

- **Классические архитектуры ЭВМ** — это фундаментальные модели построения вычислительных систем, заложившие основы современных компьютеров. Они определяют структуру, принципы взаимодействия компонентов и способы обработки данных.
- Наиболее значимыми считаются **архитектура фон Неймана** и **Гарвардская архитектура**.

- **Кроме данных архитектур, можно выделить:**
  - Архитектура с конвейеризацией (pipeline)
  - Суперскалярные и внеочередные (out-of-order) архитектуры
  - CISC (Complex Instruction Set Computing) и RISC (Reduced Instruction Set Computing) архитектуры
  - SISD, SIMD, MISD, MIMD (классификация Флинна) архитектуры
  - Архитектуры с разделяемой и распределённой памятью
  - и многие, многие другие



Архитектура Джона фон Неймана

Архитектура фон Неймана

- **В 1945 г., Джон фон Нейман** выделил и детально описал ключевые компоненты того, что сегодня называют «**Архитектурой фон Неймана**» современного компьютера.

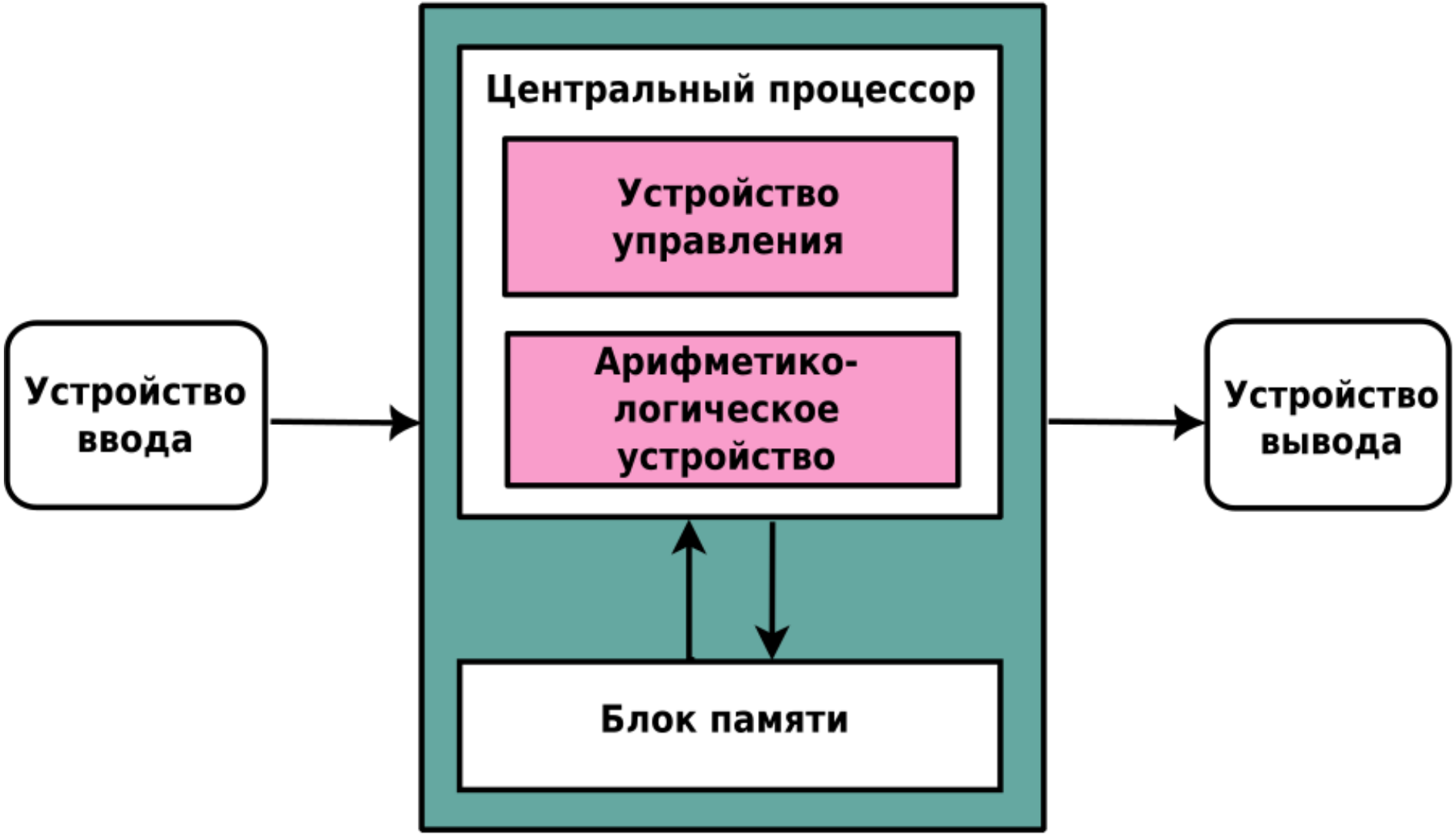
- **Чтобы компьютер был и эффективным, и универсальным инструментом, он должен включать следующие структуры:**
  - **1)** арифметико-логическое устройство (АЛУ), выполняющее арифметические и логические операции;
  - **2)** устройство управления, организующее процесс выполнения программ;
  - **3)** запоминающее устройство или память для хранения программ и данных;
  - **4)** устройство ввода-вывода информации.

Джон фон Нейман
(1903-1957)

Архитектура фон Неймана

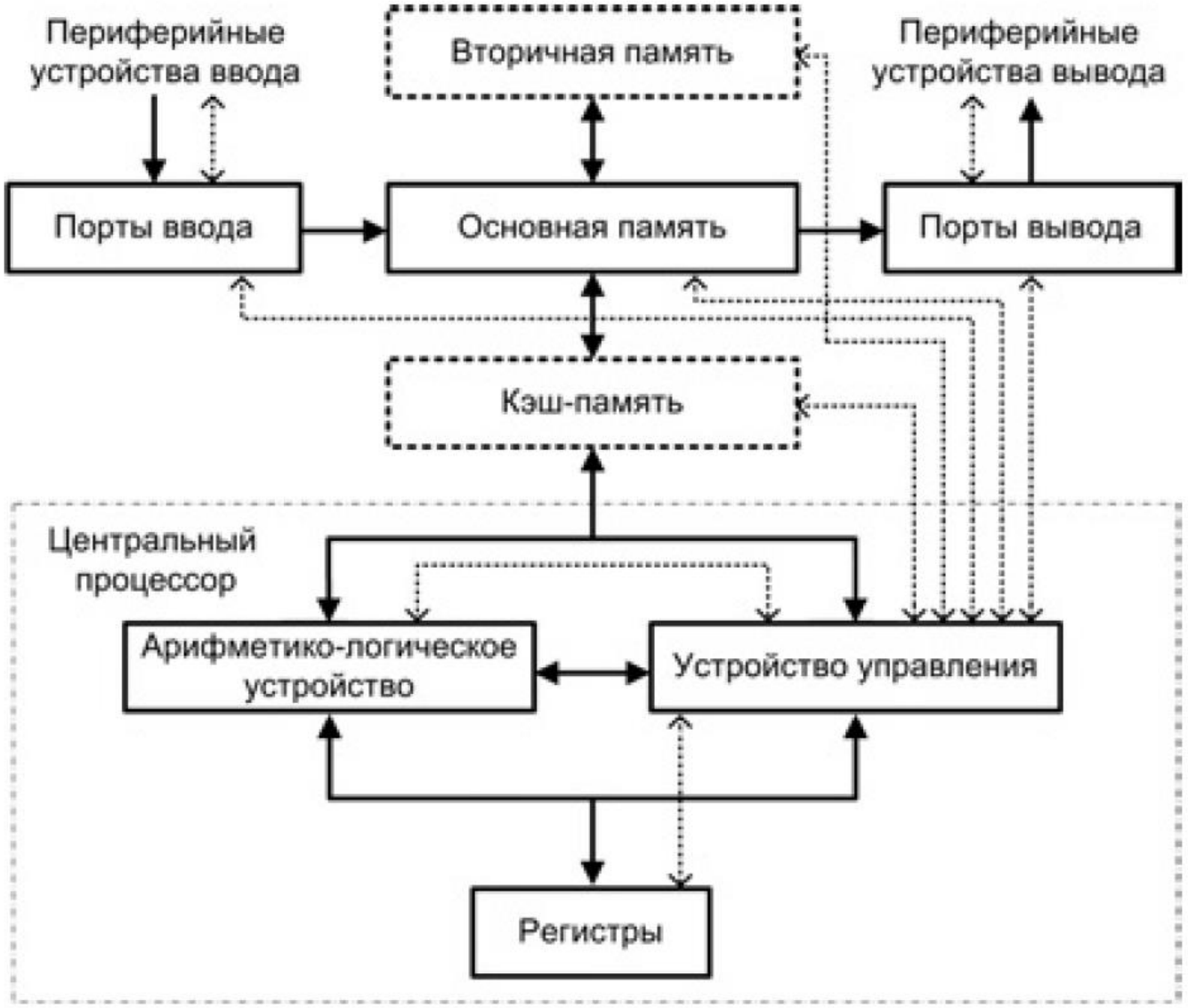
- **Архитектура фон Неймана** – это базовая модель организации современных компьютеров, где программы и данные хранятся в одной памяти, а процессор последовательно извлекает и выполняет команды.
- **Архитектура фон Неймана** – фундаментальная модель построения вычислительных систем, заложившая основы современных компьютеров. Её принципы, сформулированные в 1945 году, до сих пор лежат в основе подавляющего большинства универсальных ЭВМ.
- **Она состоит из процессора** (состоящего из устройства управления и арифметико-логического устройства), **оперативной памяти, устройств ввода/вывода и шин**, связывающих их.
- **Эта универсальная модель** обусловила широкое распространение, хотя и имеет недостаток в виде ограниченной скорости из-за **общего доступа к памяти** ("**бутылочное горлышко**").

Архитектура фон Неймана



Схематичное изображение архитектуры компьютера фон Неймана (память, УУ, АЛУ, аккумулятор, ввод-вывод)

Архитектура фон Неймана



Расширенная структура фон-Неймановской вычислительной машины

Архитектура фон Неймана

Архитектура фон Неймана
Принципы фон Неймана

1. Принцип двоичного кодирования
2. Принцип программного управления
3. Принцип однородности памяти или принцип хранимой программы
4. Принцип адресности
5. Принцип иерархичности запоминающих устройств
6. Принцип параллельный организации вычислительного процесса

Архитектура фон Неймана
Принципы фон Неймана

1. Принцип двоичного кодирования.

2. Принцип программного управления работой электронно-вычислительной машины. Программы состоят из набора команд, которые выполняются процессором автоматически друг за другом в определенной последовательности.

3. Принцип однородности памяти или принцип хранимой программы. Программы и данные хранятся в одной и той же памяти. Поэтому ЭВМ не различает, что хранится в данной ячейке памяти – числа, текст или команда. Над командами можно выполнять такие же действия, как и над данными. Это позволяет создавать программы, способные в процессе вычислений изменять самих себя.

Архитектура фон Неймана
Принципы фон Неймана

4. Принцип адресности. Структурно основная память состоит из пронумерованных ячеек. Процессору в произвольный момент времени доступна любая ячейка.

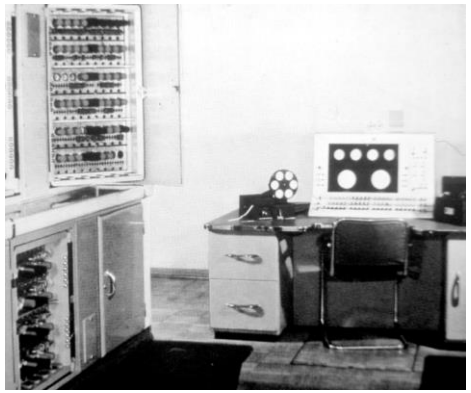
5. Принцип иерархичности запоминающих устройств. Наиболее часто используемые данные хранятся в самом быстром запоминающем устройстве сравнительно малой емкости, а более редко используемые – в самом медленном, но гораздо большей емкости.

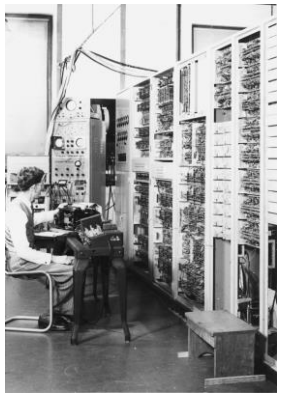
6. Принцип параллельный организации вычислительного процесса: операции над словами производятся одновременно во всех разрядах слова.

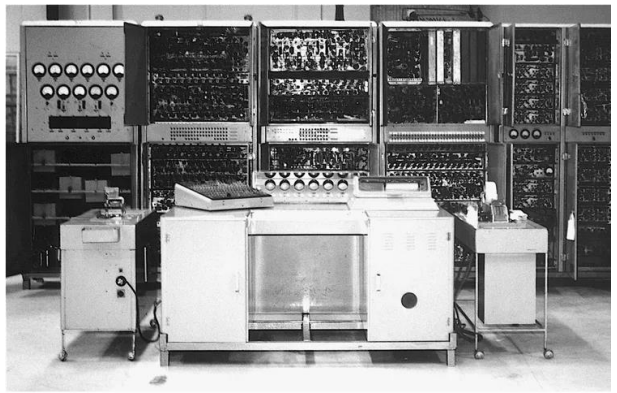
Архитектура фон Неймана

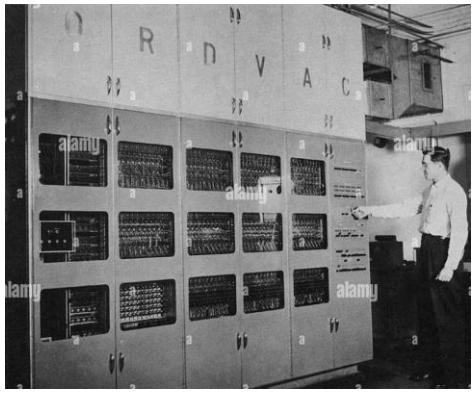
- **Первыми компьютерами, в которых были реализованы основные особенности архитектуры фон Неймана**, были:
  1. прототип — **Манчестерская малая экспериментальная машина** — Манчестерский университет, Великобритания, 21 июня 1948 года;
  2. **EDSAC** — Кембриджский университет, Великобритания, 6 мая 1949 года;
  3. **Манчестерский Марк I** — Манчестерский университет, Великобритания, 1949 год;
  4. **BINAC** — США, апрель или август 1949 года;
  5. **CSIR Mk 1** — Австралия, ноябрь 1949 года;
  6. **EDVAC** — США, август 1949 года — фактически запущен в 1952 году;
  7. **CSIRAC** — Австралия, ноябрь 1949 года;
  8. **SEAC** — США, 9 мая 1950 года;
  9. **ORDVAC** — США, ноябрь 1951 года;
  10. **IAS-машина** — США, 10 июня 1952 года;
  11. **MANIAC I** — США, март 1952 года;
  12. **AVIDAC** — США, 28 января 1953 года;
  13. **ORACLE** — США, конец 1953 года;
  14. **WEIZAC** — Израиль, 1955 год;
  15. **SILLIAC** — Австралия, 4 июля 1956 года.
- **В СССР первой полностью электронной вычислительной машиной, близкой к принципам фон Неймана, стала МЭСМ**, построенная Лебедевым (на базе киевского Института электротехники АН УССР). МЭСМ как прототип впервые была публично запущена 6 ноября 1950 года и уже в качестве полноценной машины прошла государственные приёмочные испытания 25 декабря 1951 года.

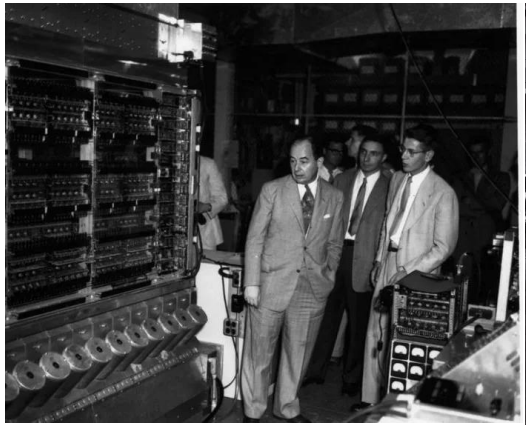
Архитектура фон Неймана

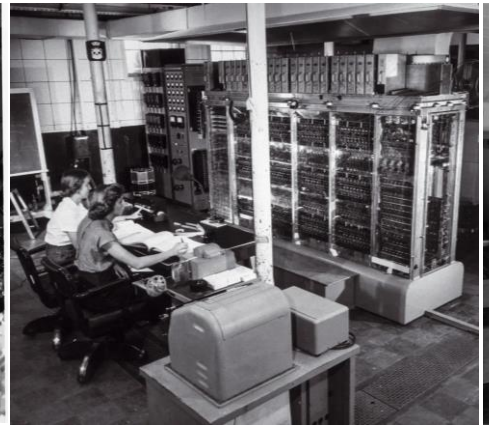


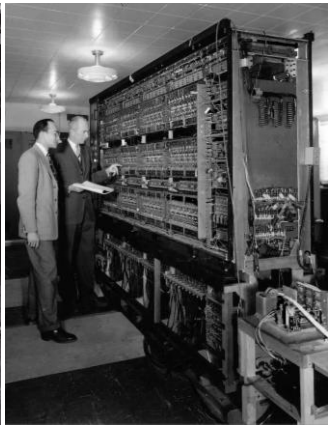


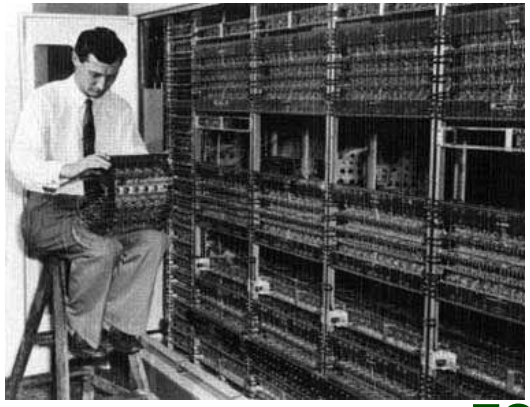




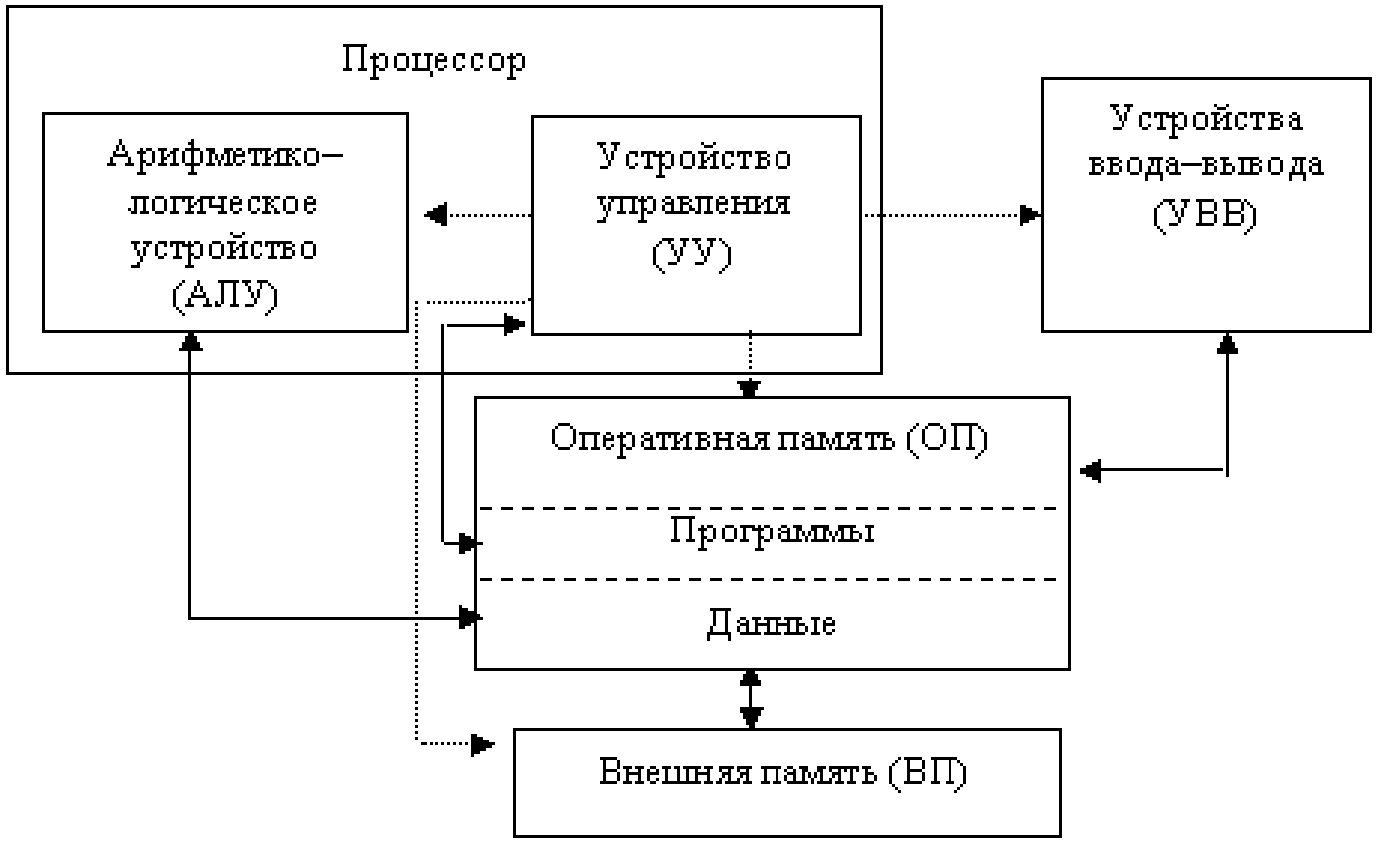








Классическая структурная схема ЭВМ



Классическая структурная схема ЭВМ, построенной в соответствии с принципами фон Неймана.

- **На схеме представлены:**
- **память** (оперативная и внешняя), состоящая из перенумерованных ячеек (номер ячейки называют адресом);
- **процессор**, включающий в себя устройство управления (УУ) и арифметико–логическое устройство (АЛУ); устройство ввода;
- **устройство вывода**.
- Эти устройства соединены каналами связи, по которым передается информация (управляющая – прерывистые стрелки, данные – непрерывные стрелки).

Преимущества архитектуры

- Архитектура фон Неймана, несмотря на возраст и известные ограничения, остаётся **основой подавляющего большинства современных процессоров** – от смартфонов и ноутбуков до серверов и суперкомпьютеров.
- Это не случайность, а результат сочетания её **фундаментальных преимуществ**, которые делают её **универсальной, гибкой, масштабируемой и экономически эффективной**.

- Подробно разоберем ключевые преимущества и причины, по которым современные процессоры всё ещё строятся на её основе.

Преимущества архитектуры

- **1. Универсальность (принцип хранимой программы)**

«Один и тот же компьютер может решать любую задачу – достаточно изменить программу».

- Это **главное преимущество**, изменившее всю историю вычислений.
- До фон Неймана машины были жёстко запрограммированы – для смены задачи требовалось переключать провода или менять перфокарты.
- Архитектура фон Неймана ввела идею, что **программа – это данные**, которые можно хранить в памяти и менять динамически.

- **Результат:** Появление **операционных систем, интерпретаторов, компиляторов, виртуальных машин** – всего того, что делает современные компьютеры универсальными инструментами.

Преимущества архитектуры

- **2. Простота и ясность структуры**

«ЦПУ + память + шина + ввод/вывод» – легко понять, спроектировать и масштабировать.

- Архитектура предлагает **понятную и логичную модель**:
  - Процессор выбирает команду из памяти → декодирует → выполняет → переходит к следующей.
  - Все компоненты связаны через **системную шину**.
  - Память едина и адресуема.

- **Результат:** Упрощает обучение, проектирование процессоров, разработку компиляторов и отладку. Это идеальная модель для образования и базового понимания работы компьютера.

Преимущества архитектуры

- **3. Гибкость и динамическое поведение**

«Программа может менять себя в процессе выполнения».

- Поскольку код и данные хранятся в одной памяти и неотличимы на аппаратном уровне, программа может:
  - Генерировать новые команды на лету (JIT-компиляция, самоизменяющийся код).
  - Загружать модули динамически (DLL, .so).
  - Реализовывать рекурсию, полиморфизм, виртуальные функции.

- **Результат:** Поддержка сложных парадигм программирования (ООП, функциональное программирование, метапрограммирование).

Преимущества архитектуры

- **4. Экономическая и экосистемная эффективность**

«Инвестиции в ПО, ОС, инструменты, обучение – окупаются десятилетиями».

- Архитектура фон Неймана создала **гигантскую совместимую экосистему:**
  - Миллиарды строк кода, написанных под x86, ARM, RISC-V.
  - Сотни операционных систем (Windows, Linux, macOS, Android, iOS).
  - Тысячи компиляторов, отладчиков, виртуальных машин (JVM, .NET CLR).

- **Результат:** Переход на радикально новую архитектуру потребовал бы **колоссальных затрат и полной перезаписи всего ПО** – экономически нецелесообразно.

Преимущества архитектуры

- **5. Масштабируемость и адаптивность**

«Можно добавлять ядра, кэши, предсказатели, SIMD – не меняя основу».

- Хотя «узкое место фон Неймана» существует, инженеры научились **обходить его**, не отказываясь от архитектуры:
  - **Кэш-память (L1, L2, L3)** – уменьшает задержки доступа к данным.
  - **Конвейеризация** – позволяет выполнять несколько команд одновременно на разных стадиях.
  - **Суперскалярность** – несколько АЛУ в одном ядре.
  - **Предсказание переходов** – уменьшает простои при ветвлениях.
  - **Многоядерность** – параллелизм на уровне ядер.
  - **SIMD-инструкции** (AVX, Neon) – параллельная обработка данных.
- **Результат**: Современные процессоры – это **гибридные системы**, где фон-неймановская основа дополняется десятками механизмов, смягчающих её недостатки.

Преимущества архитектуры

- **6. Поддержка абстракций высокого уровня**

«Архитектура позволяет строить виртуальные машины, контейнеры, облачные среды».

- Благодаря единообразию памяти и управляемому выполнению команд, **легко реализуются**:
  - **Виртуализация** (гипервизоры, VM).
  - **Контейнеризация** (Docker, Kubernetes).
  - **Управляемые среды** (Java, Python, .NET).
  - **Защита памяти** (MMU, страничная адресация).

- **Результат**: Современная ИТ-инфраструктура – от облаков до мобильных приложений – построена на этих абстракциях, которые невозможны без фон-неймановской основы.

Недостаток архитектуры

- Архитектура фон Неймана, несмотря на свою универсальность и доминирование в вычислительной индустрии, имеет **фундаментальные недостатки**, которые становятся всё более критичными по мере роста требований к производительности, энергоэффективности и безопасности.
- Эти недостатки не просто технические мелочи – они **встроены в саму концепцию архитектуры** и потому не могут быть полностью устранены без перехода к иным моделям вычислений.

Недостаток архитектуры

- **Главный недостаток: «Узкое место фон Неймана» (von Neumann Bottleneck)**

«Процессор быстрый, а память – медленная. И они соединены одной шиной»

- Это – центральная и самая известная проблема архитектуры.
- **Суть проблемы:**
- Все команды и данные передаются между процессором и памятью **по одной и той же системной шине**. Процессор не может одновременно:
  - читать команду из памяти,
  - и читать/писать данные.
- Это создаёт **очередь на шине**, и процессор вынужден **ждать данные**, простаивая десятки или сотни тактов.
- **Последствия:**
  - **Ограничение производительности:** Даже самый быстрый CPU упирается в пропускную способность и задержки памяти.
  - **Разрыв скоростей (Memory Wall):** Современные процессоры работают на частотах 3–5 ГГц (цикл ~0.2–0.3 нс), а доступ к DRAM занимает 50–150 нс – за это время CPU мог бы выполнить сотни инструкций.
  - **Энергетические потери:** Перемещение данных между CPU и памятью потребляет больше энергии, чем сами вычисления.
- **Пример**: В задачах машинного обучения или обработки больших данных CPU тратит до 80–90% времени на ожидание данных из памяти.

Недостаток архитектуры

- **Другие ключевые недостатки**
- **1. Отсутствие разделения кода и данных в памяти**

«Для процессора нет разницы между числом 42 и командой ADD»

- Это следствие принципа однородности памяти – и данные, и команды хранятся в одной памяти и представлены одинаково.
- **Проблемы:**
  - Уязвимости безопасности: Вредоносный код может перезаписать участок памяти с командами → выполнение произвольного кода (например, buffer overflow, ROP-атаки).
  - Сложность защиты: Требуются дополнительные механизмы (NX-бит, DEP, ASLR) для предотвращения исполнения данных как кода – но это «заплатки», а не решение на архитектурном уровне.
  - Ошибки программирования: Случайная запись в область памяти с кодом может привести к краху всей системы.
- **Примеры уязвимостей:** Spectre, Meltdown, Heartbleed – все они эксплуатируют особенности фон-неймановской модели.

Недостаток архитектуры

- **2. Последовательное выполнение команд**

«Выполняй одну команду за другой – даже если они не зависят друг от друга»

- Хотя современные процессоры используют конвейеризацию, суперскалярность и спекулятивное выполнение, на логическом уровне программа всё ещё воспринимается как линейная последовательность инструкций.
- **Проблемы:**
  - Сложность параллелизма: Программист должен явно указывать, какие части кода можно выполнять параллельно (многопоточность, OpenMP, CUDA).
  - Зависимости данных: Команды часто ждут результатов предыдущих операций → простой конвейера.
  - Накладные расходы на управление потоками: Переключение контекста, синхронизация, блокировки – всё это снижает эффективность.
- **Пример**: Даже в 16-ядерном процессоре одна нитка, зависящая от результата другой, может простаивать, пока данные не придут.

Недостаток архитектуры

- **3. Сложности с многозадачностью и виртуализацией**

«Как запустить несколько программ, чтобы они не мешали друг другу?»

- В «чистой» фон-неймановской модели нет механизмов:
  - Разделения адресного пространства.
  - Защиты памяти.
  - Прерываний и переключения контекста.
- **Что пришлось добавить «сверху»:**
  - **MMU** (Memory Management Unit) – для виртуальной памяти и защиты.
  - **Прерывания и планировщик ОС** – для переключения задач.
  - **TLB** (Translation Lookaside Buffer) – для ускорения преобразования адресов.
- Это увеличивает сложность железа и ПО, а также снижает производительность из-за накладных расходов.

Недостаток архитектуры

- **4. Низкая энергоэффективность для массовых вычислений**

«Двигать данные – дороже, чем считать»

- В фон-неймановской архитектуре большая часть энергии тратится не на вычисления, а на:
  - Передачу данных по шине.
  - Обращения к кэшам и памяти.
  - Управление конвейером и предсказанием переходов.
- **Последствия:**
  - Для мобильных устройств и дата-центров – это критично.
  - Для задач ИИ, где нужно обрабатывать гигабайты данных – крайне неэффективно.
- **Статистика**: В GPU и AI-ускорителях до 70–90% энергии уходит на перемещение данных, а не на умножение матриц.

Недостаток архитектуры

- **5. Кризис масштабируемости**

«Добавить ещё ядер – не значит стать быстрее»

- Закон Мура замедляется. Увеличение тактовой частоты упёрлось в **энергетическую стену**. Основной путь роста – **многоядерность**.
- **Проблемы**:
  - Все ядра делят одну память → конкуренция за шину и кэш.
  - Сложность синхронизации и согласованности кэшей (протоколы MESI и т.п.).
  - Программы не всегда можно распараллелить → закон Амдала ограничивает прирост.
- **Пример**: 64-ядерный сервер может простаивать, если приложение не умеет эффективно использовать все ядра.

Почему современные процессоры всё ещё используют архитектуру фон Неймана?

- **1. Обратная совместимость – священная корова индустрии**
- **«Если сломать совместимость – потеряешь рынок».**
  - **Пример**: Intel с 1978 года (8086) поддерживает обратную совместимость. Даже современные процессоры Core i9 могут запустить DOS-программу 1980-х (в режиме эмуляции). ARM тоже сохраняет совместимость между поколениями.
- **Последствие:** Переход на новую архитектуру = потеря всей существующей базы ПО = крах бизнеса.

- **2. Эволюция, а не революция**
- **«Не нужно изобретать велосипед – можно улучшать существующий».**
- Современные процессоры – это **не «чистые» фон-неймановские машины, а высокооптимизированные гибриды**:
  - Внутри ядра – гарвардская структура (отдельные кэши команд и данных).
  - Вне ядра – фон-неймановская шина (единая память).
  - Используются внеархитектурные методы: спекулятивное выполнение, OoO (out-of-order), SMT (hyper-threading).
- **Пример:** Процессор Apple M2 или Intel Core i9 – это фон Нейман «сверху», но внутри – сложнейшая система, обходящая его ограничения.

Почему современные процессоры всё ещё используют архитектуру фон Неймана?

- **3. Отсутствие зрелой альтернативы**
- **«Dataflow, нейроморфные, квантовые – пока не готовы заменить массовые процессоры».**
- Хотя активно развиваются альтернативы:
  - **Гарвардская архитектура** – используется в микроконтроллерах (AVR, PIC), но не подходит для универсальных ПК.
  - Dataflow / потоковые архитектуры – хороши для AI, но сложны в программировании.
  - **Нейроморфные чипы** (Loihi, SpiNNaker) – экспериментальны, не поддерживают ОС.
  - **Квантовые компьютеры** – не заменяют классические, а дополняют их.
- **Вывод**: Пока нет архитектуры, которая бы универсально, эффективно и совместимо заменила фон Неймана.

- **4. Программисты и инструменты «привыкли» к последовательности**
- **«Большинство языков и алгоритмов написаны под последовательную модель».**
- C, C++, Java, Python, JavaScript – все они предполагают **последовательное выполнение инструкций**. Переход на dataflow или параллельные модели требует **кардинального переобучения разработчиков** и переписывания методик.
- **Пример**: Даже в многопоточных программах программист думает в терминах «потоков», а не «потоков данных».

Будущее: фон Нейман не умрёт – он трансформируется

- Современные и будущие процессоры будут всё больше **отходить от «чистой» фон-неймановской модели, но сохраняя её как внешний интерфейс:**
  - **Гетерогенные системы**: CPU (фон Нейман) + GPU (dataflow) + NPU (нейроморфный) + FPGA (реконфигурируемый).
  - **Память-вычисления** (Processing-in-Memory, PIM): Вычисления прямо в памяти – частичное преодоление «узкого места».
  - **Аппаратная поддержка ИИ**: Тензорные ядра, матричные ускорители – расширяют, но не заменяют CPU.

- **Фон Нейман – это не железо, это идея.** И эта идея пока незаменима.
- **Архитектура фон Неймана – это не «устаревшая модель», а «живая основа», на которую наложены десятки слоёв оптимизаций.** Она выжила не потому, что идеальна, а потому, что достаточно хороша, гибка и поддерживает гигантскую экосистему. Её эволюция – лучший пример того, как инженеры превращают теоретические ограничения в практические решения.

Вывод

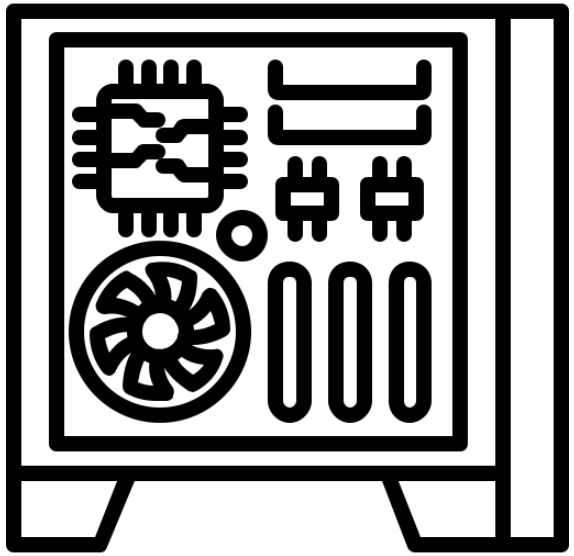
- **Архитектура фон Неймана — это фундамент, на котором построена вся современная вычислительная индустрия.** Предложенная в середине XX века, она ввела революционную идею хранимой программы, объединив код и данные в едином адресном пространстве памяти. Это позволило создать универсальные машины, способные решать любые задачи за счёт смены программного обеспечения, а не перестройки аппаратуры. Благодаря своей простоте, логичности и гибкости, архитектура быстро стала стандартом, вокруг которого выросла гигантская экосистема — от языков программирования и операционных систем до компиляторов и облачных платформ. Её принципы до сих пор лежат в основе процессоров Intel, AMD, ARM и RISC-V, что делает её, пожалуй, самой успешной и долгоживущей архитектурной моделью в истории технологий.

- **Однако за универсальность и гибкость приходится платить фундаментальными ограничениями.** Главный недостаток — «узкое место фон Неймана» — возникает из-за того, что команды и данные передаются по одной шине, вынуждая быстрый процессор простаивать в ожидании медленной памяти. К этому добавляются проблемы безопасности (отсутствие разделения кода и данных), сложности с эффективным параллелизмом, высокое энергопотребление на перемещение информации и трудности масштабирования. Эти ограничения не являются следствием устаревания технологии — они заложены в саму концепцию архитектуры и становятся всё более критичными в эпоху искусственного интеллекта, больших данных и мобильных устройств, где важны скорость, энергоэффективность и безопасность.

Вывод

- **Современные процессоры — это не «чистые» фон-неймановские машины, а сложные гибриды, которые эволюционно адаптировались к вызовам времени.** Инженеры научились обходить ограничения архитектуры: многоуровневые кэши уменьшают задержки доступа к данным, конвейеризация и спекулятивное выполнение повышают производительность, а многоядерность и SIMD-инструкции добавляют параллелизм. Внутри процессора часто используется гарвардская структура (раздельные кэши команд и данных), а снаружи — сохраняется совместимость с фон-неймановской моделью. Такой подход позволяет сохранять обратную совместимость и не переписывать триллионы строк существующего ПО, одновременно повышая эффективность систем.

- **Будущее вычислений, скорее всего, будет не в полном отказе от архитектуры фон Неймана, а в её постепенном дополнении и трансформации.** Мы уже видим, как CPU сосуществует с GPU, NPU, FPGA и другими специализированными ускорителями, каждый из которых использует свои архитектурные принципы. Появляются технологии вроде Processing-in-Memory (PIM), нейроморфных и потоковых вычислений, которые стремятся преодолеть «узкое место», интегрируя память и вычисления. Архитектура фон Неймана останется основой для общих задач, но для специализированных, высокопроизводительных или энергоэффективных приложений будут доминировать новые модели. Таким образом, фон Нейман — это не пережиток прошлого, а живая, адаптивная основа, которая продолжает развиваться, уступая место новым идеям там, где её ограничения становятся непреодолимыми.



Гарвардская архитектура

Гарвардская архитектура

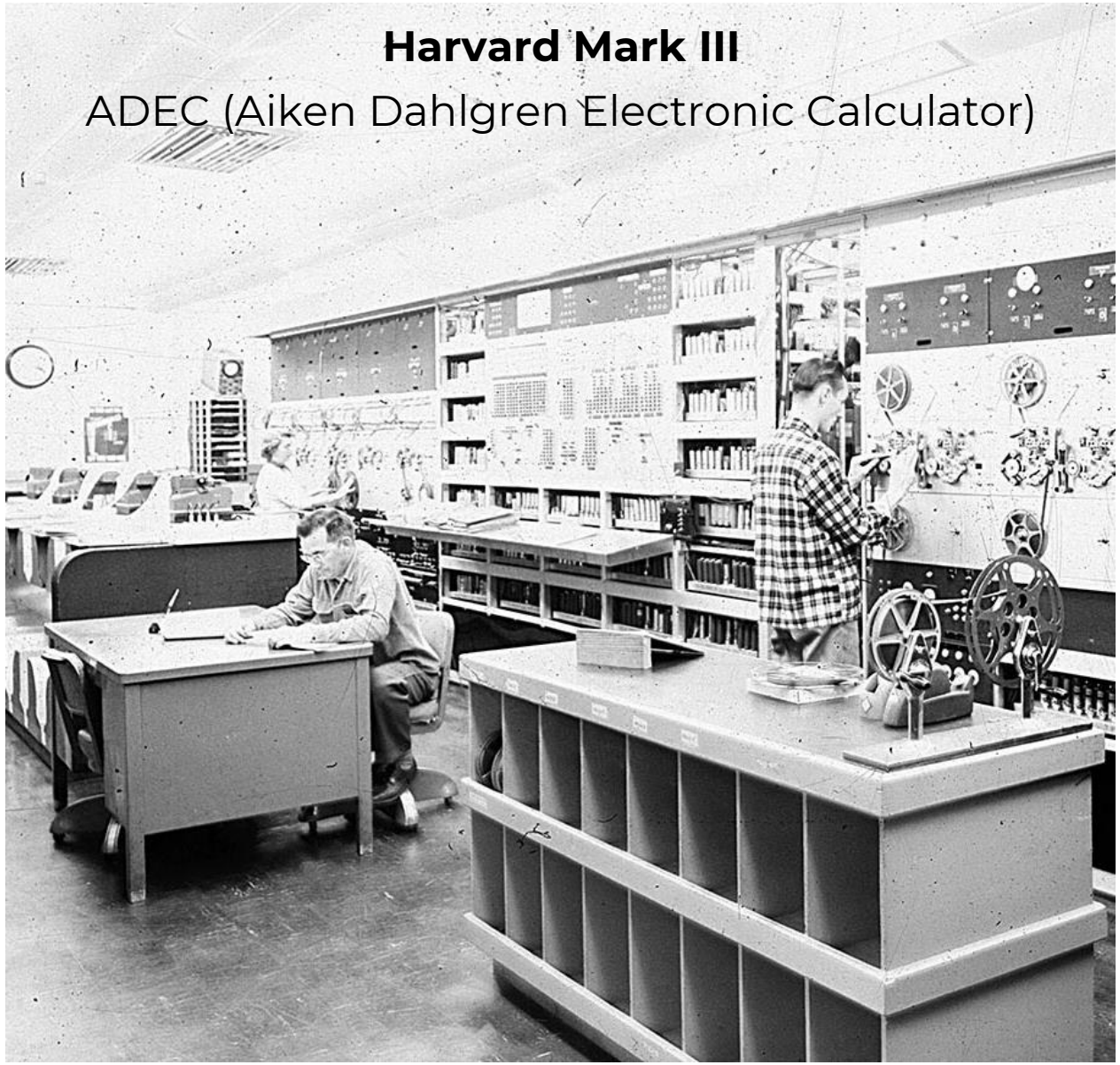
- **Гарвардская архитектура** — архитектура ЭВМ, отличительными признаками которой являются:
  - хранилище инструкций и хранилище данных представляют собой разные физические устройства;
  - канал инструкций и канал данных также физически разделены.
- Архитектура была **разработана Говардом Эйкеном в конце 1930-х годов** в Гарвардском университете.
- **Достоинство Гарвардской архитектуры** – **параллелизм передачи, чтения и обработки команд и данных**, что повышает быстродействие, но усложняет реализацию по сравнению с фон Неймановской.
- Характеристики инструкций и данных (длина слова, временные диаграммы, структуры адресов) различны.
- **Основные отличительные признаки:**
  - **Память для инструкций и данных** – разные физические устройства;
  - **Каналы передачи инструкций и данных** – разные физические устройства;
  - **Инструкции и данные** могут представляться в разных форматах.

История возникновения

- **1944 г.: Создан компьютер Harvard Mark I** (Harvard-IBM Automatic Sequence Controlled Calculator) под руководством Говарда Эйкена.
  - **Важное уточнение**: **Этот компьютер НЕ использовал разделение памяти в современном понимании!** Программы задавались перфолентой, а данные — в отдельных регистрах.
  - **Название архитектуры** позже **закрепилось за принципом разделения памяти**, который фактически не был реализован в Mark I.
- **1947 г.**: Проект **Harvard Mark III** уже включал разделённую память для команд и данных — это **считается первым применением "настоящей" Гарвардской архитектуры**.
- **1970-е гг.**: Принцип стал массово использоваться в **микроконтроллерах** (например, в ранних чипах Intel MCS-51).
- **Термин «Гарвардская архитектура» появился позже как обобщение принципов**, а не как описание конкретной машины Mark I.
- Гарвардская архитектура используется в ПЛК и микроконтроллерах, таких, как Microchip PIC, Atmel AVR, Intel 4004, Intel 8051, а также в кэш-памяти первого уровня x86-микропроцессоров, делящейся на два равных либо различных по объёму блока для данных и команд.

Harvard Mark III

ADEC (Aiken Dahlgren Electronic Calculator)



Гарвардская архитектура

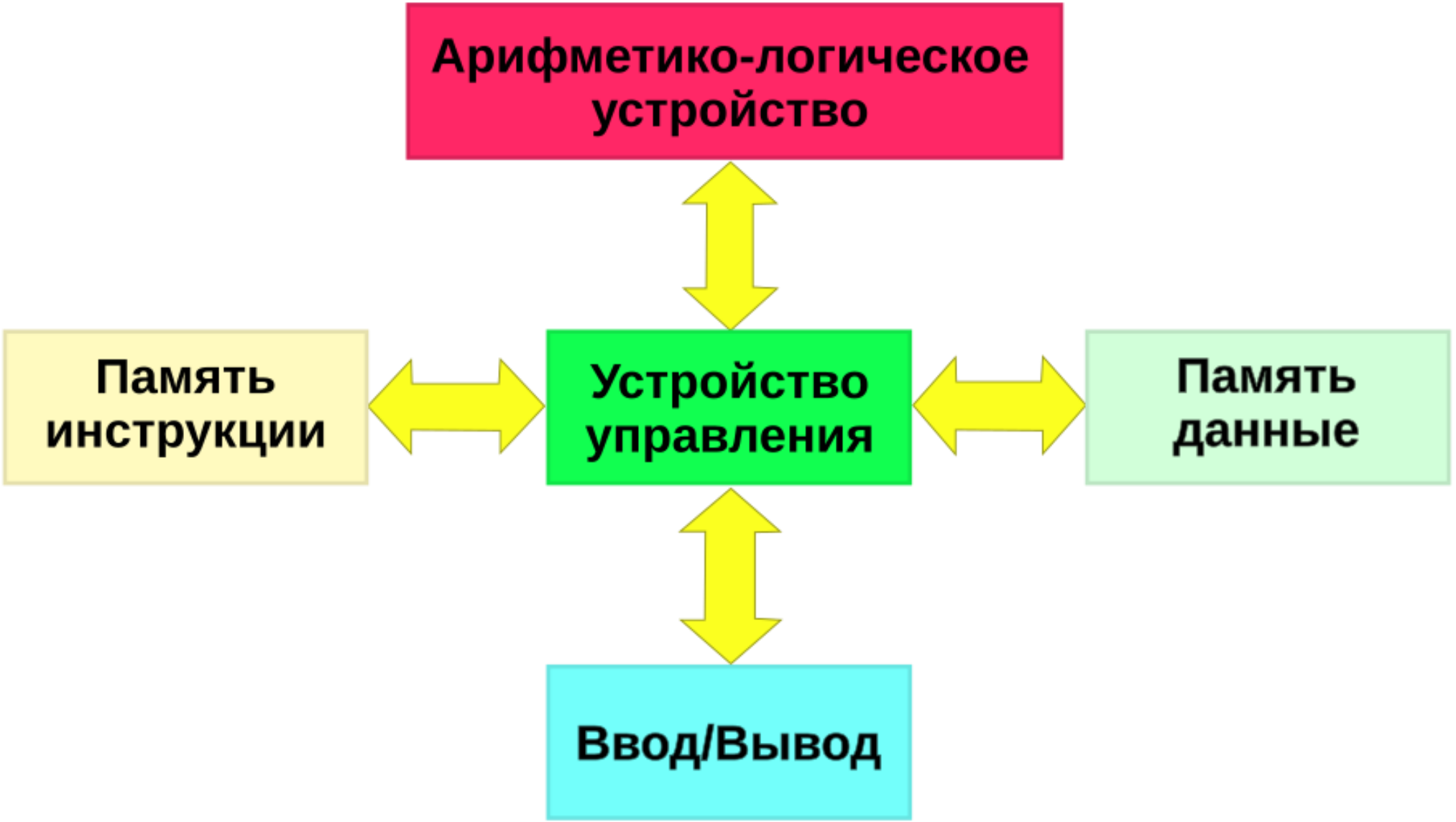
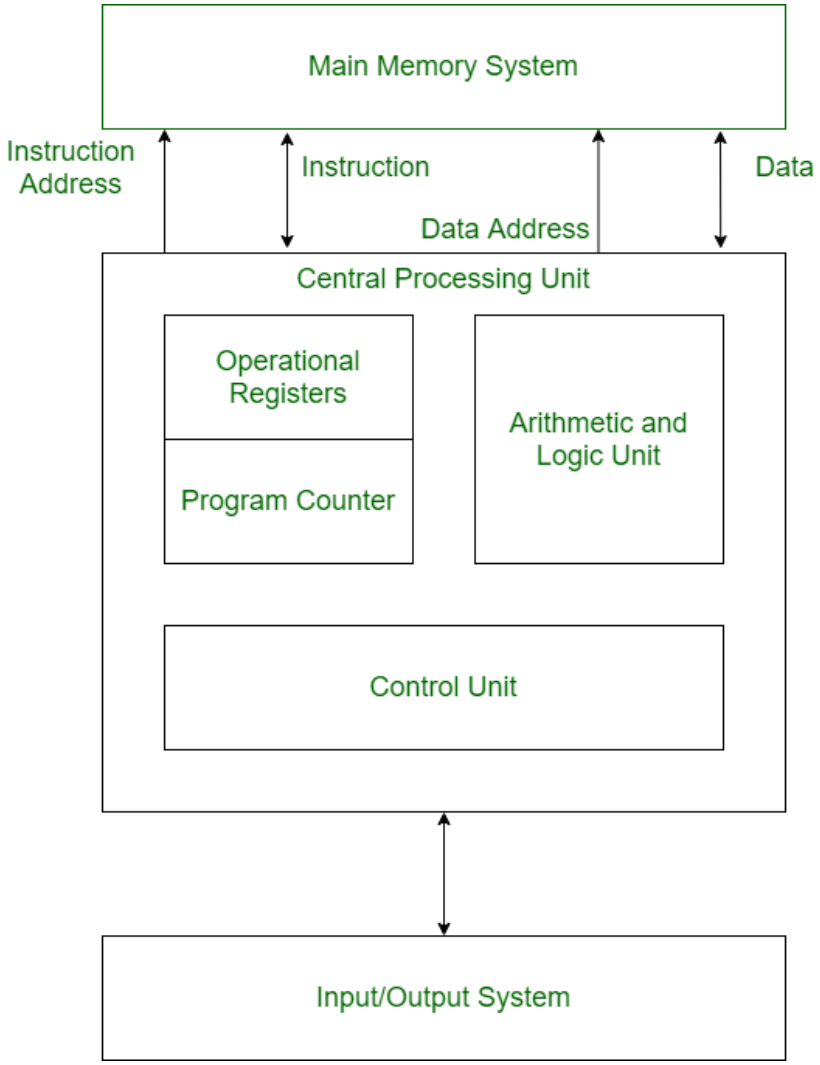


Схема Гарвардской архитектуры компьютера

Гарвардская архитектура



Гарвардская архитектура — модель ЭВМ, в которой программа и данные хранятся в физически разделённых областях памяти с независимыми шинами для их доступа.

Это позволяет:

- Одновременно читать команду и обрабатывать данные,
- Полностью изолировать программный код от данных.

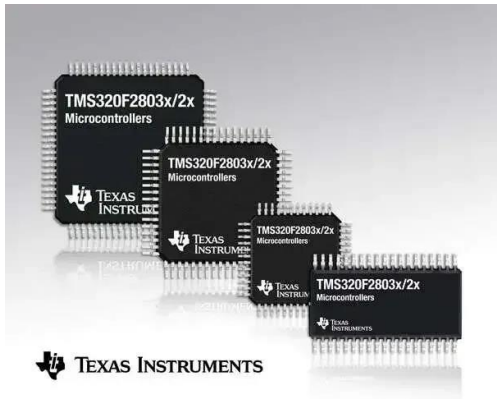
Примеры применения Гарвардской архитектуры

Микроконтроллеры

AVR (Atmel/Microchip)

PIC (Microchip) PIC16F877A

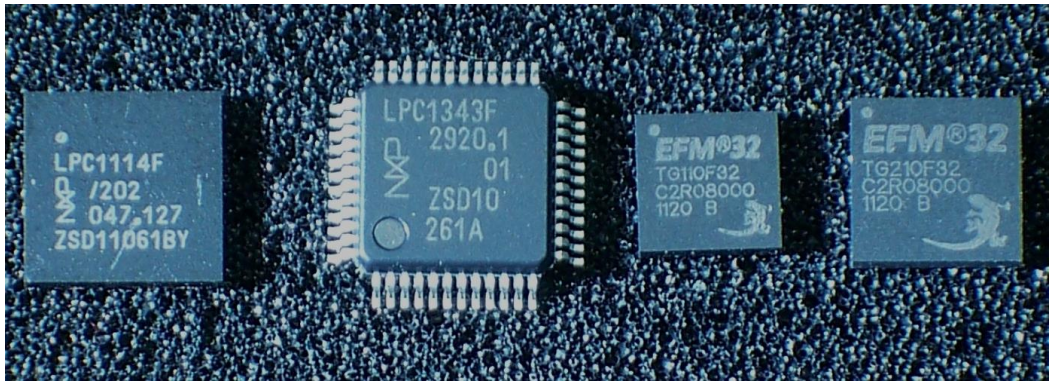
DSP-процессоры



Texas Instruments TMS320

Analog Devices Blackfin ADSP-BF533

Современные процессоры



ARM Cortex-M (Cortex-M0/M3)

RISC-V с расширением Zfinx

Чистая Гарвардская архитектура

- Чистая Гарвардская архитектура (Pure Harvard Architecture) представляет собой теоретическую модель, где **память программ и память данных являются абсолютно разными физическими устройствами**, например, ПЗУ и ОЗУ соответственно.
- Между этими устройствами полностью отсутствуют какие-либо пути для записи в память программ, что обеспечивает максимальную защиту кода.
- Системная шина полностью разделена: существуют отдельная шина команд и отдельная шина данных.
- Это позволяет процессору одновременно осуществлять доступ к инструкции и к данным, что обеспечивает максимальную производительность.
- Однако этот подход имеет существенный недостаток: большое количество выводов на кристалле процессора, что увеличивает его размер, сложность и стоимость производства.
- Из-за этих факторов чистая Гарвардская архитектура применяется редко и в основном в очень специфических и критически важных системах, где производительность и надежность превалируют над стоимостью.

Модифицированная гарвардская архитектура

- Соответствующая схема реализации доступа к памяти имеет один очевидный **недостаток — высокую стоимость**.
- При разделении каналов передачи команд и данных на кристалле процессора последний должен иметь почти вдвое больше интерфейсных выводов, так как шина адреса и шина данных составляют основную часть выводов микропроцессора.
- Способом разрешения этой проблемы стала идея **использовать общие шину данных и шину адреса для всех внешних данных, а внутри процессора использовать шину данных, шину команд и две шины адреса**.
- Такую концепцию стали называть **модифицированной гарвардской архитектурой**.
- Такая схемотехника **применяется в современных сигнальных процессорах**. Ещё дальше по пути уменьшения стоимости пошли при создании однокристальных микроЭВМ — **микроконтроллеров**. В них одна шина команд и данных применяется и внутри кристалла.
- Разделение шин в модифицированной гарвардской структуре осуществляется при помощи раздельных управляющих сигналов: чтения, записи или выбора области памяти.

Гибридные модификации с архитектурой фон Неймана

- **Существуют гибридные архитектуры, сочетающие достоинства как гарвардской, так и фон-неймановской архитектур**.
- **Современные CISC-процессоры** обладают раздельной кэш-памятью 1-го уровня для команд и данных, что позволяет им за один рабочий такт получать одновременно и команду, и данные для её выполнения.
- **То есть процессорное ядро аппаратно гарвардское, но программно оно фон-неймановское, что упрощает написание программ**.
- Обычно в данных процессорах одна шина используется и для передачи команд, и для передачи данных, что схемотехнически упрощает систему.
- Современные варианты таких процессоров могут иногда содержать встроенные контроллеры сразу нескольких разнотипных шин для работы с различными типами памяти — например, DDR RAM и Flash.
- Тем не менее, и в этом случае шины, как правило, используются и для передачи команд, и для передачи данных без разделения, что делает данные процессоры ещё более близкими к фон-неймановской архитектуре при сохранении достоинств гарвардской архитектуры.

Преимущества архитектуры

- **Преимущества** Гарвардской архитектуры обусловлены её ключевым принципом – **физическим разделением памяти и шин для команд (инструкций) и данных**. Это разделение открывает путь к ряду важных технических и эксплуатационных выгод, особенно в специализированных системах.
- **1. Повышенная производительность (параллелизм доступа к памяти)**
- **Одновременный доступ к командам и данным:** Процессор может в один и тот же такт выбирать следующую инструкцию из памяти программ и одновременно читать/писать данные из/в память данных. Это устраняет «узкое горлышко фон Неймана», где доступ к командам и данным происходит поочерёдно через одну шину.
- **Сокращение простоев процессора:** Отсутствие ожидания завершения операции с памятью для выполнения следующего шага увеличивает эффективность конвейера (pipeline).
- **Особенно важно для DSP и MCU:** В цифровых сигнальных процессорах (DSP) и микроконтроллерах (MCU), где критична скорость обработки потоков данных (например, аудио, видео, управление в реальном времени), это преимущество становится решающим.
- **Пример:** В DSP с расширенной Гарвардской архитектурой (SHARC) за один такт можно загрузить команду, два операнда и записать результат – идеально для алгоритмов типа БПФ или фильтрации.

Преимущества архитектуры

- **2. Повышенная надёжность и безопасность**
- **Защита кода от модификации:** Память программ часто реализуется как ПЗУ (ROM, Flash), физически отделённое от ОЗУ данных. Это исключает случайное или злонамеренное изменение исполняемого кода во время работы.
- **Устойчивость к сбоям:** В системах реального времени (автомобили, авионика, медицинские устройства) такая защита критически важна – сбой в данных не может повредить саму программу.
- **Снижение риска эксплойтов:** Невозможность выполнения кода из области данных (DEP – Data Execution Prevention) реализуется на аппаратном уровне, что затрудняет атаки типа buffer overflow.
- **Контраст с фон Нейманом:** В фон-неймановской архитектуре код и данные в одной памяти – переполнение буфера может привести к перезаписи кода и выполнению вредоносных инструкций.

Преимущества архитектуры

- **3. Оптимизация под специализированные задачи**
- **Раздельная оптимизация памяти:** Память программ может быть медленной, но энергоэффективной (Flash), а память данных – быстрой (SRAM). Это позволяет гибко подбирать типы памяти под задачи.
- **Разные адресные пространства:** Программа и данные могут иметь независимые адресные пространства, что упрощает проектирование и позволяет эффективнее использовать ограниченные ресурсы (например, в 8-битных MCU).
- **Поддержка реального времени (RTOS):** Предсказуемое время доступа к памяти команд и данных упрощает планирование задач в системах с жёсткими временными ограничениями.

Преимущества архитектуры

- **4. Энергоэффективность (в некоторых реализациях)**
- **Целенаправленное управление питанием:** Раздельные шины и контроллеры памяти позволяют отключать неиспользуемые блоки (например, шину данных, если выполняется только выборка команд).
- **Меньше конфликтов шины:** Отсутствие арбитража за общую шину снижает энергопотребление на уровне сигнализации.
- **Актуально для IoT и мобильных устройств:** В устройствах с батарейным питанием экономия энергии критична – Гарвардская архитектура помогает продлить время автономной работы.

Преимущества архитектуры

- **5. Упрощение аппаратной реализации в микроконтроллерах**
- **Модифицированная Гарвардская архитектура** (Modified Harvard) – наиболее распространённый вариант – сочетает внутреннее разделение шин (для производительности) с внешней общей шиной (для экономии выводов и стоимости).
- **Меньше конфликтов при доступе к памяти:** Внутри чипа нет необходимости в сложных схемах арбитража шины, что упрощает дизайн и повышает частоту работы.
- **Идеальна для встраиваемых систем:** Низкая стоимость, малый размер кристалла, высокая предсказуемость поведения – всё это делает её выбором №1 для MCU (PIC, AVR, 8051, ARM Cortex-M).

Преимущества архитектуры

- **6. Совместимость с современными гибридными архитектурами**
- **Гарвардские принципы используются даже в фон-неймановских процессорах:** Современные CPU (x86, ARM Cortex-A) используют раздельные кэши L1 для инструкций и данных – это фактически Гарвардская архитектура на уровне кэша.
- **Лучшее из двух миров:** Программист видит единое адресное пространство (фон Нейман), но аппаратно система работает как Гарвардская – обеспечивая и гибкость, и производительность.
- **Эволюционное преимущество:** Гарвардская архитектура не вытеснена – она интегрирована и адаптирована, став неотъемлемой частью современных процессоров.

Когда Гарвардская архитектура особенно выгодна?

- **В встраиваемых системах** (микроконтроллеры, датчики, бытовая техника).
- **В цифровой обработке сигналов** (DSP – аудио, видео, связь).
- **В системах реального времени** (автомобили, промышленная автоматика, авионика).
- **В устройствах с ограниченным питанием** (IoT, носимая электроника).
- **В критически важных системах**, где отказ недопустим (медицинские приборы, военная техника).

Недостатки архитектуры

- Несмотря на значительные преимущества, Гарвардская архитектура имеет и существенные недостатки, которые ограничивают её применение в универсальных вычислительных системах и создают определённые сложности при разработке программного обеспечения и аппаратного дизайна.
- Эти недостатки особенно заметны при сравнении с фон-неймановской архитектурой.

Недостатки архитектуры

- **1. Сложность и высокая стоимость аппаратной реализации**
- **Удвоение шин и контроллеров:** Требуются отдельные шины команд и данных, отдельные контроллеры памяти, буферы и дешифраторы адресов → увеличение площади кристалла и числа выводов (pins).
- **Дороже в производстве:** Особенно в ранние годы электроники (1940–1970-е), когда каждый транзистор и вывод имели высокую стоимость. Это стало одной из главных причин доминирования фон-неймановской архитектуры в массовых компьютерах.
- **Проблема с внешними устройствами:** В чистой Гарвардской архитектуре внешние устройства (например, загрузчик) должны иметь отдельные интерфейсы для записи в память программ и данных → усложнение системной платы.
- **Решение:** Модифицированная Гарвардская архитектура использует общую внешнюю шину – это снижает стоимость, но частично жертвует преимуществами чистой реализации.

Недостатки архитектуры

- **2. Неэффективное использование памяти**
- **Жёсткое разделение объёмов памяти:** Память программ и память данных физически разделены – нельзя динамически перераспределить свободное место из одной области в другую.
  - **Пример:** если в программе мало кода, но много данных, свободное место в памяти программ остаётся неиспользованным.
- **Фрагментация ресурсов:** Разработчик вынужден заранее оценивать объёмы кода и данных, что может привести к неоптимальному использованию чипа.
- **Ограничение гибкости:** В фон-неймановской архитектуре можно динамически выделять память под стек, кучу, код (например, JIT-компиляция), что невозможно в чистой Гарвардской системе.
- **Современные MCU частично решают это:** например, в некоторых ARM Cortex-M есть возможность выполнять код из RAM (XIP – Execute In Place), но это требует дополнительных механизмов и снижает безопасность.

Недостатки архитектуры

- **3. Сложность программирования и отладки**
- **Раздельные адресные пространства:** Программист должен явно указывать, к какой памяти он обращается – к памяти программ (например, для чтения констант) или к памяти данных.
  - В языках низкого уровня (ассемблер, C с расширениями) это требует использования специальных директив или функций (например, PROGMEM в AVR-GCC).
- **Ограниченная поддержка динамических структур:** Невозможно легко реализовать самомодифицирующийся код, JIT-компиляторы, динамическую загрузку модулей – всё это естественно для фон-неймановской архитектуры.
- **Сложности с отладкой:** Отладчики должны уметь работать с двумя разными пространствами памяти, что усложняет реализацию симуляторов и эмуляторов.
- **Пример:** В микроконтроллерах AVR для чтения строки из Flash-памяти нужно использовать специальную функцию pgm_read_byte(), а не обычный указатель – это создаёт дополнительную нагрузку на разработчика.

Недостатки архитектуры

- **4. Ограниченная гибкость архитектуры**
- **Невозможность динамической генерации кода:** Такие технологии, как JIT (Just-In-Time компиляция), самоизменяющийся код, динамические плагины – невозможны или сильно затруднены.
- **Проблемы с современными ОС и виртуализацией:** Большинство операционных систем предполагают единое адресное пространство и возможность выполнения кода из любой области памяти – это несовместимо с чистой Гарвардской моделью.
- **Трудности с поддержкой сложных языков:** Языки с интенсивным использованием метапрограммирования, reflection, динамической компиляции (например, Python, JavaScript, Java в некоторых режимах) работают неэффективно или требуют эмуляции.

Недостатки архитектуры

- **5. Компромиссы в модифицированных реализациях**
- **Потеря преимуществ при использовании общей внешней шины:** В модифицированной Гарвардской архитектуре внутренние шины разделены, но внешняя – общая. При частом обращении к внешней памяти возникает «узкое горлышко», как в фон-неймановской системе.
- **Зависимость от кэшей:** В гибридных системах (например, ARM/x86 с раздельными L1-кэшами) производительность сильно зависит от эффективности кэширования. При промахах кэша система деградирует до фон-неймановского поведения.
- **Сложность синхронизации:** Если код может обновляться (например, через загрузчик), требуется механизм синхронизации между памятью программ и кэшем инструкций – иначе процессор может выполнять устаревший код.

Недостатки архитектуры

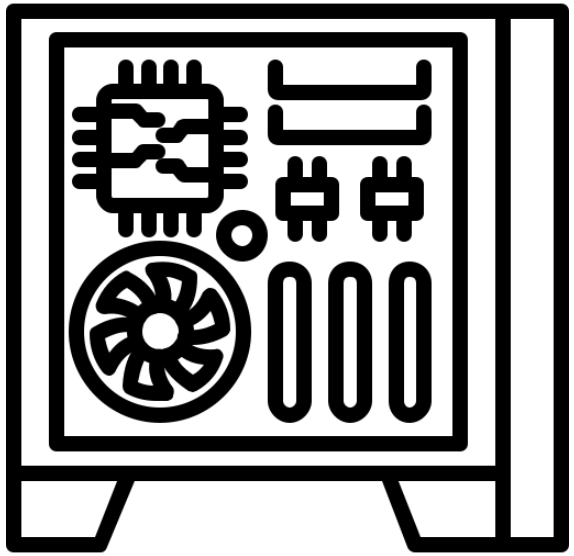
- **6. Ограниченная применимость в универсальных системах**
- **Не подходит для ПК, серверов, рабочих станций:** Современные ОС, языки программирования, среды выполнения (runtime) и приложения требуют гибкости фон-неймановской модели.
- **Невозможность реализации некоторых алгоритмов:** Например, генетические алгоритмы, эволюционное программирование, где код модифицируется в процессе выполнения – несовместимы с жёсткой изоляцией кода.
- **Меньше поддержки со стороны инструментов:** Компиляторы, линковщики, отладчики для фон-неймановских систем более развиты и стандартизированы.

Выводы

- **Гарвардская архитектура** представляет собой фундаментальный подход к организации вычислительных систем, **основанный на физическом разделении памяти и шин для команд и данных**. Её **ключевые преимущества** – **повышенная производительность** за счёт параллельного доступа к инструкциям и операндам, а также **повышенная надёжность** и безопасность благодаря аппаратной изоляции кода от данных. Эти особенности делают её идеальной для систем, где критичны предсказуемость, скорость реакции и устойчивость к сбоям. Однако за эти преимущества приходится платить: **Гарвардская архитектура сложнее в реализации, дороже в производстве**, менее гибка в программировании и неэффективно использует память. Особенно остро эти недостатки проявляются в универсальных вычислительных системах, где требуется динамическое управление памятью, поддержка сложных языков и операционных систем.
- Современное развитие показывает, что чистая Гарвардская архитектура практически не используется – вместо неё доминируют **модифицированные и гибридные реализации**. **Микроконтроллеры применяют модифицированную версию** (внутреннее разделение шин, внешняя – общая), что снижает стоимость и сохраняет производительность. Универсальные процессоры (x86, ARM Cortex-A) используют гибридный подход: на уровне программиста – фон-неймановская модель, на аппаратном уровне – раздельные кэши L1 для команд и данных. Это позволяет сочетать гибкость универсальных систем с производительностью, характерной для Гарвардской архитектуры. Таким образом, принципы Гарвардской архитектуры не исчезли – они эволюционировали и стали неотъемлемой частью современного процессорного дизайна.

Область применения

- **Гарвардская архитектура** (в чистом, модифицированном или гибридном виде) находит широкое применение там, где важны **производительность, энергоэффективность, надёжность и предсказуемость**:
  - **Микроконтроллеры (MCU)** – основа встраиваемых систем: PIC, AVR, 8051, ARM Cortex-M. Используются в бытовой технике, автомобильной электронике, промышленных контроллерах, IoT-устройствах.
  - **Цифровые сигнальные процессоры (DSP)** – обработка аудио, видео, радиосигналов в реальном времени: семейства Texas Instruments C6000, Analog Devices SHARC, Freescale MSC.
  - **Системы реального времени (RTOS)** – авионика, медицинское оборудование, АСУ ТП, робототехника — где сбой недопустим, а реакция должна быть мгновенной.
  - **Безопасные и критически важные системы** – военная техника, системы управления АЭС, железнодорожная автоматика – где изоляция кода предотвращает катастрофические последствия.
  - **Устройства с ограниченным питанием** – носимая электроника, сенсоры, беспроводные узлы IoT – где энергоэффективность и автономность важнее гибкости.
- Таким образом, Гарвардская архитектура остаётся одной из самых востребованных в мире цифровой электроники – не как конкурент фон-неймановской модели, а как её мощное **дополнение, оптимизированное под специализированные, надёжные и высокопроизводительные задачи**.



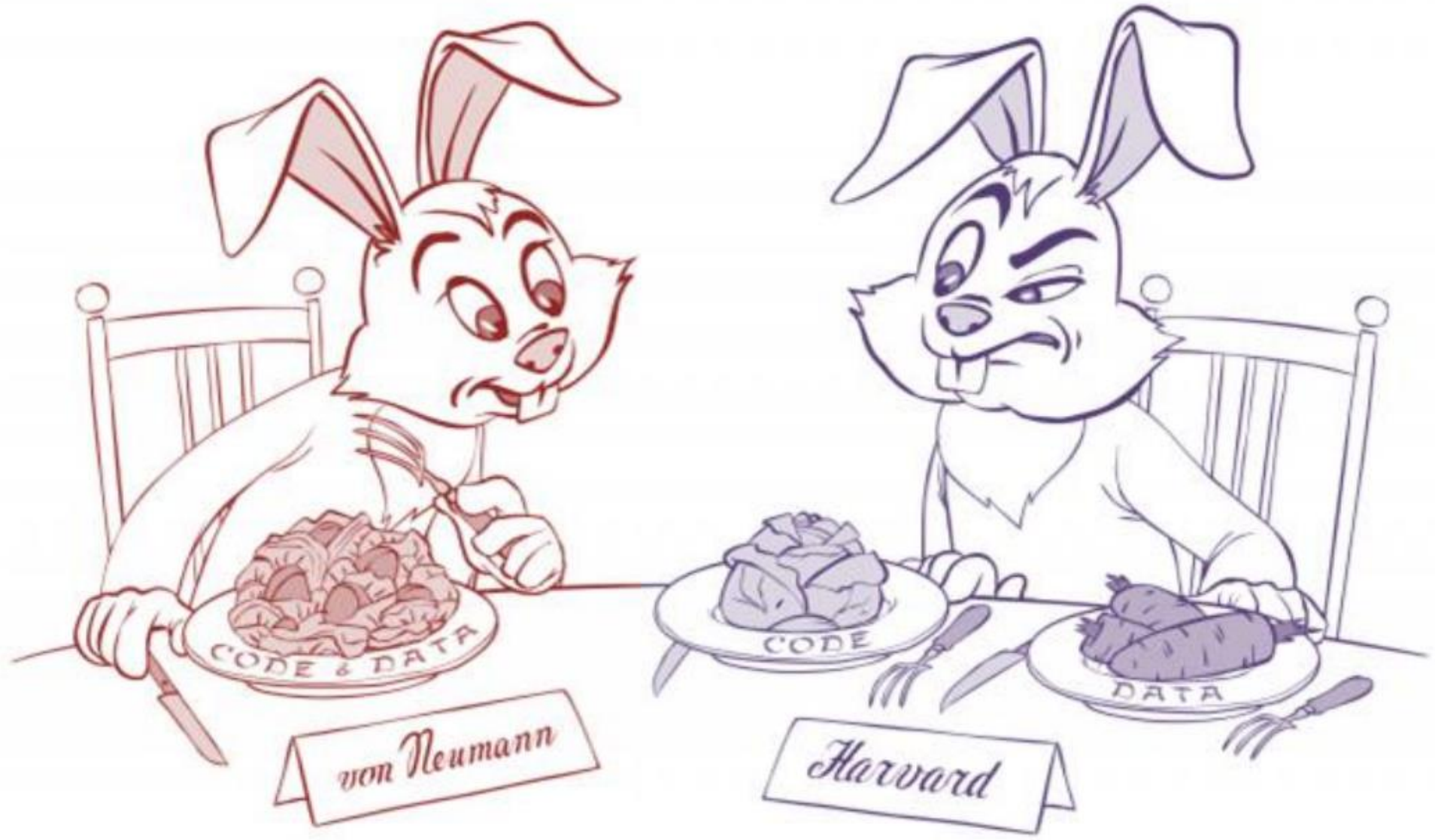
Архитектура фон Неймана vs Гарвардская архитектура

Классические архитектуры ЭВМ

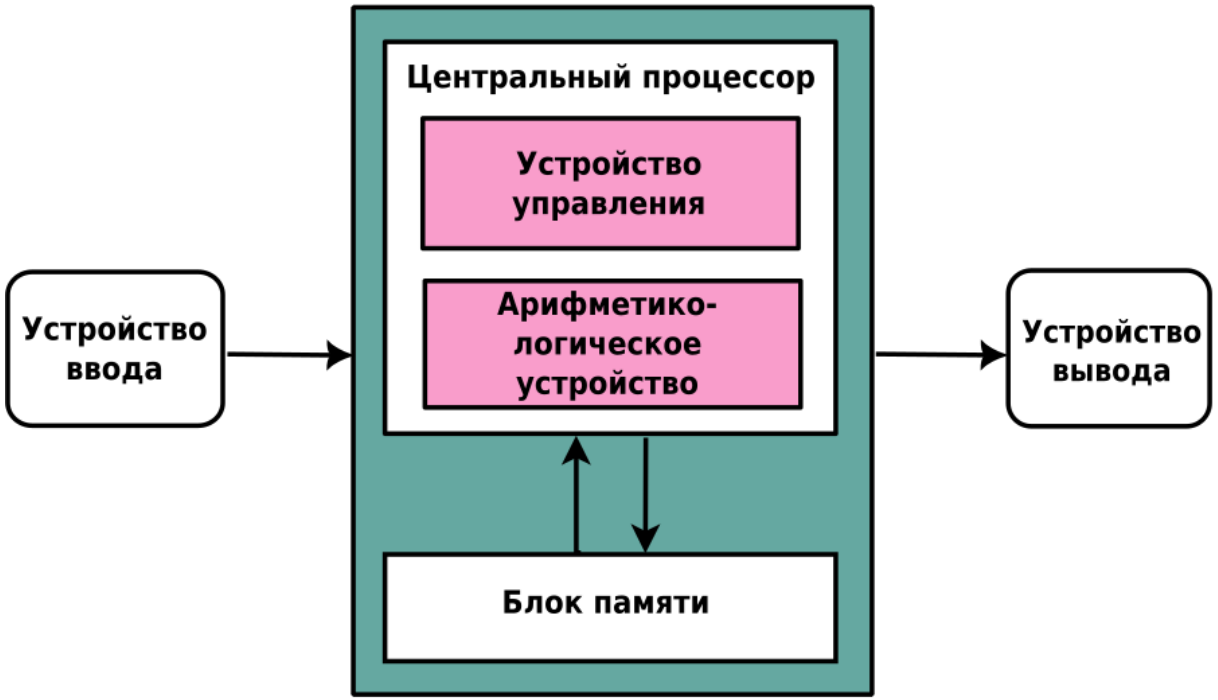
- **Гарвардская архитектура предполагает раздельное использование памяти программ и данных**. Обычно такую архитектуру используют для повышения быстродействия системы за счет разделения путей доступа к памяти программ и данных. Большинство специализированных микропроцессоров (особенно **микроконтроллеры**) имеют данную архитектуру.

- Антипод гарвардской – **архитектура фон Неймана – предполагает хранение программ и данных в общей памяти** и наиболее характерна для микропроцессоров, ориентированных на использование в компьютерах. Примером могут служить **микропроцессоры семейства x86**.

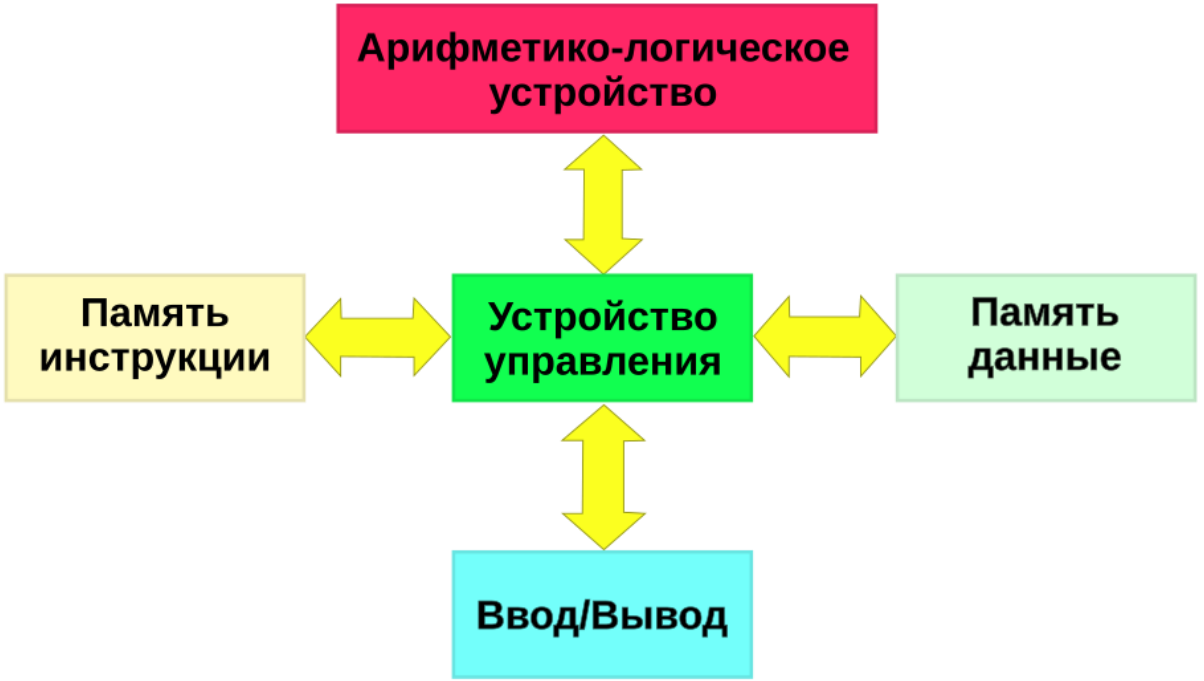
Архитектура фон Неймана vs Гарвардская архитектура



Архитектура фон Неймана vs Гарвардская архитектура

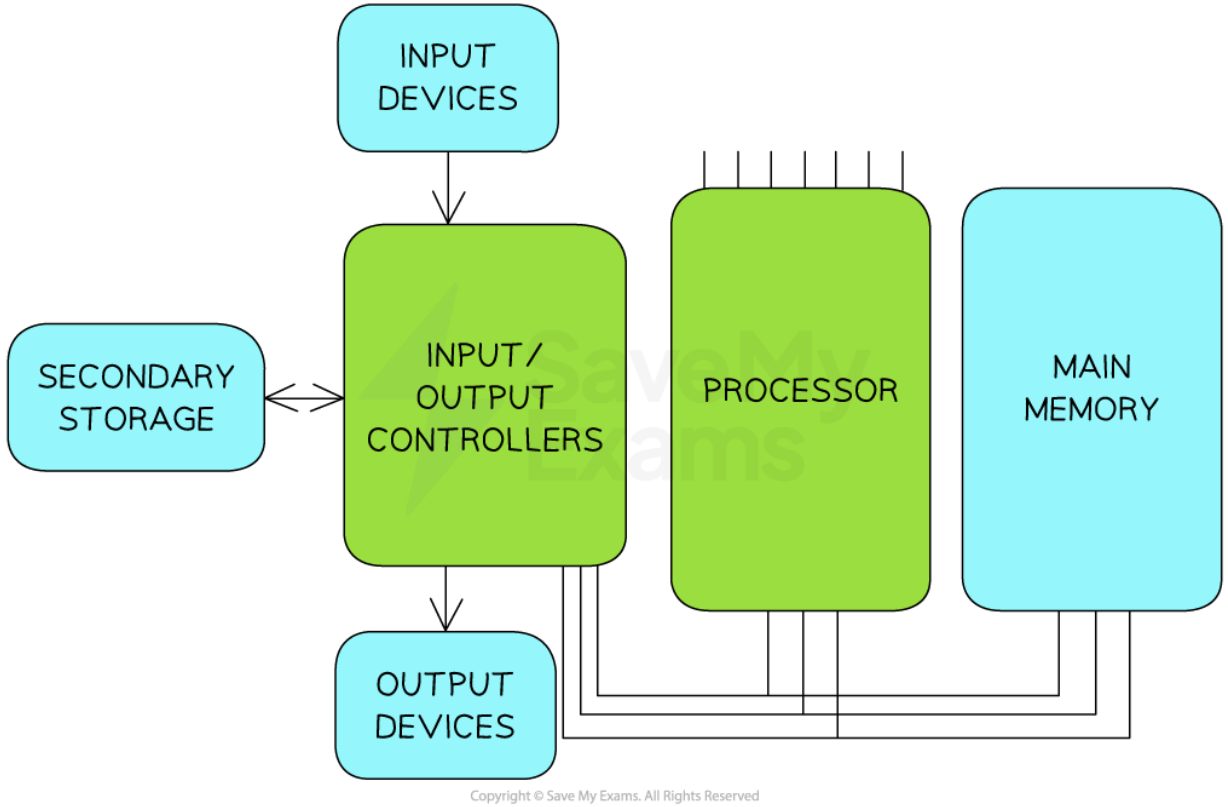


Схематичное изображение архитектуры фон Неймана

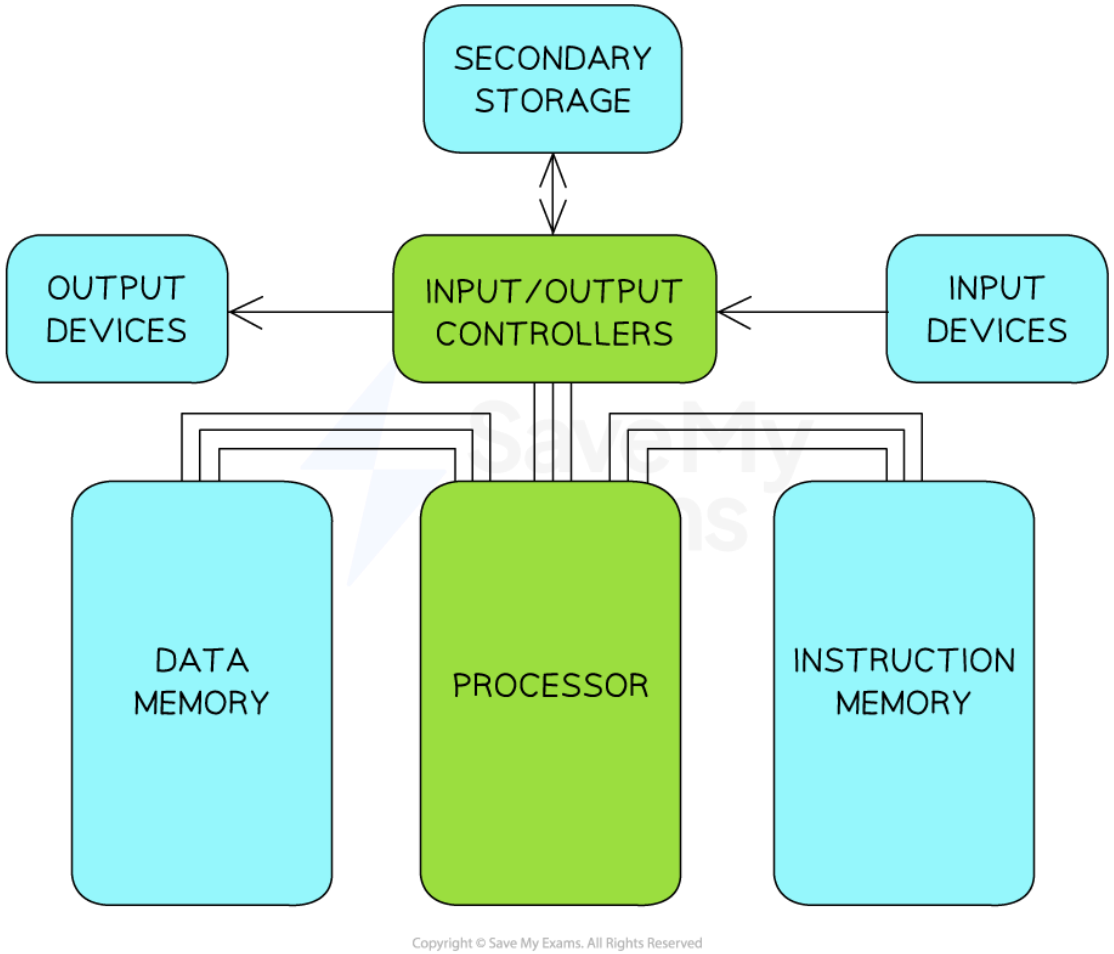


Схематичное изображение Гарвардской архитектуры

Архитектура фон Неймана vs Гарвардская архитектура



Layout of Von Neumann Architecture



Layout of Harvard Architecture

Архитектура фон Неймана vs Гарвардская архитектура

- Гарвардская архитектура эффективна для встраиваемых систем.
- Архитектура фон Неймана — для универсальных компьютеров.

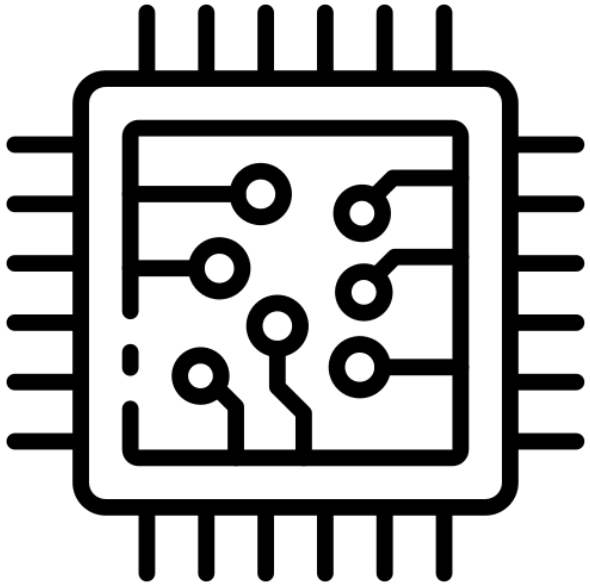
Область применения	Гарвардская архитектура	Архитектура фон Неймана
Микроконтроллеры (MCU)	✅ Доминирует(PIC, AVR, Cortex-M)	❌ Не применяется
Цифровые сигнальные процессоры (DSP)	✅ Доминирует(TMS320, SHARC)	❌ Не применяется
Системы реального времени (RTOS)	✅ Предпочтительна(автомобили, авионика)	⚠ Применяется, но с риском
Персональные компьютеры и серверы	❌ Не применяется	✅ Доминирует(Intel, AMD)
Смартфоны и планшеты	❌ Не применяется	✅ Доминирует(ARM, Apple Silicon)
Суперкомпьютеры	❌ Не применяется	✅ Доминирует(Xeon, EPYC)
Облачные вычисления	❌ Не применяется	✅ Доминирует

Основные различия: Структура и Принципы

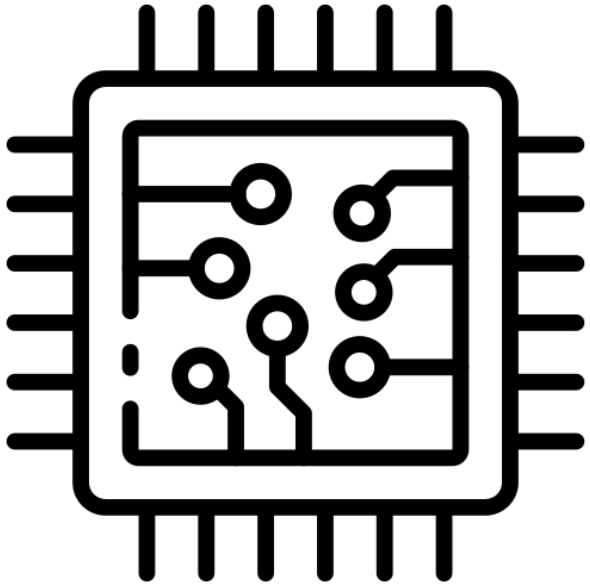
Характеристика	Гарвардская архитектура	Архитектура фон Неймана (Принстонская)
Память	**Физически разделена:** отдельная память для команд (обычно Flash/ROM) и отдельная для данных (обычно RAM).	**Единая память:** команды и данные хранятся в одном адресном пространстве (обычно RAM).
Шины	**Раздельные шины:** отдельная шина для доступа к командам и отдельная для данных.	**Единая системная шина:** одна шина для передачи и команд, и данных.
Производительность	**Выше в специализированных задачах:** процессор может одновременно выбирать команду и читать/писать данные.	**Ниже из-за "узкого места":** процессор не может одновременно обращаться к команде и данным — это создает задержку.
Безопасность и надежность	Выше: **код в ПЗУ защищен** от случайного или злонамеренного изменения.	Ниже: отсутствие разделения кода и данных делает систему **уязвимой к атакам** (например, переполнение буфера).
Гибкость	Ниже: **сложнее реализовать динамическую загрузку кода**, JIT-компиляцию.	Выше: **программа может динамически изменять себя**, загружать модули, что критично для универсальных ОС.
Сложность и стоимость	Выше (в чистом виде): **требуется больше** выводов, отдельные контроллеры памяти.	Ниже: **проще и дешевле** реализовать, особенно на ранних этапах развития электроники.

Итог

- Выбор между Гарвардской и фон Неймановской архитектурой – это **компромисс между специализацией и универсальностью**.
  - **Гарвардская архитектура** – это выбор для "встраиваемого мира", где миллиарды устройств должны работать надежно, энергоэффективно и предсказуемо. Она идеальна для задач с жесткими требованиями к времени реакции и безопасности.
  - **Архитектура фон Неймана** – это выбор для "мира универсальных вычислений", где главным требованием является гибкость, способность запускать любое ПО и поддерживать сложные операционные системы. Она — основа для ПК, серверов и смартфонов.
- Современная реальность показывает, что **чистых реализаций почти не существует**. Даже самые "фон-неймановские" процессоры используют гарвардские принципы на уровне кэшей, а "гарвардские" микроконтроллеры часто имеют механизмы для выполнения кода из RAM.
- Это свидетельствует о том, что лучшие идеи обеих архитектур успешно дополняют друг друга, создавая оптимальные решения для конкретных задач.



Классификация Архитектур ЭВМ



Архитектура ЭВМ. По организации памяти и кода

Классификация По организации памяти и кода

- **Архитектура ЭВМ по организации памяти и кода классифицируется прежде всего по фундаментальному принципу хранения и доступа к инструкциям и данным.**
- На одном полюсе находится **архитектура фон Неймана**, где программа и данные сосуществуют в едином адресном пространстве и передаются по одной общей системной шине.
- Эта модель обеспечивает максимальную универсальность и гибкость — программа может динамически изменять себя, что лежит в основе всей современной программной экосистемы (операционные системы, виртуальные машины, JIT-компиляция).
- Однако ее главный недостаток — «узкое место фон Неймана» — возникает из-за невозможности одновременного доступа к команде и операнду, что ограничивает производительность.
- На противоположном полюсе — **гарвардская архитектура**, физически разделяющая память и шины для команд и данных, что позволяет процессору работать параллельно и значительно повышает скорость и надежность (код защищен от случайной перезаписи).
- Эта модель доминирует в специализированных системах: микроконтроллерах (AVR, PIC) и цифровых сигнальных процессорах (DSP).
- Современные универсальные процессоры (x86, ARM) используют **гибридный подход**: на уровне кэш-памяти первого уровня (L1) они следуют гарвардской модели (раздельные кэши команд и данных), а на уровне системной памяти сохраняют единую фон-неймановскую адресацию, сочетая производительность с гибкостью.

Классификация По организации памяти и кода

- Дальнейшая классификация описывает структуру адресного пространства и способы доступа к памяти в сложных системах.
- **Единое (flat) адресное пространство** предоставляет программисту линейный массив памяти, что упрощает программирование, в то время как сегментированное делит память на логические блоки (код, данные, стек), что было характерно для ранних архитектур, таких как x86 в реальном режиме.
- В многопроцессорных системах ключевым различием является способ доступа к памяти: **UMA (Uniform Memory Access)** обеспечивает всем процессорам равный и быстрый доступ к общей памяти, но плохо масштабируется; **NUMA (Non-Uniform Memory Access)** распределяет память по узлам, обеспечивая высокую масштабируемость ценой неравномерного времени доступа.
- **CC-NUMA** добавляет к NUMA аппаратную когерентность кэшей, упрощая программирование.
- Для крупномасштабных систем используются **распределенная память (кластеры, MPP)**, где каждый узел имеет свою изолированную память, и обмен происходит через сеть (MPI), а также **DSM (Distributed Shared Memory)**, создающая иллюзию единой памяти поверх распределенной.
- Критически важной является **виртуальная память** (страничная, сегментная), которая абстрагирует физическую память, обеспечивая изоляцию процессов и возможность использования дискового пространства.
- Перспективные направления, такие как **Near-/In-Memory Computing (PIM)** и **унифицированная память CPU–GPU (hUMA)**, стремятся преодолеть классическое разделение вычислений и памяти, интегрируя их для повышения производительности и энергоэффективности.

По организации памяти и кода (укрупненно)

- **Фон Неймана: единая память для инструкций и данных.**
- **Гарвардская: раздельные памяти/шины для инструкций и данных.**
- **Модифицированная гарвардская:** общий адресное пространство, но раздельные кэши I/D.
- **Единое адресное пространство** (flat) vs **сегментированное** (segmented).
- **UMA** (Uniform Memory Access): равномерный доступ ко всем ячейкам памяти.
- **NUMA** (Non-Uniform Memory Access): разное время доступа в зависимости от узла.
- **CC-NUMA**: NUMA с когерентностью кэшей.
- **Распределённая память** (cluster/distributed): память локальна узлам, обмен по сети.
- **Виртуальная память**: страничная/сегментно-страничная, TLB.
- и др.

По организации памяти и кода

- **Архитектура фон Неймана (Принстонская):**
  - **Единая память** для хранения и команд (кода программы), и данных.
  - **Единая системная шина** для доступа к памяти.
  - **Преимущества**: Универсальность, гибкость (программа может менять себя), простота программной модели.
  - **Недостатки**: **"Узкое место фон Неймана"** — невозможность одновременного доступа к команде и данным, что ограничивает производительность. Уязвимость безопасности (код и данные не разделены).
  - **Применение**: Основа для подавляющего большинства универсальных компьютеров (ПК, серверы, смартфоны).
- **Гарвардская архитектура:**
  - **Физически раздельная память для команд и данных.**
  - **Раздельные шины** для доступа к каждой из них.
  - **Преимущества**: Высокая производительность (параллельный доступ к коду и данным), повышенная надежность и безопасность (код защищен от случайной перезаписи).
  - **Недостатки**: Сложнее и дороже в реализации, менее гибкая.
  - **Применение**: Микроконтроллеры (AVR, PIC), цифровые сигнальные процессоры (DSP).
- **Модифицированная гарвардская архитектура (Гибридная):**
  - **На уровне кэш-памяти L1** — раздельные кэши для инструкций (L1I) и данных (L1D) — это гарвардский принцип.
  - **На уровне системной памяти (DRAM) и адресного пространства** — единое, как у фон Неймана.
  - **Преимущества**: Комбинирует производительность Гарвардской (параллелизм на уровне кэша) с гибкостью фон Неймана (единое адресное пространство).
  - **Применение**: Стандарт де-факто для всех современных высокопроизводительных процессоров (Intel Core, AMD Ryzen, Apple Silicon, ARM Cortex-A).

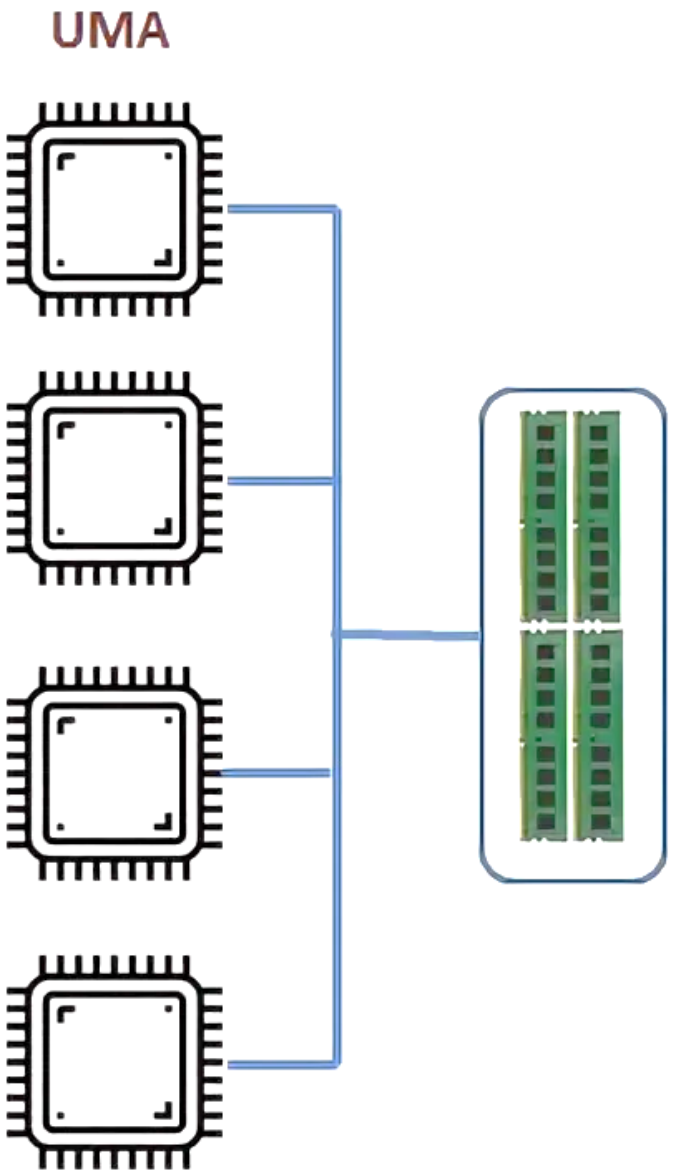
По организации памяти и кода

- **Единое (flat) адресное пространство**
  - Все ячейки памяти (код, данные, стек, устройства) доступны через единый непрерывный диапазон адресов. **Вся память** (ОЗУ, ПЗУ, порты ввода-вывода) **представлена как один непрерывный линейный массив байтов**. Каждый байт имеет уникальный адрес.
  - Упрощает программирование и управление памятью.
  - **Противоположность**: сегментированная модель.
  - **Плюсы**: Простота программирования и управления памятью.
  - **Минусы**: Может быть неэффективно для очень больших объемов памяти или систем с разнородными устройствами.
  - **Применение**: Современные 32/64-битные системы с виртуальной памятью. Используется в современных ОС и процессорах.

По организации памяти и кода

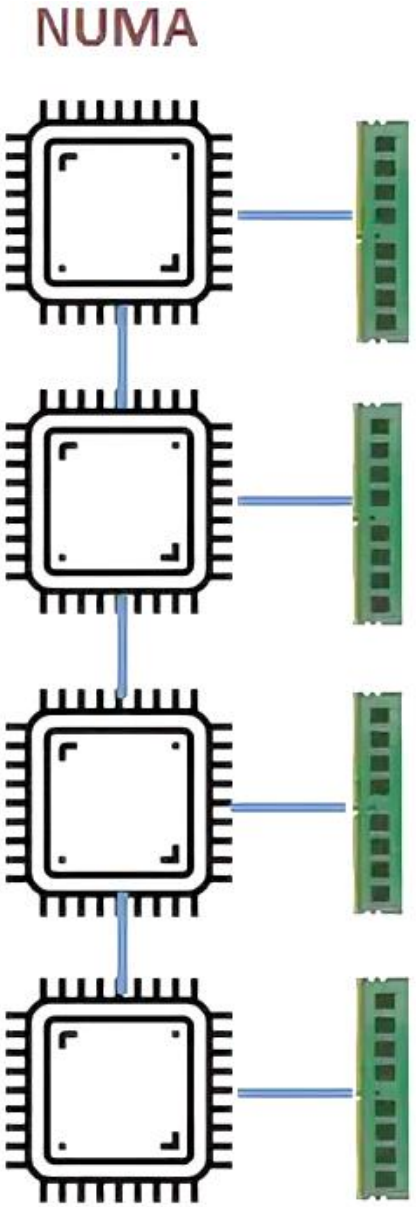
- **Сегментированное адресное пространство**
  - **Память делится на сегменты** (например, сегмент кода, сегмент данных, сегмент стека).
  - Адрес состоит из селектора сегмента и смещения.
  - Позволяет изолировать области памяти, но усложняет адресацию.
  - **Плюсы**: Позволяет эффективно использовать 16-битные регистры для адресации больших объемов памяти (как в x86 в реальном режиме). Упрощает изоляцию кода, данных и стека.
  - **Минусы**: Сложность программирования, фрагментация памяти.
  - **Применение**: Ранние процессоры (Intel 8086), режим совместимости в x86-64. Пример: x86 в реальном режиме и защищённом режиме (до 64-битного режима).

По организации памяти и кода

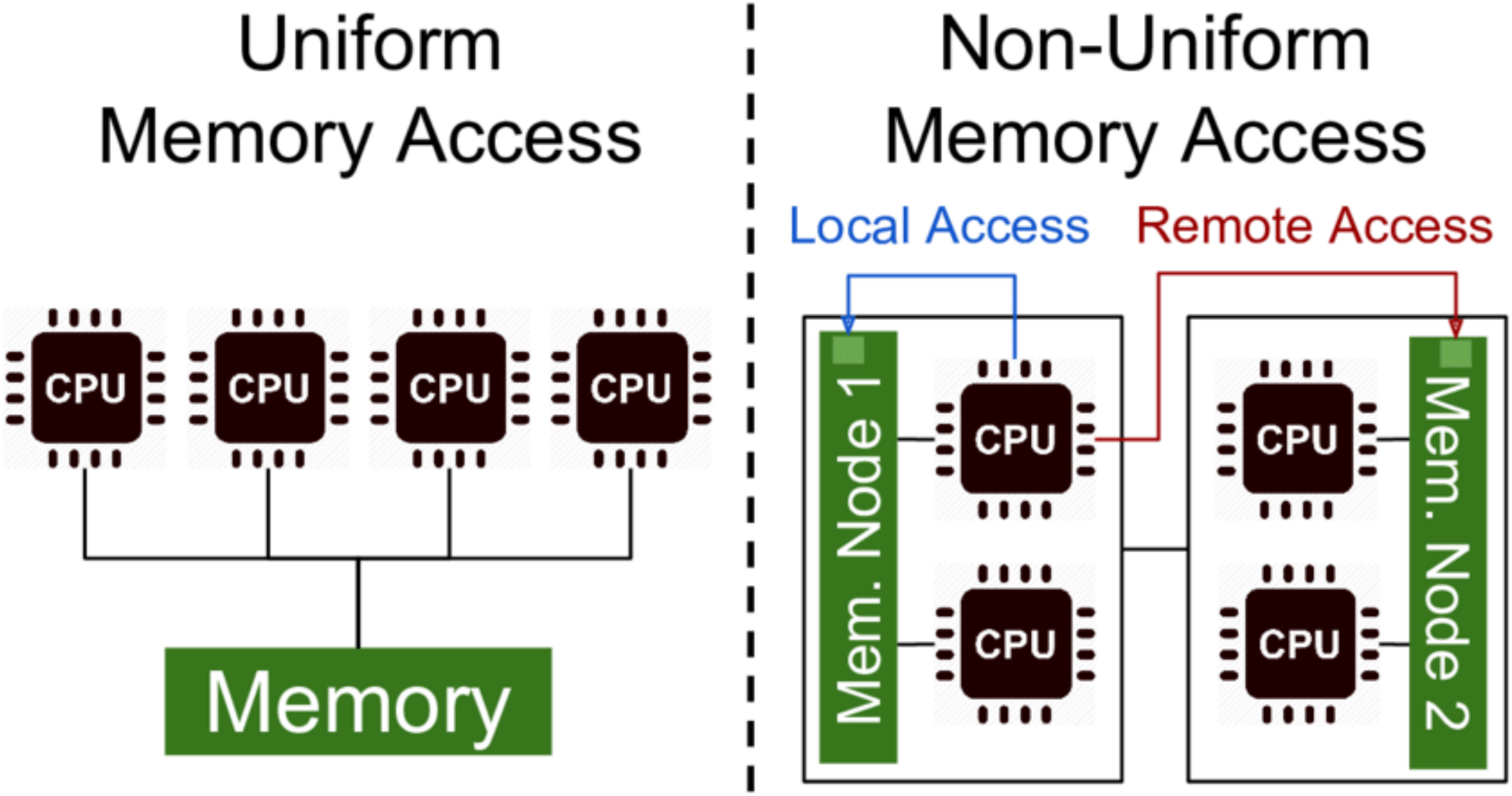


- **UMA (Uniform Memory Access)**
  - **Однородный доступ к памяти**.
  - Все процессоры в многопроцессорной системе имеют одинаковое время доступа к любой ячейке ОЗУ.
  - Общая шина или кроссбар-коммутатор.
  - Простота, но масштабируемость ограничена.
  - Пример: SMP-системы с общей памятью.

- **NUMA (Non-Uniform Memory Access)**
  - **Неоднородный доступ к памяти**.
  - Память логически едина, но физически распределена по узлам.
  - Доступ к "локальной" памяти быстрее, чем к "удалённой".
  - Используется в многопроцессорных серверах.
  - Пример: системы с несколькими CPU-сокетами (Intel Xeon, AMD EPYC).

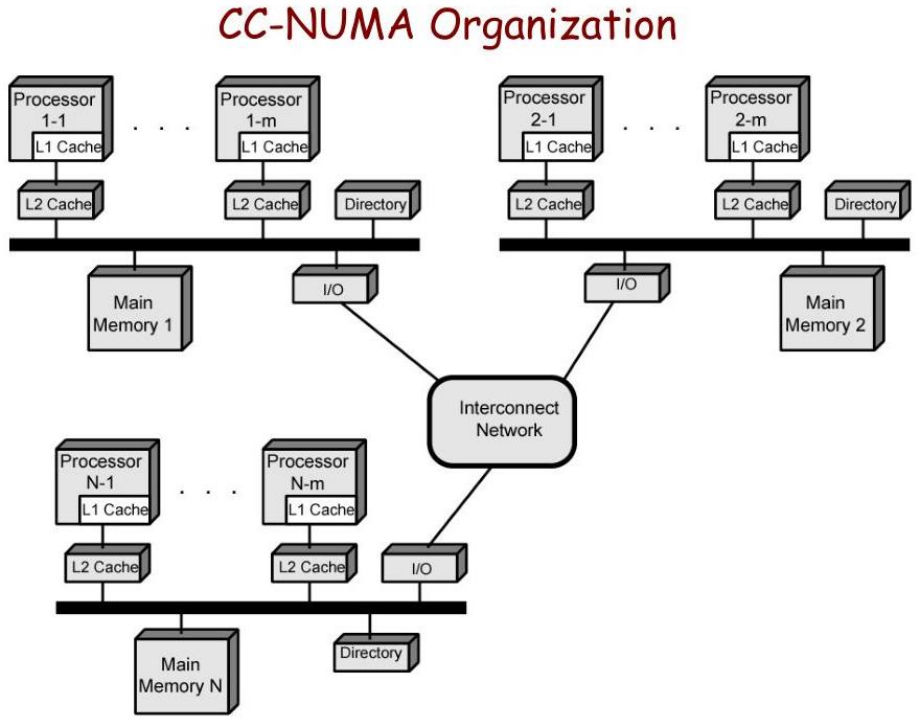


По организации памяти и кода

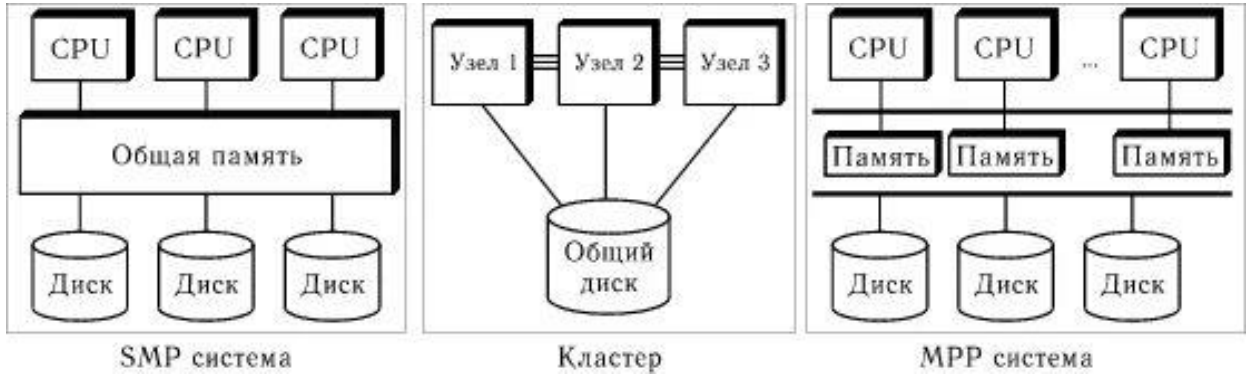


По организации памяти и кода

- **CC-NUMA (Cache-Coherent NUMA)**
  - NUMA с когерентностью кэшей.
  - Обеспечивает согласованность данных в кэшах разных процессоров.
  - Использует протоколы когерентности (MESI, MOESI).
  - Позволяет программам работать с общей памятью, как в UMA, но с NUMA-производительностью.
  - Пример: современные серверные архитектуры (AMD EPYC, Intel Scalable Processors).

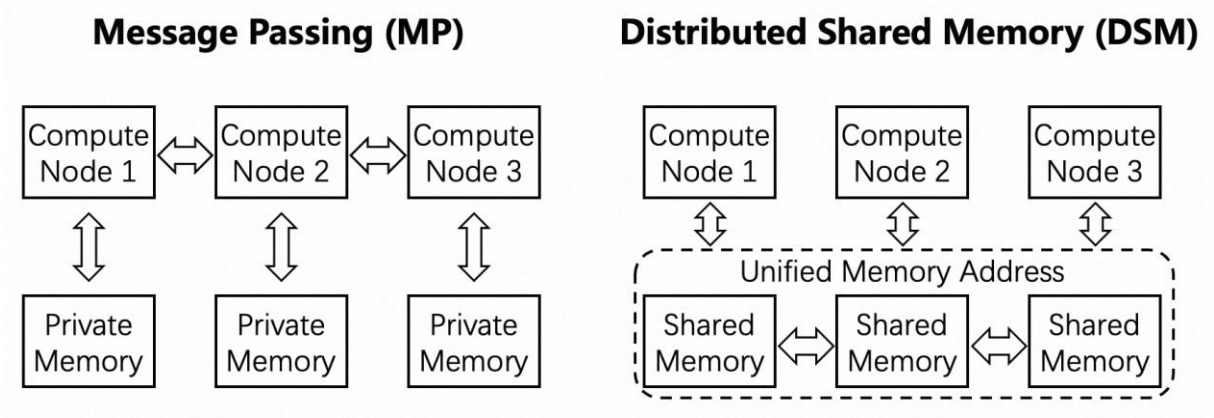


- **Распределённая память (cluster, MPP)**
  - Каждый узел (нода) имеет свою собственную память.
  - Нет единого адресного пространства — обмен данными через сети (MPI, RDMA).
  - MPP (Massively Parallel Processor) — системы с тысячами процессоров (суперкомпьютеры).
  - Высокая масштабируемость, но сложное программирование.
  - Пример: кластеры, суперкомпьютеры (типа "Ломоносов").

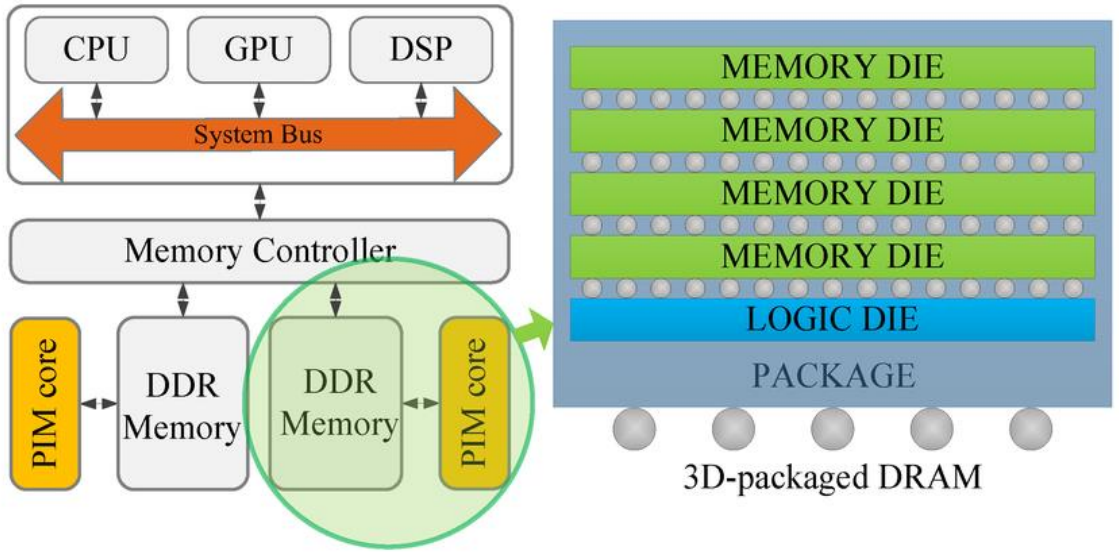


По организации памяти и кода

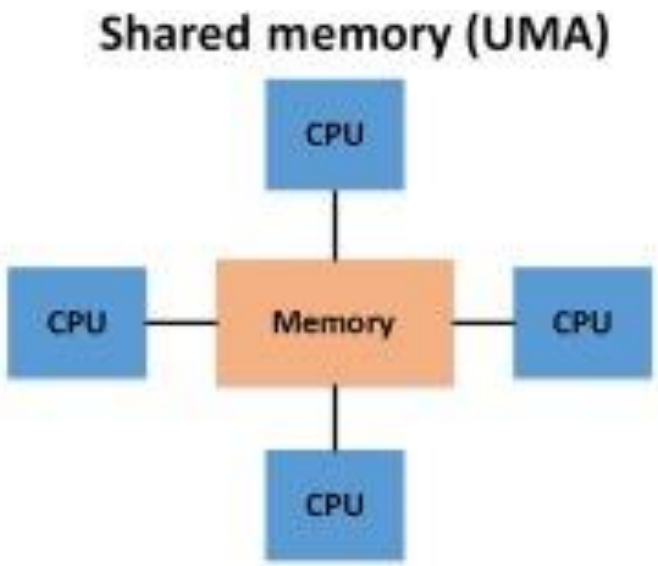
- **DSM (Distributed Shared Memory)**
  - Распределённая разделяемая память.
  - Создаёт иллюзию единого адресного пространства на системе с распределённой памятью.
  - Реализуется:
  - Hard DSM: аппаратно (специальные коммутаторы, когерентные интерконнекты).
  - Soft DSM: программно (через ОС или среду выполнения).
  - Цель — упростить программирование параллельных систем.
  - Пример: InfiniBand + RDMA, некоторые HPC-системы.

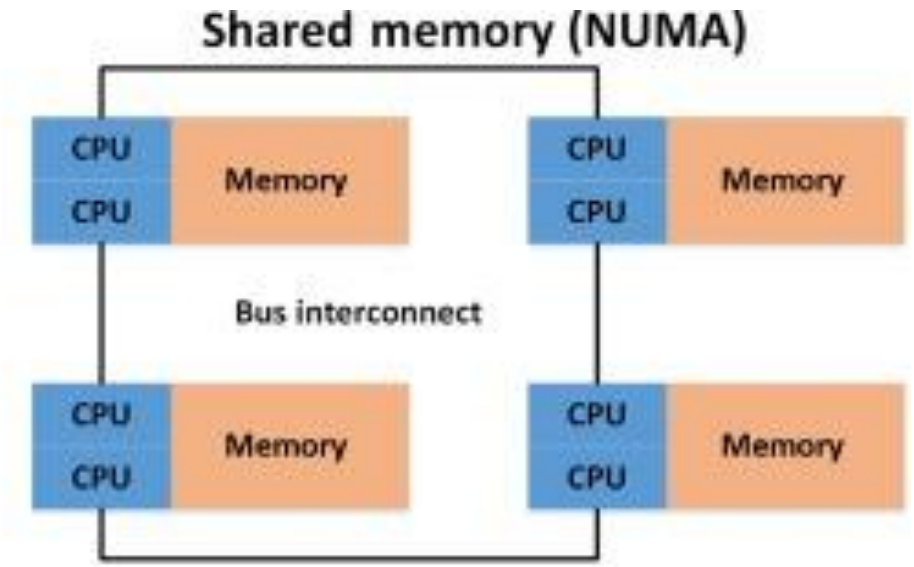


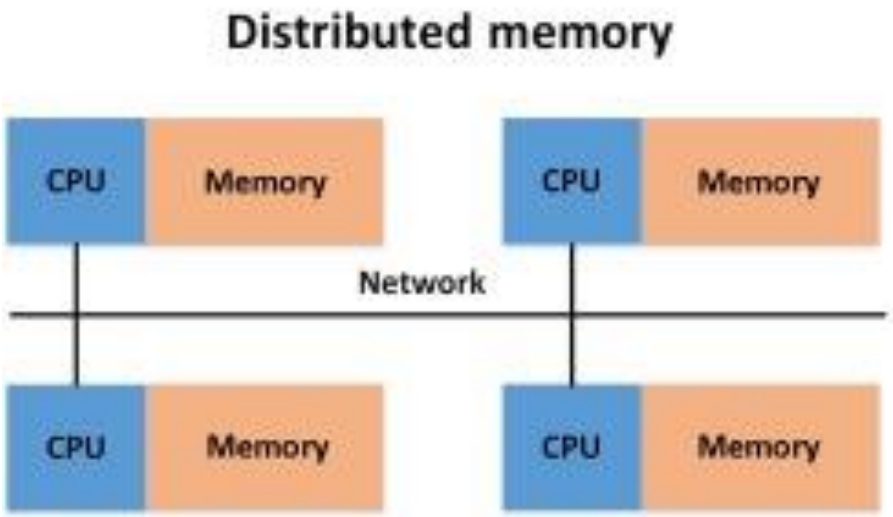
- **Near-/In-Memory Computing (PIM — Processing-in-Memory)**
  - Цель: преодолеть "узкое место памяти" за счёт вычислений рядом с памятью или внутри памяти.
  - Near-Memory: процессор и память близко (например, 3D-stacked HBM + логика).
  - In-Memory Computing: вычисления выполняются прямо в ячейках памяти (например, с помощью RRAM, memristors).
  - Применение: ИИ, нейросети, big data.
  - Пример: Samsung HBM-PIM, Intel Optane с вычислительными возможностями.

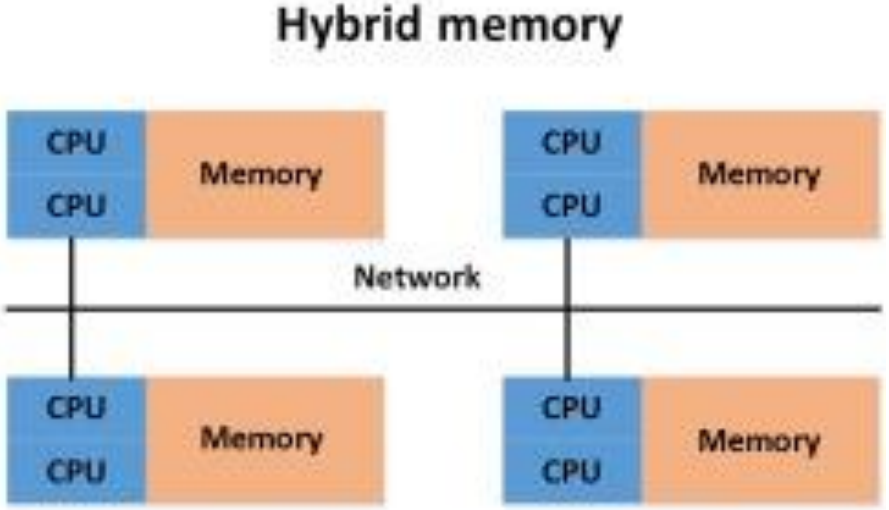


По организации памяти и кода



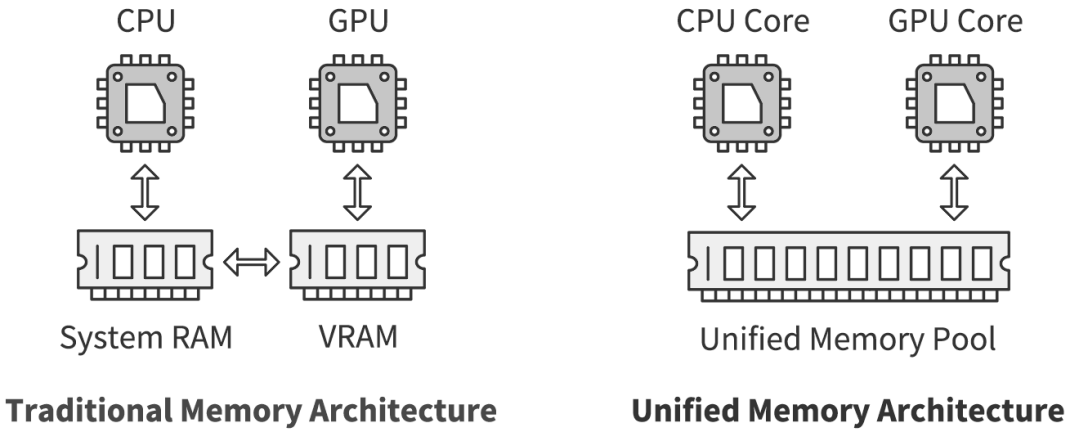




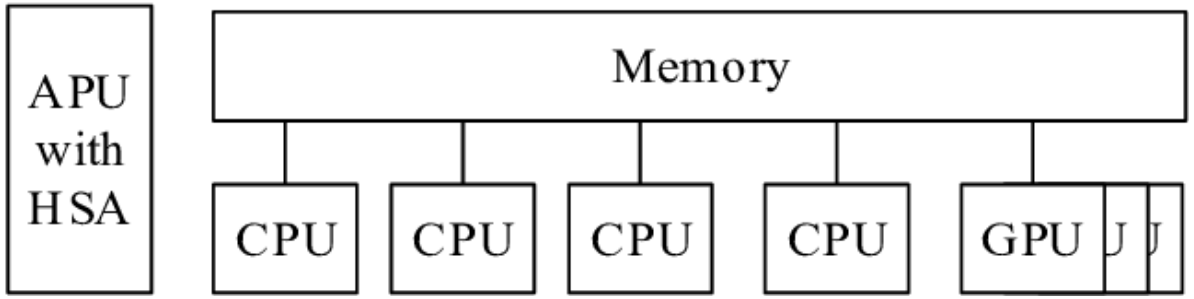


По организации памяти и кода

- **Унифицированная память CPU–GPU (UMA в SoC, hUMA / HSA)**
- **UMA (Unified Memory Architecture):**
  - CPU и GPU разделяют одну физическую память (в SoC: смартфоны, APU).
  - Упрощает обмен данными между CPU и GPU.

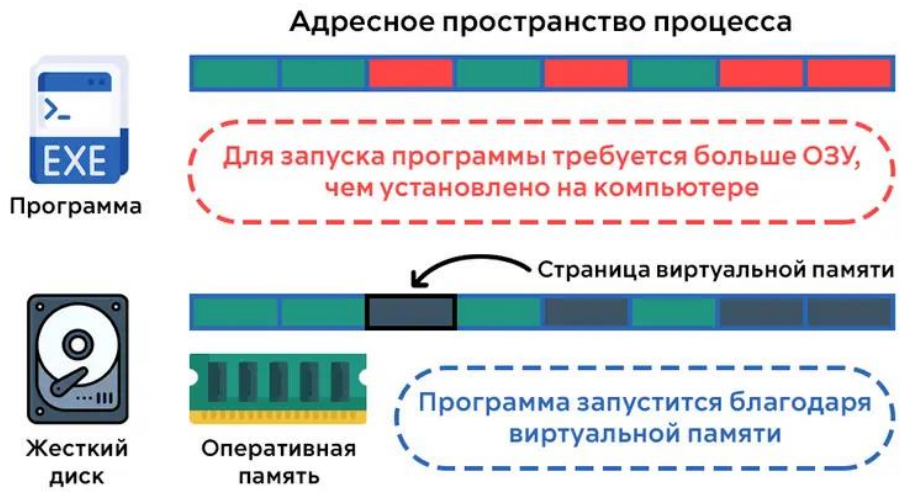


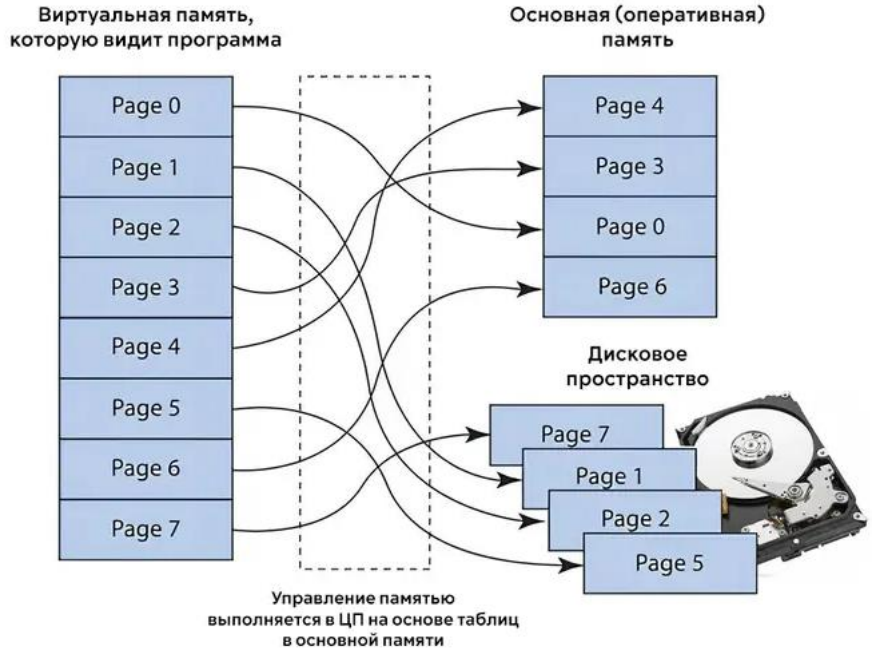
- **hUMA (heterogeneous Uniform Memory Access):**
  - Часть инициативы HSA (Heterogeneous System Architecture).
  - CPU и GPU имеют единое виртуальное адресное пространство.
  - Поддержка когерентности кэшей между CPU и GPU.
  - Позволяет GPU напрямую обращаться к данным CPU и наоборот.
  - Пример: AMD APU (Ryzen с Vega), некоторые ARM Mali-системы.



По организации памяти и кода

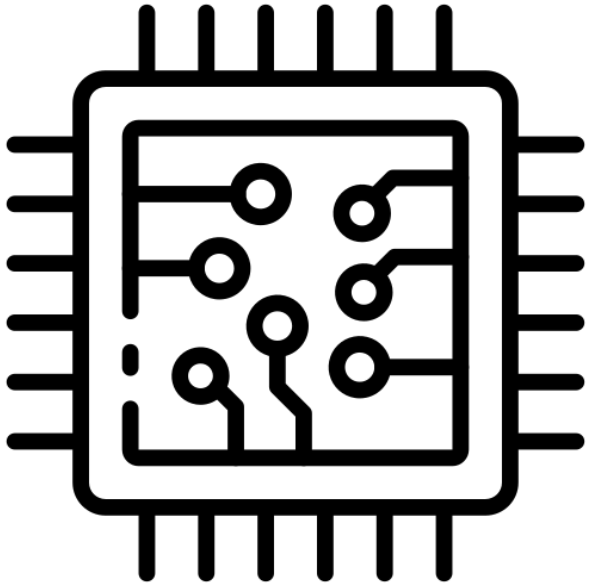
- **Виртуальная память**
- Позволяет программам использовать "виртуальные" адреса, независимо от физической памяти.
- **Типы:**
- **Страничная (paging):**
  - Память делится на фиксированные страницы (обычно 4 КБ).
  - Виртуальные адреса транслируются в физические через таблицы страниц (MMU).
  - Поддерживает подкачку на диск (swap).
  - Используется везде: Linux, Windows, macOS.
- **Сегментная (segmentation):**
  - Память делится на сегменты переменной длины (код, данные, стек).
  - Каждый сегмент имеет базовый адрес и длину.
  - Позволяет гибкое управление, но сложна в реализации.
  - Пример: x86 в защищённом режиме (устаревшее).
- **Странично-сегментная:**
  - Комбинация: сегменты делятся на страницы.
  - Позволяет извлечь преимущества обоих подходов.
  - Пример: IBM System/360, некоторых старых ОС.





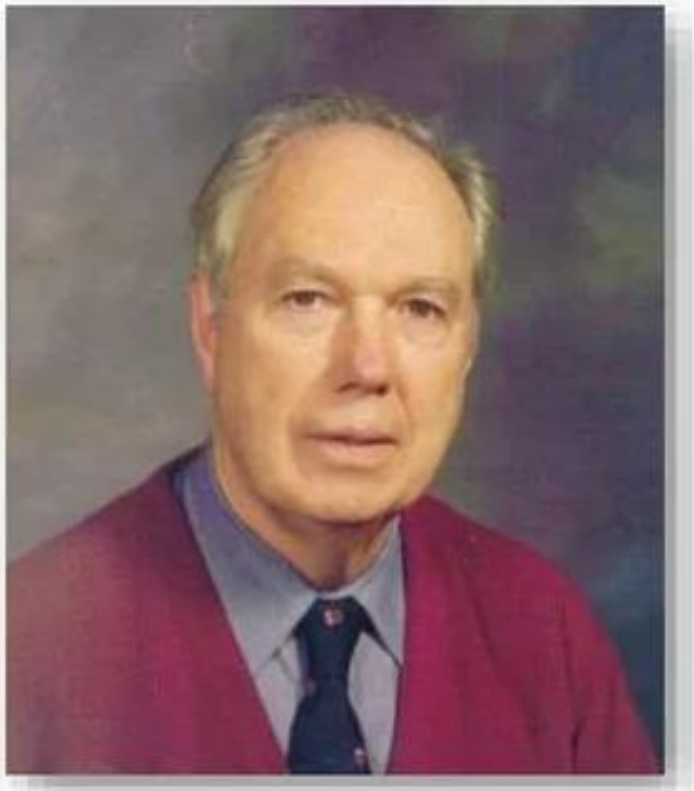
По организации памяти и кода

Концепция	Ключевая особенность	Где используется
Фон Неймана	Единая память для кода и данных	Универсальные CPU
Гарвардская	Раздельные память и шины	Микроконтроллеры, DSP
Модифицированная Гарвардская	L1I/L1D разделены, единое адресное пространство	x86, ARM, RISC-V
Flat memory	Линейное адресное пространство	Современные ОС
Сегментированная память	Адрес = сегмент + смещение	x86 (режимы 16/32 бит)
UMA	Одинаковый доступ к памяти	SMP, десктопы
NUMA	Неоднородный доступ	Многосокетные серверы
CC-NUMA	NUMA + когерентность кэшей	Серверы, HPC
Распределённая память	Нет общей памяти	Кластеры, MPP
DSM	Виртуальная общая память	HPC, распределённые системы
Виртуальная память	Абстракция физической памяти	Все современные ОС
PIM / In-Memory	Вычисления в памяти	ИИ, ускорители
hUMA / HSA	Общая виртуальная память CPU-GPU	APU, SoC, GPGPU



Архитектура ЭВМ. По модели параллелизма (классификация Флинна)

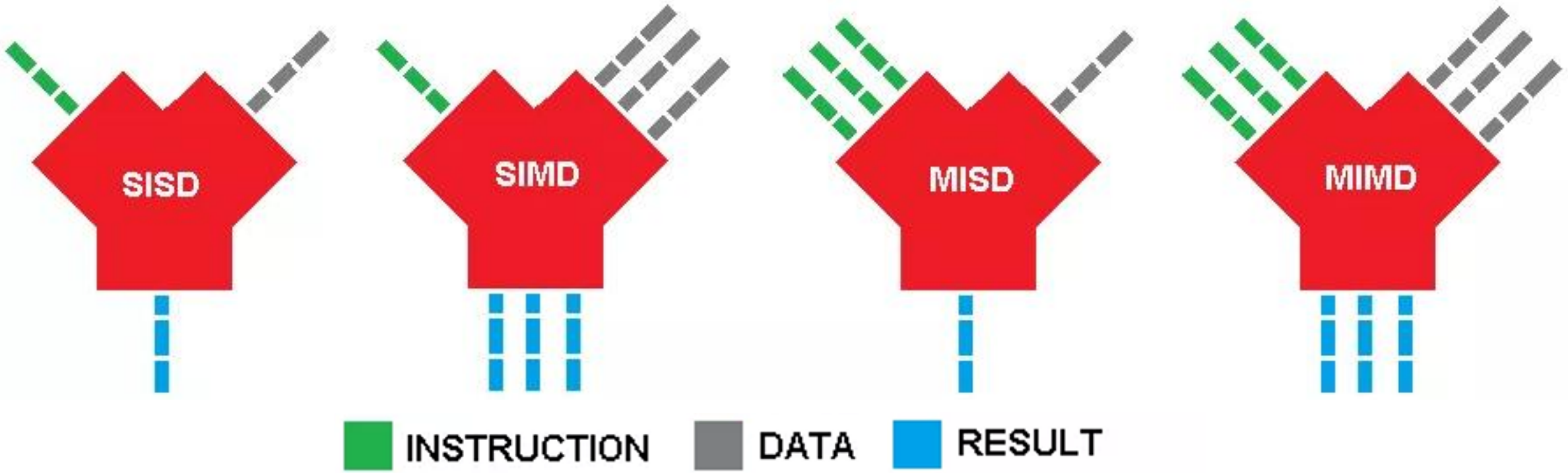
Классификация Флинна



Майкл Флинн

Классификация Флинна (1966 г.) – это фундаментальная схема, которая делит архитектуры компьютеров на четыре класса на основе количества **потоков команд** (Instruction Stream) и **потоков данных** (Data Stream), обрабатываемых одновременно.

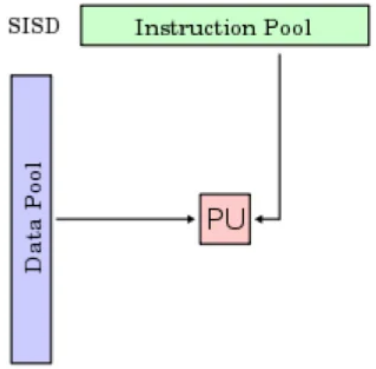
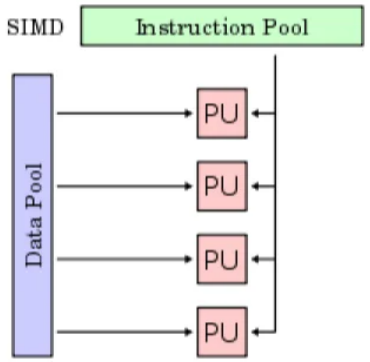
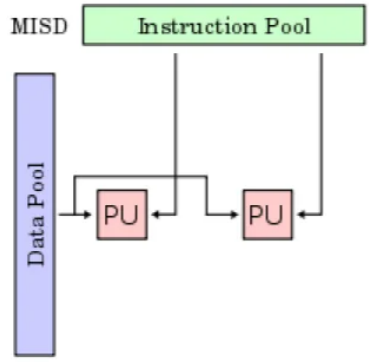
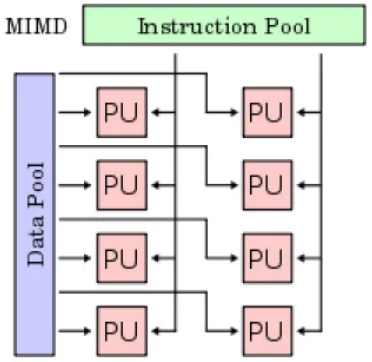
Классификация Флинна



В зависимости от количества потоков команд и потоков данных Флинн выделяет четыре класса архитектур: SISD, MISD, SIMD, MIMD.

SISD (Single Instruction, Single Data): один поток инструкций, один поток данных.
SIMD (Single Instruction, Multiple Data): один поток инструкций, много потоков данных (векторные процессоры, расширения SSE/AVX, NEON; классические векторные Cray).
MISD (Multiple Instruction, Single Data): много потоков инструкций, один поток данных (редко; примеры – системы с избыточностью для отказоустойчивости).
MIMD (Multiple Instruction, Multiple Data): много потоков инструкций и данных (многопроцессорные и многоядерные системы).

Классификация Флинна

	Single data stream	Multiple data streams
Single instr. stream	SISD: фон-Неймановская архитектура. **Один процессор** выполняет один поток команд, оперируя одним потоком данных. В каждый момент времени выполняется лишь одна операция над одним элементом данных (числовым или каким-либо другим значением). 	SIMD: параллелизм на уровне данных. SIMD-компьютеры состоят из **одного командного процессора** (управляющего модуля), называемого контроллером, **и нескольких модулей обработки данных**, называемых процессорными элементами. Управляющий модуль принимает, анализирует и выполняет команды. Если в команде встречаются данные, контроллер рассылает на все процессорные элементы команду, и эта команда выполняется на нескольких или на всех процессорных элементах. Каждый процессорный элемент имеет свою собственную память для хранения данных. В SIMD компьютере управление выполняется контроллером, а «арифметика» отдана процессорным элементам. 
Multiple instr. streams	MISD: систолический массив процессоров, ~~конвейерная архитектура - т.к. данные будут различаться после обработки на каждой стадии в конвейере~~. Несколько функциональных модулей (два или более) выполняют различные операции над одними данными. Отказоустойчивые компьютеры, выполняющие одни и те же команды избыточно с целью обнаружения ошибок. 	MIMD: параллелизм вычислений. **Несколько процессоров**, которые функционируют асинхронно и независимо. В любой момент, различные процессоры могут выполнять различные команды над различными частями данных. MIMD-машины могут быть либо с общей памятью, либо с распределяемой памятью. Обработка разделена на несколько потоков, каждый с собственным аппаратным состоянием процессора, в рамках единственного определённого программным обеспечением процесса или в пределах множественных процессов. Поскольку система имеет несколько потоков, ожидающих выполнения (системные или пользовательские потоки), эта архитектура эффективно использует аппаратные ресурсы. В MIMD могут возникнуть проблемы взаимной блокировки и состязания за обладание ресурсами, так как потоки, пытаясь получить доступ к ресурсам, могут столкнуться непредсказуемым способом. 

Классификация Флинна (SISD, SIMD, MISD, MIMD) – базовая модель классификации архитектур.

SPMD, SIMT, MPMD – современные модели программирования и исполнения, расширяющие или уточняющие классификацию Флинна.

SIMT – фактически реализует SIMD-подобное поведение, но с семантикой многопоточности.

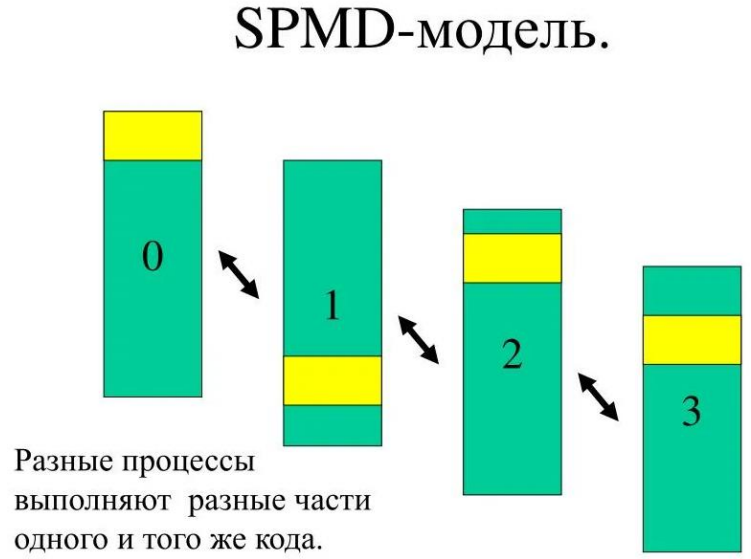
SPMD – наиболее частая модель в параллельных вычислениях, реализуемая на MIMD-архитектурах.

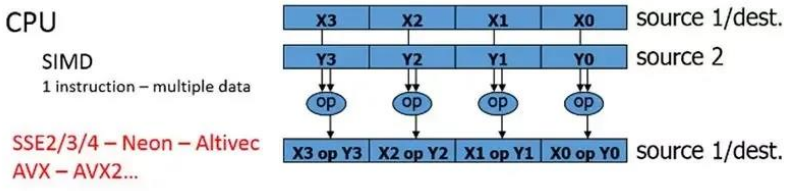
По модели параллелизма

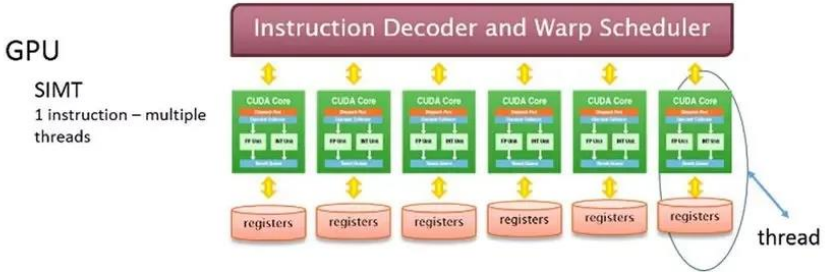
- **SPMD (Single Program, Multiple Data)** – программная модель, при которой одна и та же программа параллельно выполняется на разных процессорах или ядрах с разными данными. Широко используется в параллельных и распределённых системах (MPI, OpenMP, GPU-ядра). **SPMD** — стандарт для параллельных вычислений: гибкий и масштабируемый.

- **SIMT (Single Instruction, Multiple Thread)** – аппаратная модель выполнения в GPU, где одна инструкция применяется к множеству потоков одновременно. Поддерживает дивергенцию ветвлений (разные потоки могут выполнять разные ветки кода), в отличие от классического SIMD. **SIMT** — ключ к высокой производительности GPU, но чувствителен к структуре кода.

- **MPMD (Multiple Program, Multiple Data)** – модель, в которой разные процессоры или узлы выполняют разные программы на своих наборах данных. Используется в гетерогенных и гибридных HPC-системах (например, CPU + GPU + FPGA). **MPMD** — путь к максимальной эффективности в гетерогенных системах, требует сложного управления.

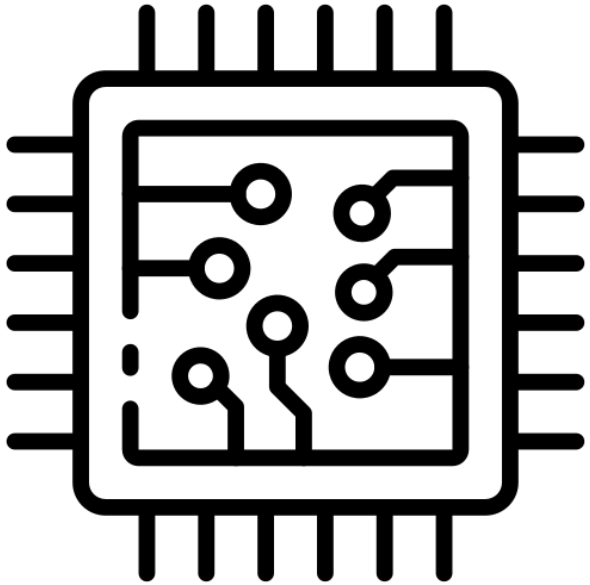






По модели параллелизма

Модель	Тип	Описание	Примеры использования	Примечания
SISD **(Single Instruction, Single Data)**	Архитектура	Один поток команд обрабатывает один поток данных. Последовательная обработка.	Классические однопроцессорные ЭВМ, ранние CPU.	Базовый случай, без параллелизма.
SIMD **(Single Instruction, Multiple Data)**	Архитектура	Одна команда применяется к нескольким данным одновременно.	Векторные процессоры (Cray), SSE/AVX (x86), NEON (ARM), GPU (на уровне warp/wavefront).	Высокая производительность для регулярных данных. Не поддерживает дивергенцию ветвлений.
MISD **(Multiple Instruction, Single Data)**	Архитектура	Несколько команд обрабатывают один поток данных.	Системы с избыточностью (например, отказоустойчивые: три модуля обрабатывают один сигнал).	Теоретически редкая модель, используется в специализированных системах.
MIMD **(Multiple Instruction, Multiple Data)**	Архитектура	Несколько потоков команд и данных. Независимое выполнение.	Многоядерные процессоры, многопроцессорные системы, кластеры, SMP, NUMA.	Наиболее распространённая модель в современных многопроцессорных системах.
SPMD **(Single Program, Multiple Data)**	Программная модель	Одна программа запускается на нескольких узлах/потоках, но работает с разными данными.	MPI-приложения, параллельные вычисления на CPU/GPU.	Частный случай MIMD. Поддерживает условные ветвления.
SIMT **(Single Instruction, Multiple Thread)**	Аппаратная модель	Расширение SIMD: одна инструкция — много потоков, но с возможностьюдивергенции ветвлений.	GPU (NVIDIA CUDA, AMD HIP, OpenCL).	Аппаратно реализуется на GPU. Потоки группируются в warps/wavefronts.
MPMD **(Multiple Program, Multiple Data)**	Программная модель	Разные узлы выполняют разные программы на своих данных.	Гетерогенные HPC-системы (CPU + GPU + FPGA), эмбеддед-системы.	Наиболее гибкая, но сложная в управлении модель.



Архитектура ЭВМ. По способу организации процессора (микроархитектура)

Классификация по способу организации процессора (микроархитектура)

- Обобщая классификацию архитектур ЭВМ **по способу организации процессора (микроархитектуре), можно выделить эволюционную последовательность подходов, направленных на извлечение параллелизма на уровне инструкций** (ILP — Instruction-Level Parallelism) **и потоков** (TLP — Thread-Level Parallelism).
- На начальном этапе доминировали **неконвейерные процессоры**, в которых каждая инструкция выполнялась последовательно, за один или несколько тактов, без перекрытия с другими командами.
- Такие архитектуры просты в реализации, но обладают низкой производительностью при высоких тактовых частотах.

Классификация по способу организации процессора (микроархитектура)

- Следующим шагом стало внедрение **конвейерной обработки (pipeline)**, позволившей разбить выполнение инструкции на стадии и обрабатывать несколько команд одновременно на разных этапах конвейера.
- Это резко повысило пропускную способность.
- Для дальнейшего роста производительности были разработаны **суперскалярные архитектуры**, способные запускать и выполнять несколько независимых инструкций за один такт за счёт дублированных исполнительных блоков.
- Однако эффективность суперскалярности ограничена зависимостями между инструкциями, что привело к появлению **внеочередного исполнения (Out-of-Order Execution, OoO)** и **спекулятивного выполнения** с использованием сложных механизмов **предсказания ветвлений**.
- Эти технологии позволяют процессору динамически переупорядочивать команды и выполнять их по мере готовности данных, маскируя задержки и повышая загрузку исполнительных устройств.

Классификация по способу организации процессора (микроархитектура)

- Параллельно развивались два принципиально разных подхода к планированию инструкций: **динамическое** и **статическое**.
- Динамическое планирование (например, алгоритм Томасуло) реализуется аппаратно внутри процессора и лежит в основе большинства современных CPU (x86, ARM).
- В противоположность этому, **статическое планирование** возлагает ответственность за распараллеливание на компилятор.
- Такой подход используется в архитектурах **VLIW** (Very Long Instruction Word) и **EPIC** (Explicitly Parallel Instruction Computing), где одна «длинная» инструкция содержит несколько независимых операций.
- Хотя это упрощает аппаратуру, оно требует очень «умного» компилятора и плохо адаптируется к непредсказуемым условиям выполнения — что и стало причиной ограниченного успеха платформы Intel Itanium.

Классификация по способу организации процессора (микроархитектура)

- Современная микроархитектура всё больше смещается от универсального ILP к гетерогенности и специализации.
- Широко применяются **гетерогенные ядра** (например, ARM big.LITTLE, Intel Performance/Efficiency cores), сочетающие производительные и энергоэффективные блоки на одном чипе.
- Для масштабирования производительности используются **многоядерность** и **многопоточность**: **SMT** (Simultaneous Multithreading), известный как Hyper-Threading у Intel, позволяет одному физическому ядру исполнять несколько логических потоков, повышая утилизацию ресурсов; в то время как **CMT** (Clustered Multithreading), как в AMD Bulldozer, объединяет ядра в модули с общими ресурсами.
- Наконец, всё большую роль играют **доменно-специализированные ускорители** — GPU для параллельных вычислений, NPU/TPU для нейросетей, DSP для обработки сигналов, DLA для глубокого обучения.
- Таким образом, современная микроархитектура представляет собой сложную, многоуровневую, гибридную систему, сочетающую универсальные CPU-ядра с целым зоопарком специализированных вычислительных блоков для достижения максимальной эффективности в конкретных рабочих нагрузках.

По способу организации процессора (микроархитектура)

- **Неконвейерные** (один цикл на инструкцию/много микрошагов).
- **Конвейерные** (pipeline).
- **Суперскалярные** (несколько инструкций за такт).
- **Внеочередное исполнение** (Out-of-Order, OoO).
- **Спекулятивное исполнение**, предсказание ветвлений.
- **Динамическое/статическое планирование** (Tomasulo, VLIW/EPIC).
- **VLIW/EPIC**: очень длинное слово, статический ILP (Itanium, DSP).
- **Гетерогенные ядра**: big.LITTLE, архитектуры с разнородными исполнительными блоками.
- **Многоядерные, многопоточные**:
  - **SMT/Hyper-Threading**: логические потоки в одном ядре.
  - **CMT**: модульная многопоточность (например, AMD Bulldozer).
- **Доменно-специализированные ускорители**: GPU, TPU/NPU, DSP, DLA.

По способу организации процессора (микроархитектура)

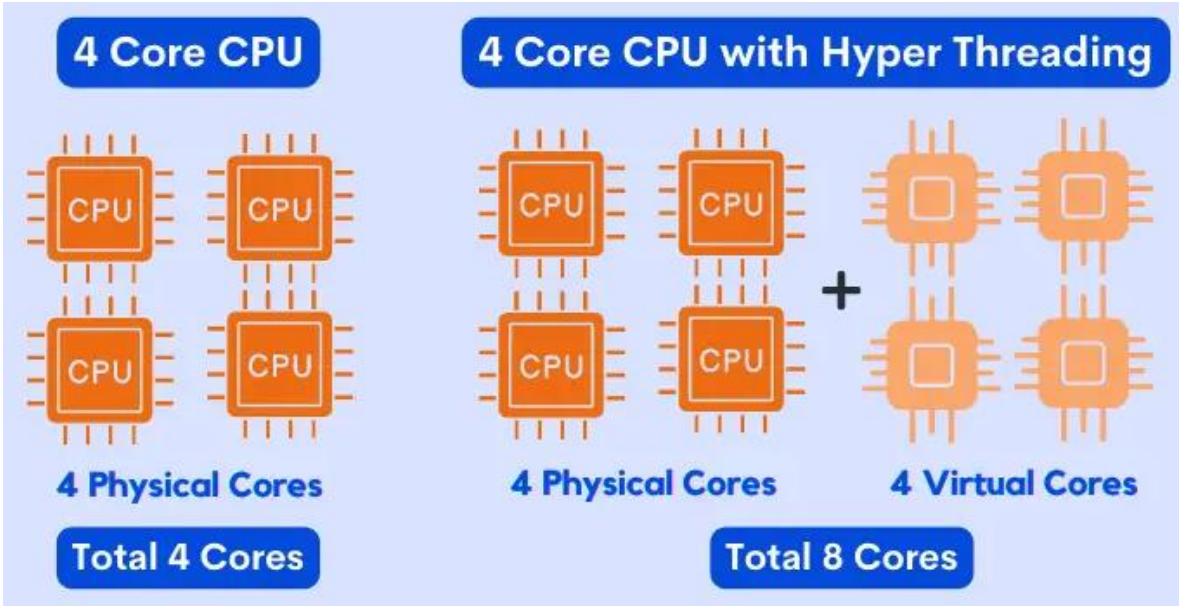
- **Неконвейерные процессоры** — каждая инструкция выполняется за один или несколько тактов последовательно, без перекрытия этапов. Простые, но низкая производительность.
- **Конвейерные (pipeline)** — выполнение инструкций разбито на стадии (выборка, декод, выполнение и т.д.), что позволяет обрабатывать несколько инструкций одновременно на разных стадиях.
- **Суперскалярные** — способны за один такт выбирать и выполнять несколько инструкций параллельно за счёт наличия нескольких исполнительных блоков.
- **Внеочередное исполнение (Out-of-Order, OoO)** — процессор исполняет инструкции не в порядке программы, а в порядке готовности операндов, максимизируя использование ресурсов.

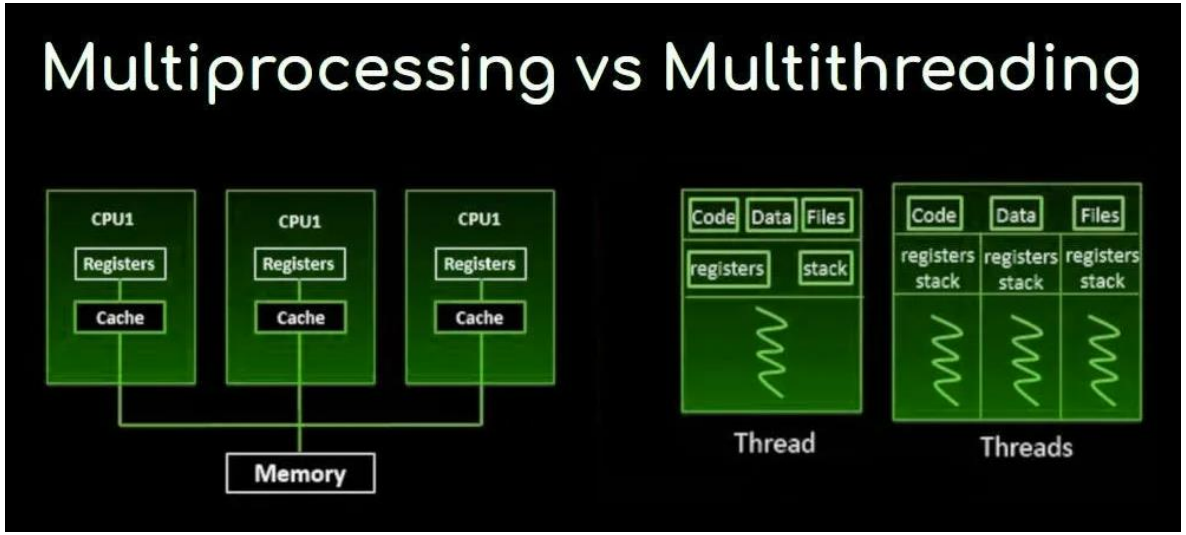
По способу организации процессора (микроархитектура)

- **Спекулятивное исполнение и предсказание ветвлений** — процессор **предсказывает** результат условных переходов и исполняет код заранее. При ошибке — откат. Увеличивает эффективность конвейера.
- **Динамическое планирование (например, Tomasulo)** — аппаратное управление зависимостями и параллелизмом на этапе выполнения.
- **Статическое планирование (VLIW/EPIC)** — параллелизм определяется компилятором, процессор просто выполняет "пакеты" инструкций.
- **VLIW / EPIC** — Very Long Instruction Word / Explicitly Parallel Instruction Computing. Компилятор упаковывает несколько независимых операций в одну длинную инструкцию. Пример: Intel Itanium, DSP-процессоры.

По способу организации процессора (микроархитектура)

- **Гетерогенные ядра** – в одном чипе сочетаются разные по производительности и энерго-потреблению ядра (например, big.LITTLE в ARM: высоко-производительные + энерго-эффективные).
- **Многоядерные, многопоточные:**
  - **SMT / Hyper-Threading** – одно физическое ядро выполняет несколько логических потоков, чередуя их на общих ресурсах.
  - **CMT (Clustered Multithreading)** – как в AMD Bulldozer: часть блоков (например, ALU) разделяется между потоками, часть – дублируется.

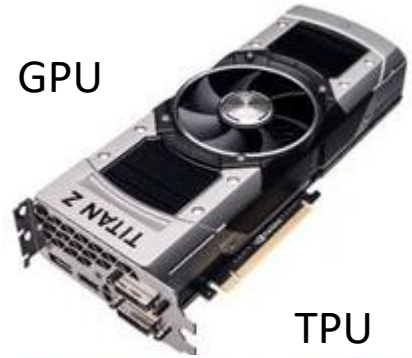




По способу организации процессора (микроархитектура)

- **Доменно-специализированные ускорители** — это специализированные аппаратные блоки, оптимизированные для выполнения определённых классов задач с высокой эффективностью, таких как:
  - **GPU (Graphics Processing Unit)** — ускоряет параллельную обработку графики и массово-параллельных вычислений (GPGPU), используется в играх, научных расчётах и ИИ.
  - **TPU/NPU (Tensor Processing Unit / Neural Processing Unit)** — специализированный ускоритель для операций в нейронных сетях (матричное умножение, свёртки), обеспечивает высокую производительность при низком энергопотреблении.
  - **DSP (Digital Signal Processor)** — оптимизирован для обработки аудио, видео, радиосигналов в реальном времени (в телефонах, модемах, аудиосистемах).
  - **DLA (Deep Learning Accelerator)** — встраиваемый блок для выполнения задач глубокого обучения, часто используется в мобильных и IoT-устройствах.

GPU



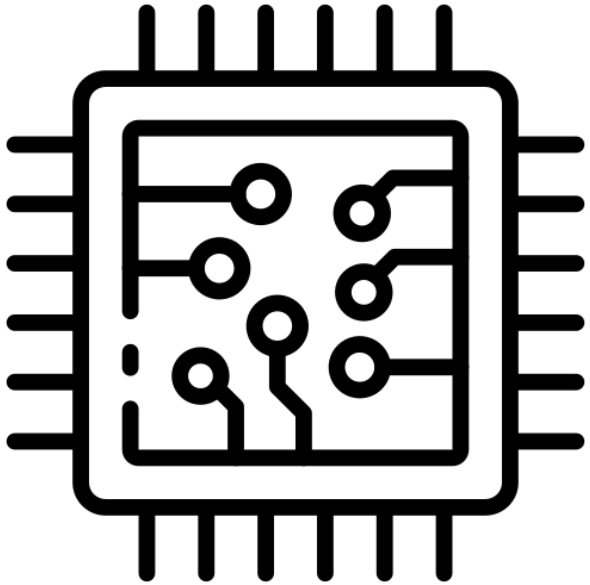
TPU

DSP

DLA

По способу организации процессора (микроархитектура)

Микроархитектурный подход	Описание	Ключевые особенности	Примеры	Преимущества	Недостатки
Неконвейерный	Каждая инструкция выполняется последовательно, без перекрытия этапов.	Один цикл (или много) на инструкцию. Простая логика.	Ранние CPU, микроконтроллеры	Простота, низкое энергопотребление	Низкая производительность
Конвейерный (pipeline)	Выполнение инструкций разбито на стадии (IF, ID, EX, MEM, WB), что позволяет обрабатывать несколько инструкций одновременно.	Перекрытие выполнения инструкций.	MIPS, ARM7, x86 (базовый уровень)	Повышенная пропускная способность	Остановки при зависимостях и ветвлениях
Суперскалярный	Может выполнять **несколько инструкций за один такт** за счёт нескольких исполнительных блоков.	Дублирование функциональных устройств (ALU, FPU и др.).	Intel Core, AMD Ryzen, современные ARM	Высокая производительность	Сложность управления параллелизмом
Внеочередное исполнение (OoO)	Инструкции выполняются **не по порядку**, а когда готовы операнды и ресурсы.	Динамическое переупорядочивание.	Intel Core, Apple M1/M2, высокопроизводительные ARM	Максимальное использование ресурсов	Высокая сложность и энергопотребление
Спекулятивное исполнение + предсказание ветвлений	Процессор **предсказывает** переходы и исполняет код заранее. При ошибке — откат.	Комбинируется с OoO и конвейером.	Все современные CPU	Снижает простои конвейера	Риск ошибок предсказания, уязвимости (Spectre, Meltdown)
Динамическое планирование (Tomasulo)	Аппаратное управление зависимостями и параллелизмом во время выполнения.	Регистровое переименование, reservation stations.	IBM System/360, современные CPU	Позволяет OoO и борьбу с зависимостями	Сложная аппаратура
Статическое планирование (VLIW / EPIC)	Параллелизм определяется **компилятором**, процессор просто выполняет "пакеты" инструкций.	Очень длинные командные слова.	Intel Itanium, TI DSP, MAXQ	Низкая сложность CPU, высокая эффективность при хорошем компиляторе	Чувствителен к задержкам памяти, плохая обратная совместимость
Гетерогенные ядра (big.LITTLE)	В одном чипе — разные по производительности и энергопотреблению ядра.	Динамическое переключение задач между "большими" и "малыми" ядрами.	ARM big.LITTLE, Apple M-series (P/E-ядра)	Энергоэффективность, баланс производительности	Сложность планировщика ОС
SMT / Hyper-Threading	Одно физическое ядро выполняет **несколько потоков**(логических процессоров).	Разделение ресурсов (ALU, кэш), дублирование контекста.	Intel HT, AMD SMT	Лучшее использование ресурсов ядра	Конкуренция за ресурсы, возможен overhead
CMT (Clustered Multithreading)	Группа потоков разделяет ключевые блоки (например, front-end), но имеет отдельные ALU.	Модульный подход к многопоточности.	AMD Bulldozer/Piledriver	Экономия площади чипа	Ограниченный параллелизм между потоками
Векторные/матричные блоки (AMX, Tensor Cores, systolic)	Специализированные блоки для операций с векторами и матрицами.	Высокий ILP для DNN и HPC.	Intel AMX, NVIDIA Tensor Cores, Google TPU	Экстремально высокая производительность в ИИ	Специализация — не универсальны
Асинхронные / GALS (Globally Asynchronous, Locally Synchronous)	Отказ от глобального тактового сигнала. Работа по принципу "готов — отправляй".	Локальная синхронизация, самосинхронизация.	Экспериментальные чипы, IoT, космические системы	Очень низкое энергопотребление, отказоустойчивость	Сложность проектирования, малая экосистема



Архитектура ЭВМ. По типу набора инструкций (ISA)

Классификация По типу набора инструкций (ISA)

- Обобщая классификацию архитектур ЭВМ по типу набора инструкций (ISA), можно выделить несколько ключевых парадигм, каждая из которых отражает определённую философию проектирования и ориентирована на конкретные задачи, эпохи или компромиссы между сложностью аппаратуры, эффективностью кода и производительностью.
- Наиболее влиятельными стали **CISC** (Complex Instruction Set Computer) и **RISC** (Reduced Instruction Set Computer) , хотя сейчас на практике граница между ними со почти стёрлась.
- **CISC-архитектуры**, такие как **x86** и историческая **VAX**, стремились максимизировать выразительность отдельных инструкций, часто встраивая в них сложные последовательности операций — это уменьшало размер программ и облегчало ручное программирование, но усложняло аппаратную реализацию.
- В ответ на это появилась **RISC-философия**, реализованная в таких ISA, как **ARM**, **MIPS**, **SPARC**, **Power** и особенно в открытой RISC-V, где акцент сделан на простоте, регулярности и эффективной конвейеризации: фиксированная длина инструкций, load/store-модель и явное использование регистров позволяют добиваться высокой производительности при относительно низком энергопотреблении.

Классификация По типу набора инструкций (ISA)

- Помимо этих двух доминирующих направлений, существуют специализированные типы ISA.
- **VLIW/EPIC** (например, **Intel Itanium** и DSP **TMS320C6x**) переносят бремя распараллеливания с аппаратуры на компилятор: одна инструкция содержит несколько независимых операций, выполняемых параллельно.
- Хотя такой подход упрощает процессор, он требует исключительно «умного» компилятора и плохо справляется с непредсказуемыми данными, что ограничило его применение в основном цифровой обработкой сигналов.
- В сфере встраиваемых систем распространены **микроконтроллерные ISA**, такие как **AVR**, **PIC** и **ARM Cortex-M** (в режиме Thumb), где важны компактность кода, низкое энергопотребление и простота реализации.
- Они часто используют 16-битные или переменной длины инструкции для экономии памяти.
- Для задач, требующих массового параллелизма, применяются **векторные ISA**.
- Исторически они доминировали в суперкомпьютерах (**Cray, NEC SX**), а сегодня реализуются в виде расширений к основным архитектурам: **AVX-512** для x86, **SVE/SVE2** для ARM и **RVV** (RISC-V Vector Extension).

Классификация По типу набора инструкций (ISA)

- Эти расширения позволяют одной инструкцией обрабатывать десятки или сотни элементов данных, что критично для ИИ, научных вычислений и мультимедиа.
- В виртуальных средах и языковых машинах популярны **стековые ISA**, такие как в **Forth-машинах**, **JVM** или **WebAssembly**, где операнды не указываются явно, а берутся из вершины стека — это упрощает генерацию кода и портируемость, но снижает производительность на реальном железе.
- Наконец, исторически важны **аккумуляторные ISA**, характерные для ранних 8-битных процессоров (Intel 8008/8080, MOS 6502), где одна из операнд всегда подразумевалась — аккумулятор, — что экономило биты в инструкции, но ограничивало гибкость.
- Современные архитектуры почти повсеместно используют **регистровые ISA**, где все операнды явно задаются через регистры, что обеспечивает максимальную гибкость и эффективность для компиляторов и аппаратных оптимизаций.
- Таким образом, эволюция ISA отражает переход от экономии памяти и упрощения программирования к максимизации производительности, энергоэффективности и специализации под конкретные домены вычислений.

По типу набора инструкций (ISA)

- **CISC** (complex instruction set computing или complex instruction set computer): сложные инструкции (x86, VAX).
- **RISC** (reduced instruction set computer): сокращённый, регулярный набор (ARM, MIPS, RISC-V, SPARC, Power).
- **VLIW/EPIC ISA**: Itanium, TMS320C6x.
- **Микроконтроллерные ISA**: AVR, PIC, ARM Cortex-M(Thumb).
- **Векторные ISA**: Cray, NEC SX; векторные расширения к RISC/CISC (RVV, SVE, AVX-512).
- **Стековые ISA**: Forth-машины, Java Virtual Machine, WebAssembly (стековая виртуальная).
- **Аккумуляторные ISA**: ранние 8-битные (Intel 8008/8080, MOS 6502).
- **Регистровые ISA**: явные регистры-операнды (современные RISC/CISC).

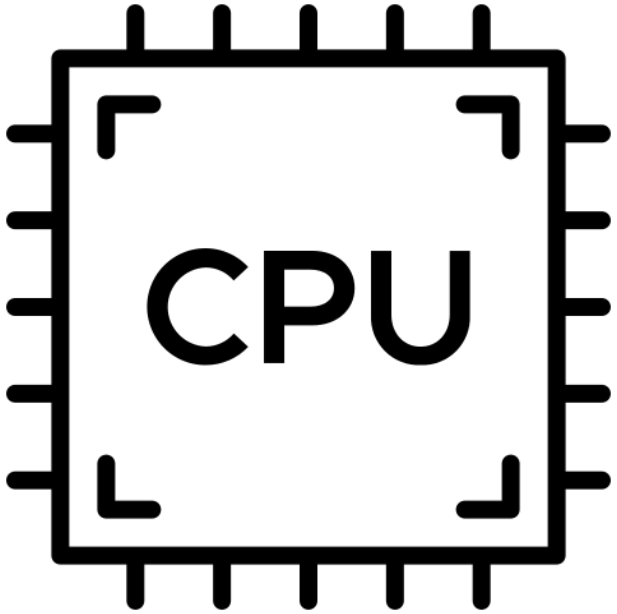
Таблица сравнения по типу ISA

Тип ISA	Описание	Ключевые особенности	Примеры	Преимущества	Недостатки
CISC (Complex Instruction Set Computing)	**Сложные**, многофункциональные инструкции, часто переменной длины.	Одна инструкция может выполнять несколько операций (напр., загрузка + вычисление). Поддержка сложной адресации.	x86, x86-64, VAX (истор.)	Высокая плотность кода, удобство для ассемблера	Сложность декодирования, низкая эффективность конвейера
RISC (Reduced Instruction Set Computing)	**Упрощённый**, регулярный набор инструкций фиксированной длины.	Опора на LOAD/STORE архитектуру, много регистров, простое декодирование.	ARM, RISC-V, MIPS, SPARC, Power	Простота, высокая частота, энергоэффективность, хороший ILP	Более длинный код, требуется больше инструкций
VLIW / EPIC (Very Long Instruction Word / Explicitly Parallel Instruction Computing)	Компилятор упаковывает независимые инструкции в одно "длинное слово".	Параллелизм определяется на этапе компиляции. Нет динамического планирования.	Intel Itanium, TI TMS320C6x, Elbrus	Высокая производительность при хорошем компиляторе	Чувствителен к задержкам, плохая масштабируемость, зависимость от ПО
Микроконтроллерные ISA	**Оптимизированы под низкое энергопотребление**, малый размер кода и простоту.	Часто — урезанные или специализированные версии RISC.	AVR, PIC, ARM Cortex-M (с Thumb/Thumb-2)	Низкое энергопотребление, компактный код	Ограниченная производительность
Векторные ISA	Поддержка операций над массивами (векторами) данных одной инструкцией.	SIMD на стероидах: длинные векторы, маскирование, гибкая длина.	Cray, NEC SX; расширения: AVX-512, SVE/SVE2 (ARM), RVV (RISC-V)	Высокая производительность в HPC, ИИ, мультимедиа	Сложность аппаратуры, не для всех задач
Стековые ISA	Операнды берутся из стека, результаты кладутся туда же. Нет явных регистров в инструкциях.	Простота декодирования, компактные инструкции.	Forth-машины, JVM, WebAssembly	Простая реализация, компактный код	Низкая производительность, трудно оптимизировать
Аккумуляторные ISA	Одна инструкция работает с аккумулятором и одним операндом.	Большинство операций — между аккумулятором и памятью.	Intel 8008/8080, MOS 6502 (Apple II, NES)	Простота	Очень ограниченный параллелизм, узкое место — аккумулятор
Регистровые ISA	Явное указание регистров-источников и регистра-назначения.	Современный стандарт: триадные инструкции (Rd ← R1 op R2).	Все современные RISC (ARM, RISC-V), x86-64 (внутри)	Высокий ILP, удобство для компиляторов	Требует больше бит на кодирование регистров
Capability-based ISA	**ISA с аппаратной поддержкой безопасности**: указатели-возможности (capabilities), защита памяти на уровне ISA.	Контроль доступа на уровне инструкций.	CHERI (на базе MIPS, ARM, RISC-V)	Высокая безопасность (защита от переполнения буфера и т.п.)	Экспериментальная, ограниченная поддержка ПО

Дополнительная классификация по примерам

Категория	ISA	Тип
RISC	ARM, RISC-V, MIPS, Power, SPARC	Регулярный, LOAD/STORE, фиксированная длина
CISC	x86, x86-64, VAX	Сложные инструкции, переменная длина
VLIW/EPIC	Itanium, TI TMS320C6x, Elbrus	Статический ILP, упаковка инструкций
DSP-ориентированные	TMS320, SHARC, Hexagon	Часто VLIW + SIMD, оптимизация под сигналы
Векторные ISA	Cray, NEC SX; SVE, SVE2, RVV, AVX-512	Длинные векторы, масштабируемость
Стековые / виртуальные	JVM, WebAssembly, Forth	Основаны на стеке, платформонезависимы
Аккумуляторные (истор.)	Intel 8008/8080, MOS 6502	Однорегистровая архитектура
Экспериментальные / безопасные	CHERI	Capability-based, защита памяти на уровне ISA

- **RISC vs CISC** — основной исторический контраст. Современные процессоры (включая x86) используют RISC-подобные ядра внутри (микрооперации).
- **VLIW** — переложил сложность с железа на компилятор. Не получил массового успеха из-за жёстких требований к ПО.
- **Векторные расширения (SVE, RVV, AVX-512)** — ключ к производительности в ИИ, научных вычислениях и мультимедиа.
- **WebAssembly / JVM** — виртуальные стековые машины, обеспечивают переносимость и безопасность.
- **CHERI** — перспективное направление: аппаратная безопасность на уровне архитектуры.



CISC/RISC архитектуры

CISC/RISC архитектуры

- Классификация архитектур ЭВМ на **CISC (Complex Instruction Set Computer) и RISC (Reduced Instruction Set Computer)** отражает два фундаментально разных подхода к проектированию набора команд процессора.
- **CISC-архитектуры**, зародившиеся в эпоху дорогой и медленной памяти (1970–1980-е), стремились минимизировать объём исполняемого кода за счёт сложных, многофункциональных инструкций.
- Одна команда могла выполнять целую последовательность операций — например, загрузить данные из памяти, выполнить арифметику и записать результат обратно.
- Это упрощало программирование на ассемблере и снижало требования к объёму памяти, но приводило к сложной, неоднородной и трудно конвейеризуемой микроархитектуре.
- Классическим примером CISC является семейство **x86/x86-64**, которое сохраняет обратную совместимость с инструкциями 1978 года, несмотря на радикальные изменения в микроархитектуре.

CISC/RISC архитектуры

- В ответ на ограничения CISC **в середине 1980-х возникла философия RISC**, основанная на исследованиях в университетах (Stanford, Berkeley).
- Её ключевые принципы: **упрощённый и регулярный набор команд, фиксированная длина инструкций, выполнение большинства операций только с регистрами и отдельные команды загрузки/сохранения (load/store)** для доступа к памяти.
- Такой подход позволяет легко реализовать глубокий конвейер, суперскалярность и внеочередное исполнение, что значительно повышает производительность на такт.
- RISC-архитектуры также предполагают наличие большого количества регистров и активное использование компилятора для оптимизации кода.
- Яркими представителями RISC стали **ARM, MIPS, SPARC, PowerPC** и, в особенности, открытая RISC-V, которая благодаря своей модульности и лицензионной свободе быстро завоёвывает рынок от микроконтроллеров до суперкомпьютеров.

CISC/RISC архитектуры

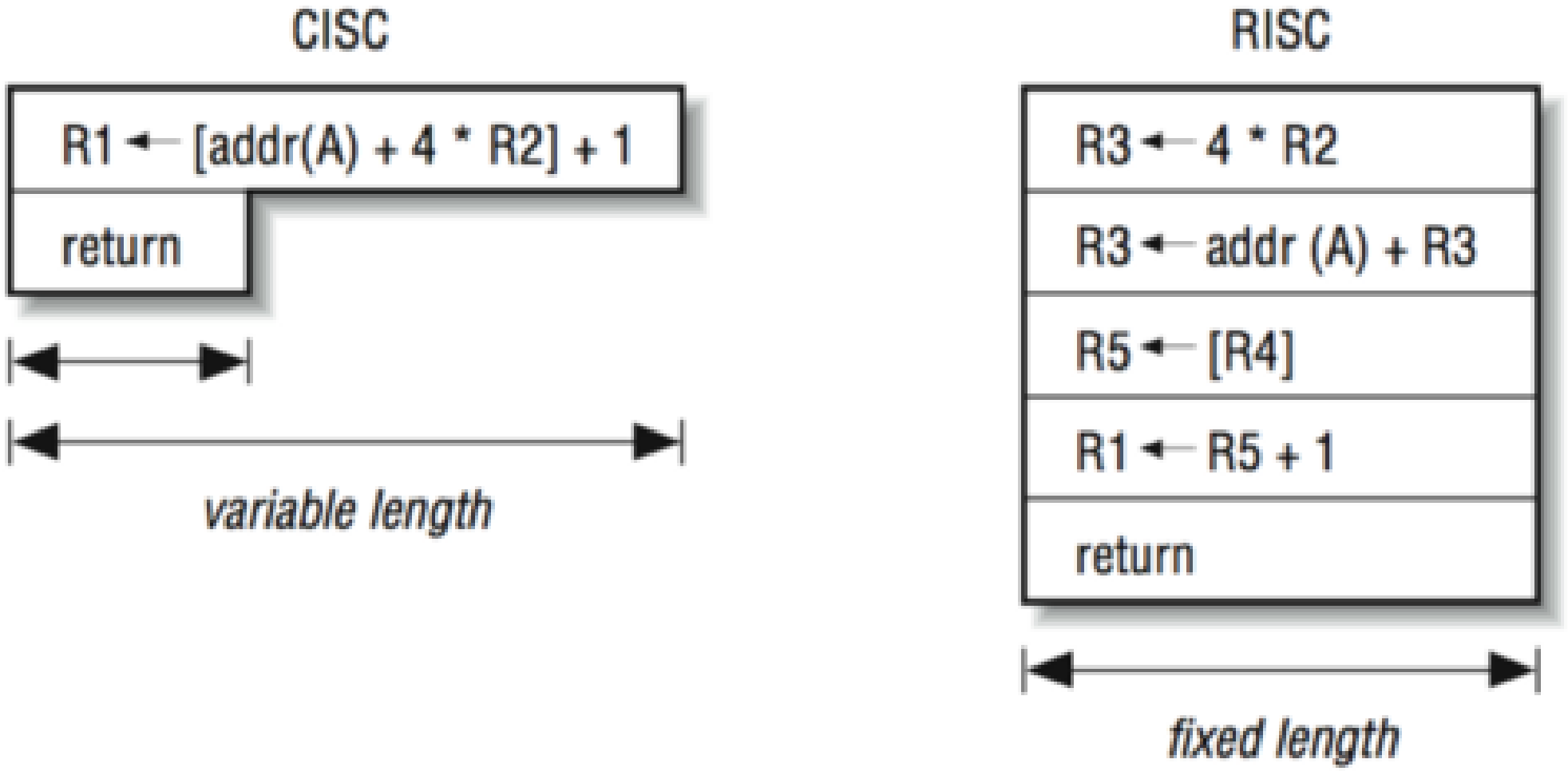
- С течением времени граница между CISC и RISC размылась.
- Современные **x86-процессоры (Intel Core, AMD Ryzen)** на самом деле являются **гидридными системами**: сложные CISC-инструкции декодируются во внутренние **микрооперации (μops)**, которые имеют RISC-подобную структуру и исполняются на суперскалярном, внеочередном ядре. Таким образом, x86 сохраняет совместимость с огромной базой ПО, но внутри работает по RISC-принципам.
- В то же время некоторые RISC-архитектуры (например, ARMv8-A) добавили сложные инструкции для криптографии, векторных вычислений и ИИ, что делает их «менее чистыми», но более практичными.
- Тем не менее, различие остаётся на уровне **ISA**: CISC по-прежнему характеризуется переменной длиной команд и память-память операциями, тогда как RISC придерживается load/store-модели и регулярности.

CISC/RISC архитектуры

- Сегодня выбор между CISC и RISC определяется не столько производительностью, сколько **экосистемой, энергоэффективностью и областью применения**.
- **x86** доминирует в настольных ПК и серверах благодаря совместимости и зрелости ПО.
- **ARM** — абсолютный лидер в мобильных и встраиваемых системах благодаря высокой энергоэффективности и модульности.
- **RISC-V** быстро набирает обороты в IoT, edge-устройствах, академических проектах и даже в национальных инициативах (Китай, Индия), где важна технологическая независимость.
- Таким образом, классификация на CISC и RISC остаётся важной не как техническое противопоставление, а как отражение разных стратегий проектирования: **максимальная совместимость и выразительность против простота, эффективность и открытость.**

CISC/RISC архитектуры

CISC/RISC архитектуры



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.

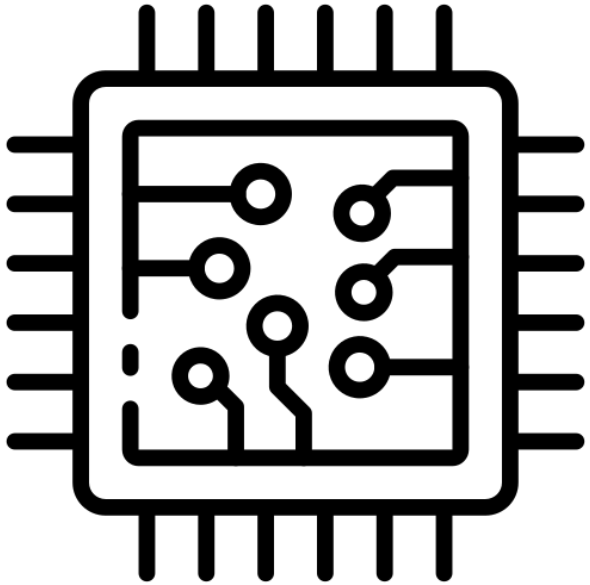
CISC архитектура

- **CISC (complex instruction set computing или complex instruction set computer) — тип процессорной архитектуры**, которая характеризуется следующим набором свойств:
  - нефиксированное значение длины команды;
  - арифметические действия кодируются в одной команде;
  - небольшое число регистров, каждый из которых выполняет строго определённую функцию.
- Методика построения системы команд CISC противостоит методике, применяемой в другом распространённом типе процессорных архитектур — RISC, где используется набор упрощённых инструкций.
- Типичными представителями CISC-архитектуры являются процессоры на основе команд x86, процессоры Motorola MC680x0, процессоры мейнфреймов zSeries.

RISC

- **RISC (reduced instruction set computer — компьютер с набором коротких (простых, быстрых) команд) — архитектура процессора**, в которой быстродействие увеличивается за счёт упрощения инструкций, чтобы их декодирование было более простым, а время выполнения — меньшим. Первые RISC-процессоры даже не имели инструкций умножения и деления. Это также облегчает повышение тактовой частоты и делает более эффективной суперскалярность (распараллеливание инструкций между несколькими исполнительными блоками).

Суперскалярный процессор (англ. *superscalar processor*) — процессор, поддерживающий параллелизм на уровне инструкций (то есть, процессор, способный выполнять несколько инструкций одновременно) за счёт включения в состав его вычислительного ядра нескольких функциональных узлов (блоков) (таких как АЛУ, FPU, умножитель (*integer multiplier*), сдвигающее устройство (*integer shifter*) и другие устройства). Планиро-вание исполнения потока инструкций осуществляется динамически вычислительным ядром (не статически компилятором).



Архитектура ЭВМ. По организации данных и вычислений

Классификация По организации данных и вычислений

- Классификация архитектур ЭВМ по организации данных и вычислений охватывает не только традиционные модели, основанные на последовательном выполнении инструкций, но и альтернативные парадигмы, в которых вычисления инициируются состоянием данных, а не управляющим потоком программы.
- Классическая **машина фон Неймана** реализует вычисления через последовательное извлечение инструкций и данных из общей памяти.
- Внутри такой модели возможны два подхода к управлению: **жёсткая логика (hardwired control),** где схема управления реализована фиксированными логическими элементами (быстро, но негибко), и **микропрограммное управление**, где последовательность микроопераций задаётся микрокодом в ПЗУ (гибко, но медленнее).
- Большинство современных процессоров используют гибрид: жёсткую логику для основных операций и микрокод — для сложных или редких инструкций.

Классификация По организации данных и вычислений

- В противовес фон Неймановской модели существуют **dataflow-архитектуры**, в которых вычисления запускаются не по счётчику команд, а по готовности всех операндов для операции.
- Инструкция (или узел графа данных) активируется, как только все её входные данные становятся доступны.
- Это позволяет естественным образом раскрывать параллелизм на уровне операций без явного управления потоками.
- Dataflow-системы делятся на **статические** (топология графа фиксирована на этапе компиляции) и **динамические** (граф строится во время выполнения).
- Хотя чистые dataflow-процессоры не получили массового распространения, их идеи легли в основу современных графовых вычислительных фреймворков (TensorFlow, PyTorch) и архитектур ускорителей.

Классификация По организации данных и вычислений

- Дальнейшее развитие альтернативных моделей привело к появлению **функциональных и редукционных архитектур**, где программа представляется в виде дерева или графа выражений, а вычисление сводится к последовательной **редукции** (упрощению) этого графа.
- Такие системы, вдохновлённые языками вроде **Haskell** или **SISAL**, избегают побочных эффектов и идеально подходят для параллельного исполнения.
- Близкими по духу являются **систолические массивы** — регулярные структуры из простых процессорных элементов (ПЭ), через которые синхронно «пульсируют» потоки данных.
- Эта архитектура лежит в основе многих ИИ-ускорителей, включая **Google TPU**, где матричные операции выполняются с высокой энергоэффективностью за счёт локальности данных и минимизации обращений к памяти.

Классификация По организации данных и вычислений

- Современные тенденции в организации вычислений всё больше уходят от универсальных последовательных моделей к **специализированным и адаптивным парадигмам**.
- **Реконфигурируемые архитектуры**, такие как **FPGA**, позволяют буквально «перестраивать» аппаратуру под конкретную задачу, обеспечивая баланс между гибкостью программного обеспечения и производительностью ASIC.
- **Асинхронные (безтактовые) процессоры** устраняют глобальный тактовый сигнал, что снижает энергопотребление и позволяет компонентам работать на своей собственной скорости.
- В условиях роста требований к энергоэффективности развиваются **приближённые вычисления (approximate computing)**, где допускается небольшая погрешность результата ради существенного выигрыша в скорости или энергии — это актуально для мультимедиа и ИИ.
- На переднем крае исследований находятся **нейроморфные архитектуры** (Intel Loihi, IBM TrueNorth), имитирующие работу биологических нейронных сетей через спайковые события и тесную интеграцию памяти с вычислениями, а также **квантовые вычислители**, основанные на принципиально иных законах физики.
- Все эти подходы демонстрируют, что будущее вычислений — не в ускорении фон Неймановской машины, а в радикальной переориентации самой модели обработки данных.

По организации данных и вычислений

- **Машины фон Неймана с микрокодом** vs **жёсткой логикой**.
- **Данные потоки (Dataflow)**: вычисление при готовности операндов; статический/динамический dataflow.
- **Редукционные/функциональные** (SISAL, Haskell-машины, графы редукции).
- **Систолические массивы**: регулярные массивы ПЭ с потоками данных (TPU).
- **Реконфигурируемые (FPGA)**: архитектура настраивается под задачу.
- **Асинхронные/безтактные** (clockless) процессоры.
- **Энергетически осведомлённые/приближённые вычисления** (approximate computing).
- **Нейроморфные**: спайковые сети, события, массивы с памятью (Loihi, TrueNorth).
- **Квантовые**: модели кубитов (гейтовые, отжигатели), хотя это уже иная парадигма.

По организации данных и вычислений

- 1. **Императивная фон-неймановская модель**
  - Стандарт: последовательное выполнение команд.
  - Данные и код в памяти, управление — через счётчик команд.
  - Подавляющее большинство современных CPU.
- **2. Dataflow / Streaming**
  - Вместо "что делать дальше" — "что можно сделать сейчас".
  - Используется в потоковых процессорах, GPU, FPGA.
  - Пример: Apache Flink, CUDA streams.
- 3. **Редукционные машины**
  - Основаны на функциональных языках.
  - Выражения редуцируются до значения — без состояния.
  - Подходит для параллельных и безопасных вычислений.
- **4. Систолические массивы**
  - Каждый ПЭ принимает данные, обрабатывает, передаёт дальше.
  - Идеально для свёрток, GEMM (матричное умножение).
  - Ядро TPU и других ИИ-ускорителей.

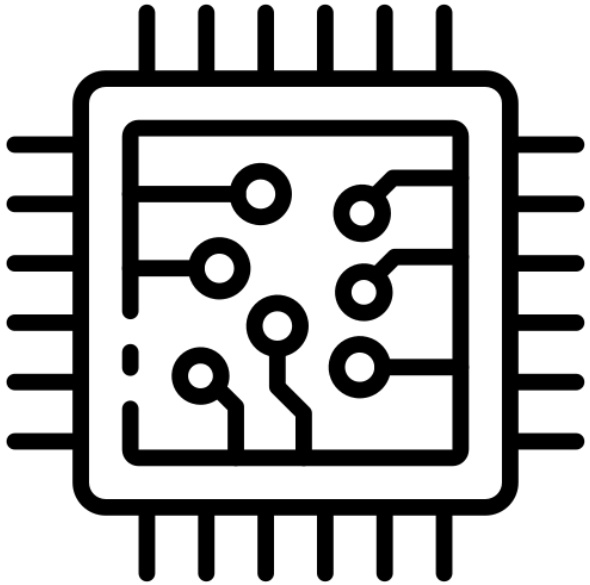
По организации данных и вычислений

- **5. Нейроморфные**
  - Не симулируют нейросети, а воспроизводят биологическую активность.
  - Событийная обработка: только при изменении сигнала.
  - Перспективны для edge-AI и робототехники.
- **6. Квантовые**
  - Парадигма вне классической логики.
  - Гейтовые: как квантовые аналоги логических схем.
  - Отжигатели (аннилинг): для оптимизации (например, D-Wave).
  - Требуют криогенных температур и сложной коррекции ошибок.
- **7. Реконфигурируемые (FPGA, ACAP, CGRA)**
  - FPGA — программируемая логика.
  - ACAP (Adaptable Compute Acceleration Platform) — гетерогенный чип (CPU + GPU + FPGA + AI).
  - CGRA (Coarse-Grained Reconfigurable Architecture) — более высокоуровневые блоки (например, ALU-массивы).
- **8. Approximate / Energy-aware computing**
  - Используется в сенсорах, ИИ, изображениях — где точность не критична.
  - Пример: 8-битные вычисления вместо 32-битных в нейросетях.
  - Цель — максимум вычислений на ватт.

По организации данных и вычислений

Модель/архитектура	Принцип работы	Ключевые особенности	Примеры / применение	Преимущества	Недостатки
Фон Нейман с микрокодом / жёсткой логикой	Императивное выполнение команд по порядку. Микрокод — интерпретатор ISA.	Микрокод: гибкость, совместимость. Жёсткая логика: быстрее, но менее гибкая.	x86 (микрокод), RISC-V (жёсткая логика)	Совместимость, универсальность	Микрокод — медленнее; жёсткая логика — сложна в изменении
Dataflow (поток данных)	Вычисления запускаютсяпри готовности операндов, а не по порядку в программе.	Граф данных; динамический или статический scheduling.	MIT Tagged-Token Dataflow, WaveScalar (эксперименты)	Высокий параллелизм, естественная OoO	Сложность управления, высокие накладные расходы
Редукционные / функциональные машины	Вычисления как редукция выражений (λ-исчисление). Нет состояния.	Основаны на функциональных языках (Haskell, SISAL).	SISAL-компиляторы, GRIP (Manchester), SKIM	Естественный параллелизм, отсутствие побочных эффектов	Сложность реализации, низкая эффективность на традиционных системах
Систолические массивы	Регулярная сетка простых процессорных элементов (ПЭ), данные "проталкиваются" по сети.	Высокая эффективность для матричных операций.	Google TPU, матричные блоки в GPU	Высокая энергоэффективность, предсказуемость	Жёсткая специализация, неуниверсальность
Нейроморфные архитектуры	Моделируют мозг: спайковые нейронные сети, обработка по событиям (event-driven).	Энергоэффективны, работают с асинхронными импульсами.	Intel Loihi, IBM TrueNorth, SpiNNaker, Apple NorthPole	Очень низкое энергопотребление, устойчивость к шуму	Сложность программирования, узкая применимость
Квантовые архитектуры	Используют квантовые биты (кубиты) и суперпозицию/ запутанность.	Не классическая логика. Два подхода: –Гейтовые(как квантовые цепи) –Квантовый отжиг(D-Wave) –Фотонные(Xanadu)	IBM Quantum, Google Sycamore, D-Wave, IonQ	Потенциал взлома шифров, оптимизация, моделирование молекул	Хрупкость состояний, охлаждение, ошибки, масштабирование сложно
Реконфигурируемые архитектуры (FPGA, ACAP, CGRA)	Аппаратная структура настраивается под задачу(логические блоки, межсоединения).	FPGA — мелкозернистая; CGRA — регулярная; ACAP — интегрировано с CPU/GPU.	Xilinx FPGA, AMD Versal (ACAP), CGRA (IMPER, ADRES)	Высокая гибкость, энергоэффективность	Сложность проектирования, высокая стоимость
Approximate / Energy-aware computing	Сознательнаяжертва точностиради энергоэффективности.	Приближённые вычисления в мультимедиа, ИИ, сенсорах.	Approximate adders, neuromorphic sensors, low-voltage режимы	Снижение энергопотребления в 2–10 раз	Неприемлемо для критических задач (финансы, медицина)
Асинхронные / безтактные процессоры	Работают без глобального тактового сигнала.	Сигналы передаются по принципу «готов — приму».	AMULET (ARM-подобный), MiniMIPS, GALS-SoC	Очень низкое энергопотребление, отсутствие джиттера	Сложность проектирования, малая экосистема

Современные архитектуры всё больше уходят от классической фон-неймановской модели к специализированным, энергоэффективным, событийно-ориентированным и реконфигурируемым решениям. Будущее — за гетерогенными системами, сочетающими CPU, GPU, FPGA, ИИ-ускорители и нейроморфные чипы.



Архитектура ЭВМ. По памяти и иерархии хранения

Классификация По памяти и иерархии хранения

- Классификация архитектур ЭВМ по памяти и иерархии хранения отражает стратегию минимизации разрыва между скоростью процессора и относительной медлительностью основной памяти.
- Современные системы строятся как **многоуровневая иерархия**, где каждый уровень отличается по скорости, ёмкости и стоимости: от сверхбыстрых **регистров** и **кэшей L1/L2/L3** до **DRAM**, **энергонезависимой памяти (NVRAM)** и долговременного **хранилища**.
- Ключевым элементом этой иерархии являются кэши — небольшие, но быстрые буферы, хранящие копии часто используемых данных.
- Организация кэшей может быть **inclusive** (все данные L1 присутствуют в L2), **exclusive** (данные не дублируются между уровнями) или **mostly exclusive** (гибридный подход, как в AMD Zen), что влияет на эффективность использования объёма и сложность поддержания когерентности.

Классификация По памяти и иерархии хранения

- Центральной проблемой в многоядерных системах является **когерентность кэшей** — обеспечение согласованности данных, когда одно и то же значение может храниться в кэшах нескольких ядер.
- Для этого используются протоколы на основе состояний: **MSI** (Modified, Shared, Invalid), **MESI** (с добавлением Exclusive), **MOESI** (с Ownership для оптимизации записи) и MESIF (с Forward state для ускорения чтения).
- Механизмы поддержания когерентности реализуются либо через **snooping** (каждое ядро «слушает» шину на предмет изменений), либо через **директорию** (централизованная или распределённая структура отслеживает, где хранится каждая строка кэша).
- Современные масштабируемые архитектуры, такие как **ARM CHI** или **Intel UPI**, используют гибридные или директорийные подходы для эффективной работы в системах с десятками и сотнями ядер.

Классификация По памяти и иерархии хранения

- Эффективность иерархии памяти также зависит от **политик записи и замещения**.
- При **write-through** данные сразу записываются в память, что просто, но медленно; при **write-back** изменения сохраняются только в кэше до вытеснения строки, что повышает производительность.
- Дополнительно применяются стратегии **write-allocate** (при промахе на запись строка загружается в кэш) и **no-write-allocate** (запись идёт напрямую в память).
- Для ускорения доступа к памяти широко используются **механизмы предвыборки (prefetching)**: аппаратные (на основе регулярных паттернов, таких как stride или stream), программные (через инструкции PREFETCH) и даже **машинно-обученные предикторы**, анализирующие поведение приложений.
- Параллельно оптимизируется работа с виртуальной памятью: **многоуровневые страничные таблицы** (до 5 уровней в x86-64 с LA57), **огромные страницы** (huge pages) и **прозрачные огромные страницы** (**THP**) снижают нагрузку на TLB (Translation Lookaside Buffer) — кэш для трансляции виртуальных адресов.

Классификация По памяти и иерархии хранения

- Новые технологии продолжают расширять иерархию памяти вглубь и вширь.
- Появление **энергонезависимой памяти (NVRAM)**, такой как **Intel Optane** (ныне снят с производства) или **NVDIMM-N/P**, создаёт промежуточный уровень между DRAM и SSD, позволяя реализовывать **постоянную память**, доступную по байтовому адресу.
- Стандарт **CXL.mem** (Compute Express Link) открывает путь к когерентному подключению удалённой памяти через высокоскоростные интерконнекты.
- На другом конце спектра — **3D-стекированная память**, такая как **HBM (High Bandwidth Memory)** или историческая **HMC (Hybrid Memory Cube)**, которая размещается в непосредственной близости от процессора или даже на том же кристалле, обеспечивая колоссальную пропускную способность для GPU и ИИ-ускорителей.
- **Таким образом, современная иерархия памяти** — это динамично развивающаяся, многоуровневая, когерентная и всё более гетерогенная система, где баланс между задержкой, пропускной способностью, ёмкостью и энергопотреблением определяет общую производительность вычислительной платформы.

По памяти и иерархии хранения

- **Многоуровневые кэши**: L1/L2/L3, inclusive/exclusive/mostly exclusive.
- **Когерентность**: MSI/MESI/MOESI/MESIF, директория vs snooping.
- **Политики записи**: write-through, write-back; write-allocate/no-write-allocate.
- **Предвыборка**: программная/аппаратная.
- **TLB и страничные таблицы**: многоуровневые, инвертированные.

По памяти и иерархии хранения

Аспект	Описание	Ключевые типы / подходы	Примеры / реализации	Преимущества	Недостатки
Иерархия памяти	Многоуровневая структура от быстрой до медленной памяти.	Регистры → L1 → L2 → L3 → ОЗУ → NVRAM/SSD	Intel Core, AMD Zen, Apple M-series	Баланс скорости, стоимости и объёма	Сложность управления, задержки при промахах
Многоуровневые кэши (L1/L2/L3)	Кэши разных уровней: L1 — самый быстрый, L3 — общий, большой.	L1: разделён (I/D), L2: ядерный, L3: общий (shared)	x86, ARM, RISC-V	Снижение задержек доступа к памяти	Промахи кэша — серьёзное падение производительности
Организация кэшей	Как кэши разных уровней пересекаются по содержимому.	-Inclusive: данные L1 есть в L2/L3 -Exclusive: данные только на одном уровне -Mostly exclusive: компромисс (например, Intel)	Intel (mostly exclusive), IBM POWER (inclusive)	Inclusive — проще когерентность; Exclusive — выше эффективность объёма	Inclusive — избыточность; Exclusive — сложнее управление
Политики записи в кэш	Когда данные записываются в нижележащие уровни.	-Write-through: сразу в память -Write-back: только при вытеснении -Write-allocate: при записи — подгружаем блок в кэш -No-write-allocate: пишем сразу в память, не в кэш	Write-back + write-allocate — стандарт для L1	Write-back — эффективнее по трафику; Write-through — проще	Write-back — сложнее когерентность; Write-through — много трафика
Когерентность кэшей (Cache Coherence)	Гарантирует согласованность данных в кэшах нескольких ядер.	Протоколы: MSI, MESI, MOESI, MESIF (Intel) Способы:Snooping(шина),Directory-based(таблица)	AMD Zen (snoop), Intel (MESIF + snoop), CC-NUMA (directory)	Без когерентности — ошибки в многопоточных программах	Накладные расходы на обновление/инвалидацию
TLB и страничные таблицы	Быстрый кэш виртуальных адресов (TLB), многоуровневые таблицы страниц.	- Многоуровневые (4-level в x86-64) - Инвертированные (hashed page tables) - Huge pages (2MB, 1GB) - THP (Transparent Huge Pages)	x86-64 (LA57), ARM64 (48/52 бит), Linux THP	Ускорение трансляции адресов, меньше TLB-промахов	Сложность управления, фрагментация памяти
Предвыборка (Prefetching)	Автоматическая загрузка данных в кэш до их использования.	- Аппаратная: stride, stream - Программная:prefetch - ML-based: обучение шаблонов доступа	Intel Data Layout Turbo, AMD, Apple M2	Снижает задержки памяти	Ложные предвыборки — лишний трафик
Persistent Memory / NVRAM	Память, сохраняющая данные при выключении, между RAM и SSD.	- Intel Optane (снят с производства) - NVDIMM-N (DRAM + резерв) - NVDIMM-P (нативная) - CXL.mem (через CXL)	Intel Optane DC PMM, Samsung CXL memory	Быстрее SSD, сохраняет данные	Высокая стоимость, ограниченная ёмкость
HBM / 3D-DRAM / HMC	Высокопроизводительная память, соединённая вертикально с процессором.	- HBM (High Bandwidth Memory) - HMC (Hybrid Memory Cube, истор.) - 3D-stacked DRAM	GPU (NVIDIA, AMD), AI-ускорители (TPU), Intel Foveros	Очень высокая пропускная способность, низкое энергопотребление	Сложное производство, дороговизна
CXL (Compute Express Link)	Протокол для когерентного доступа к памяти и устройствам через PCIe.	CXL.mem — доступ к удалённой памяти CXL.cache — кэширование для ускорителей	Современные серверы (Intel Sapphire Rapids, AMD Genoa)	Расширение памяти, когерентность CPU–GPU–FPGA	Требует поддержки на уровне чипсета и BIOS

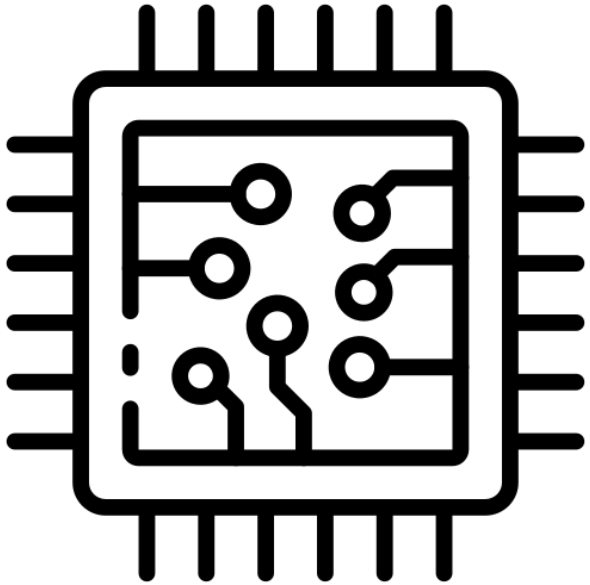
Краткие пояснения ключевых понятий

- **1. Когерентность кэшей: протоколы**
  - **MSI**: Modified, Shared, Invalid — базовый.
  - **MESI**: добавлен Exclusive — позволяет читать без рассылки.
  - **MOESI**: добавлен Owned — поддержка когерентности при записи.
  - **MESIF (Intel)**: Forward — указывает, кто отвечает за данные.
  - **Snooping** — все ядра "подслушивают" шину.
  - **Directory-based** — централизованная таблица отслеживает состояние блоков (масштабируется лучше).
- **2. TLB и страничные таблицы**
  - **TLB** (Translation Lookaside Buffer) — кэш таблиц страниц.
  - **Многоуровневые таблицы**: 4 уровня в x86-64 (PTI).
  - **Huge pages**: 2 МБ или 1 ГБ — меньше записей в TLB.
  - **THP** (Transparent Huge Pages) — автоматическое использование больших страниц (в Linux).
- **3. Предвыборка (Prefetching)**
  - **Stream/Stride**: предсказывает линейный доступ.
  - **Correlation**: учитывает связь между адресами.
  - **ML-based**: нейросети предсказывают доступ (экспериментальные).
- **4. NVRAM и Persistent Memory**
  - **NVDIMM-N**: DRAM + резервное питание + Flash.
  - **NVDIMM-P**: нативная nevolatile-память.
  - **CXL.mem**: позволяет CPU использовать память на другом узле как "локальную".
- **5. HBM / 3D-DRAM**
  - **HBM**: стек из чипов DRAM, соединённых через TSV (Through-Silicon Via) и интерпосер.
  - Используется в GPU, AI-ускорителях (NVIDIA A100, AMD Instinct).
  - **HMC** (Hybrid Memory Cube) — устаревшая технология, заменена HBM.

Иерархия памяти (сверху вниз)

Уровень	Тип	Время доступа	Примерная ёмкость	Назначение
1	**Регистры**	1 такт	128–512 Б	Самые быстрые, в CPU
2	**L1-кэш**	1–4 такта	32–64 КБ (на ядро)	Хранение активных инструкций и данных
3	**L2-кэш**	10–20 тактов	256 КБ – 1 МБ	Поддержка L1
4	**L3-кэш**	30–50 тактов	8–256 МБ (общий)	Общий кэш для всех ядер
5	**ОЗУ (DRAM)**	100–300 нс	8–512 ГБ	Основная рабочая память
6	**Persistent Memory (NVRAM)**	~100–500 нс	До ТБ	Долговременная память с RAM-задержками
7	**SSD / Storage**	10–100 мкс	ТБ–ПБ	Долговременное хранение

Современные системы стремятся к глубокой иерархии, когерентности, предсказуемости и гибкости. Технологии вроде CXL, HBM, NVRAM и ML-based prefetching стирают границы между памятью и хранилищем, открывая путь к memory-centric computing.



Архитектура ЭВМ. По вводу-выводу и шинам

Классификация По вводу-выводу и шинам

- Классификация архитектур ЭВМ по вводу-выводу (I/O) и шинам охватывает как методы взаимодействия процессора с периферийными устройствами, так и топологию соединений между компонентами системы.
- На базовом уровне выделяют три основных механизма передачи данных: **программно-управляемый I/O**, при котором CPU напрямую читает и записывает данные через циклы опроса; **прерывания**, позволяющие устройствам сигнализировать о готовности данных и освобождать процессор от постоянного опроса; и **прямой доступ к памяти (DMA)**, при котором контроллер устройства напрямую обменивается данными с оперативной памятью без участия CPU, что критически важно для высокопроизводительных задач, таких как сетевая передача или видеопотоки.
- Эти механизмы могут комбинироваться: например, DMA-передача завершается прерыванием для уведомления ОС.

Классификация По вводу-выводу и шинам

- Способ адресации устройств также различается.
- В архитектуре **x86** традиционно используется **порт-I/O** — отдельное адресное пространство для регистров устройств, доступ к которому осуществляется специальными инструкциями IN/OUT.
- В большинстве других архитектур (ARM, RISC-V) и в современных x86-системах преобладает **память**, **отображённая на устройства (MMIO — Memory-Mapped I/O),** где регистры периферии отображаются в адресное пространство памяти и доступны через обычные операции загрузки/сохранения.
- MMIO упрощает программирование и позволяет использовать кэширование и предвыборку, но требует тщательного управления правами доступа.
- Для обеспечения безопасности и изоляции в виртуализированных средах применяется **IOMMU** (Input–Output Memory Management Unit), такой как Intel VT-d или AMD-Vi, который транслирует физические адреса устройств в гостевые адреса ВМ, поддерживает **ATS** (Address Translation Services) и **PRI** (Page Request Interface), а также позволяет реализовывать **SR-IOV** (Single Root I/O Virtualization) — технологию, дающую виртуальным машинам прямой, изолированный доступ к физическим ресурсам устройства.

Классификация По вводу-выводу и шинам

- Современные системы строятся на сложных **иерархиях шин и фабрик**, которые соединяют ядра, память, контроллеры и внешние устройства.
- На уровне SoC (System-on-Chip) доминируют стандарты **AMBA** от ARM: **AXI** (высокопроизводительный канал), **AHB** (высокоскоростная шина) и **APB** (низкоскоростная периферия).
- В масштабируемых многопроцессорных системах используются межсокетные интерконнекты: **Intel UPI** (Ultra Path Interconnect) и **AMD Infinity Fabric**, обеспечивающие когерентный обмен данными между процессорами.
- Для подключения внешних устройств стандартом де-факто стал **PCI Express** (PCIe), эволюционировавший от версии 3.0 до 6.0 с экспоненциальным ростом пропускной способности.
- Над PCIe строится **CXL** (Compute Express Link) — протокол следующего поколения, поддерживающий три режима: **CXL.io** (совместимость с PCIe), **CXL.cache** (когерентный кэш для ускорителей) и **CXL.mem** (прямой доступ к памяти), что позволяет создавать пулы разделяемой памяти и гибко распределять ресурсы.

Классификация По вводу-выводу и шинам

- На уровне дата-центров и суперкомпьютеров используются ещё более мощные **кластерные интерконнекты**, такие как **InfiniBand** (с версиями HDR, NDR) и **Ethernet с RoCE** (RDMA over Converged Ethernet), обеспечивающие микросекундные задержки и высокую пропускную способность для распределённых вычислений.
- Для специализированных систем, особенно в ИИ, применяются проприетарные фабрики: **NVIDIA NVLink** и **NVSwitch** обеспечивают ультравысокоскоростное соединение между GPU и CPU (например, в архитектуре Grace Hopper), а **NVLink-C2C** расширяет эту модель на чиплеты и SoC.
- Таким образом, **современная архитектура I/O — это многоуровневая, стандартизированная, но при этом гибкая экосистема**, где шины и фабрики оптимизированы под конкретные задачи: от энергоэффективной периферии в микроконтроллерах до масштабируемых, когерентных, низколатентных сетей в экзафлопсных суперкомпьютерах.

По вводу-выводу и шинам

Категория	Подход / технология	Описание	Примеры / применение	Преимущества	Недостатки
Методы I/O	**Программно-управляемый I/O**	CPU напрямую читает/пишет в устройства.	Простые контроллеры, драйверы	Простота	Высокая нагрузка на CPU
	Прерывания (Interrupts)	Устройство уведомляет CPU о готовности данных.	Клавиатура, сетевая карта	Освобождает CPU от опроса	Накладные расходы при частых прерываниях
	DMA (Direct Memory Access)	Устройство напрямую читает/пишет в ОЗУ без участия CPU.	Диски, GPU, сетевые адаптеры	Высокая пропускная способность, низкая нагрузка на CPU	Сложность управления буферами, когерентность кэшей
Отображение устройств	**MMIO (Memory-Mapped I/O)**	Регистры устройств отображаются в адресное пространство памяти.	ARM, x86 (в современных системах)	Единый механизм доступа (load/store)	Занимает адресное пространство
	Порт I/O (Port-Mapped I/O)	Отдельное адресное пространство для устройств (in/out-инструкции).	x86 (наследие: 0–64K портов)	Не конфликтует с памятью	Ограниченные инструкции, устарело
IOMMU и виртуализация I/O	**IOMMU (VT-d, AMD-Vi)**	Переводит физические адреса DMA в гостевые (аналог MMU для устройств).	Intel VT-d, AMD-Vi	Безопасность, изоляция в виртуализации	Дополнительная задержка
	ATS / PRI (Address Translation / Page Request Interface)	Разрешает устройствам кэшировать TLB-записи (с CXL/PCIe).	CXL, PCIe ATS	Снижает задержки трансляции	Требует поддержки со стороны устройства
	SR-IOV (Single Root I/O Virtualization)	Одно физическое устройство делится на несколько виртуальных (VF).	Сетевые карты, GPU в облаке	Высокая производительность в виртуальных средах	Требует сложного драйвера и настройки
Локальные шины / NoC	**AMBA (AXI, AHB, APB)**	Семейство шин ARM для SoC. AXI — высокоскоростной, APB — для периферии.	ARM-SoC (Cortex, Mali)	Масштабируемость, стандарт де-факто	Сложность при проектировании
	ARM CMN (Coherent Mesh Network), CHI	Coherent NoC для многоядерных систем. Поддерживает когерентность.	ARM Neoverse, серверные SoC	Масштабируемость до сотен ядер	Высокая сложность
Межсокетные/ межпроцессорные	**QPI / UPI (Intel)**	Высокоскоростная шина между CPU-сокетами.	Intel Xeon (QPI → UPI)	Поддержка NUMA, когерентность	Проприетарная, заменяется CXL
	Infinity Fabric (AMD)	Единая шина для соединения ядер, кэшей, памяти, PCIe-контроллеров.	AMD EPYC, Ryzen, Instinct	Гибкость, высокая пропускная способность	Зависимость от архитектуры Zen
Высокоскоростные внешние шины	**PCIe (3.0/4.0/5.0/6.0)**	Последовательная шина для подключения GPU, SSD, NIC.	PCIe x16 (GPU), M.2 NVMe	Высокая пропускная способность, универсальность	Электрические ограничения на высоких скоростях
	CXL (Compute Express Link)	Основан на PCIe, но добавляеткогерентный доступ к памяти и кэшу. Поддерживает: -CXL.io(как PCIe) -CXL.cache(устройство кэширует CPU-память) -CXL.mem(CPU доступ к памяти на устройстве)	Intel Sapphire Rapids, AMD Genoa	Память-пулинг, когерентность CPU–ускоритель, disaggregation	Новая экосистема, требует поддержки BIOS/UEFI
	NVLink / NVSwitch	Высокоскоростная шина NVIDIA для GPU–GPU и CPU–GPU.	NVIDIA A100, H100, Grace Hopper	Пропускная способность в 5–10× выше PCIe	Проприетарная, только для NVIDIA
	NVLink-C2C (Chip-to-Chip)	Интерфейс для соединения чипов в одном пакете (аналог UCIe).	Grace Hopper Superchip	Очень высокая плотность связи	Только в собственных системах NVIDIA
Кластерные интерконнекты	**InfiniBand (HDR, NDR)**	Сверхнизкая задержка, RDMA, масштабируемость.	HPC, AI-кластеры (NVIDIA Quantum-2)	Лучшая производительность для HPC/AI	Высокая стоимость, проприетарность
	Ethernet RoCE (RDMA over Converged Ethernet)	RDMA поверх Ethernet.	Облачные центры, дата-центры	Совместимость с Ethernet, низкая задержка	Требует lossless-сети (PFC, ECN)
	Проприетарные фабрики	Собственные интерконнекты (Google, Amazon, Meta).	Google Jupiter, Meta UFM	Максимальная оптимизация под задачу	Закрытые, нестандартные

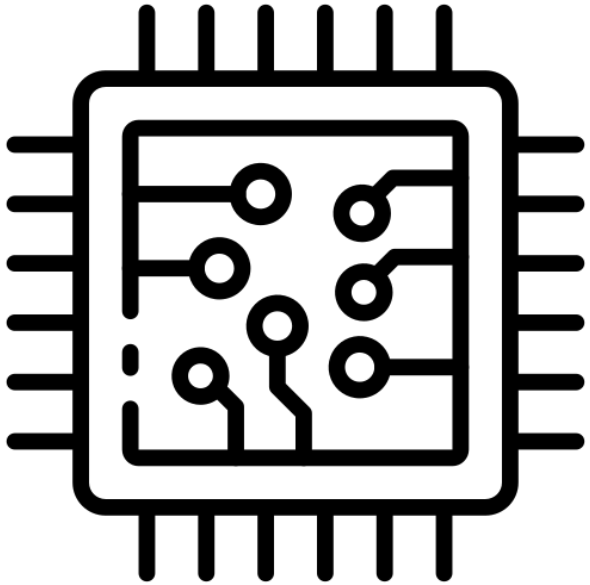
Краткие пояснения ключевых технологий

- **1. DMA и IOMMU**
  - DMA позволяет устройствам (например, GPU) напрямую читать/писать в память.
  - IOMMU (например, Intel VT-d, AMD-Vi) защищает систему: ограничивает DMA только разрешёнными областями памяти — критично для виртуализации.
- **2. MMIO vs Port I/O**
  - MMIO — современный стандарт: устройства "выглядят" как участки памяти.
  - Port I/O — устаревший способ (в x86 используется редко, в основном для совместимости).
- **3. SR-IOV**
  - Позволяет одному физическому устройству (например, сетевой карте) выступать как множество виртуальных функций (VF), каждая — для отдельной виртуальной машины. Высокая производительность без виртуального коммутатора.
- **4. CXL — следующее поколение**
  - CXL.io = PCIe с улучшениями.
  - CXL.cache = ускорители могут кэшировать данные CPU.
  - CXL.mem = CPU может использовать память на ускорителе как "удалённую RAM".
  - Цель: disaggregated memory, memory pooling, гетерогенные системы.
- **5. NVLink и Grace Hopper**
  - NVLink — основной конкурент PCIe и CXL в GPU-кластерах.
  - Grace Hopper — CPU (Grace) и GPU (Hopper) соединены через NVLink-C2C с пропускной способностью до 900 ГБ/с.
- **6. InfiniBand vs RoCE**
  - InfiniBand — лидер в HPC: низкая задержка, RDMA, масштабируемость.
  - RoCE — альтернатива на базе Ethernet: дешевле, но требует тонкой настройки сети.

Иерархия шин и фабрик (от ядра к кластеру)

Уровень	Технология	Назначение
Внутри чипа	AMBA, CMN, CHI, NoC	Соединение ядер, кэшей, контроллеров
Между чипами (в пакете)	NVLink-C2C, UCIe, Infinity Fabric	Соединение CPU, GPU, памяти в одном корпусе
Внутри узла	PCIe, CXL, UPI, Infinity Fabric	Подключение GPU, SSD, NUMA-связь
Между узлами	InfiniBand, RoCE, Ethernet	Кластеры, дата-центры, HPC

Современные системы требуют **высокой пропускной способности, низкой задержки, когерентности и гибкости**. Технологии вроде **CXL, NVLink, IOMMU, SR-IOV и RDMA** становятся стандартом для **HPC, AI, облачных платформ и гетерогенных архитектур**.



Архитектура ЭВМ. По организации многопроцессорности

Классификация По организации многопроцессорности

- Классификация архитектур ЭВМ по организации многопроцессорности отражает способы объединения вычислительных ресурсов для повышения производительности, масштабируемости и отказоустойчивости.
- На базовом уровне находятся **однопроцессорные системы**, где параллелизм достигается за счёт суперскалярности, внеочередного исполнения и многопоточности (SMT).
- Следующим этапом стала **симметричная мультипроцессорность (SMP)**, в которой несколько идентичных процессоров имеют равный доступ к общей памяти и управляются одной ОС.
- В классических SMP-системах используется **UMA (Uniform Memory Access)** — единое адресное пространство с одинаковым временем доступа ко всей памяти.
- Такие системы просты в программировании, но масштабируются плохо из-за конкуренции за шину памяти.

Классификация По организации многопроцессорности

- Для преодоления ограничений SMP были разработаны архитектуры с **неоднородным доступом к памяти (NUMA)** и её кэш-когерентной разновидностью **CC-NUMA**.
- В NUMA каждый процессор (или группу ядер) имеет локальную память, а доступ к «удалённой» памяти другого узла осуществляется через высокоскоростной межсоединитель (например, AMD Infinity Fabric или Intel UPI).
- Это позволяет строить масштабируемые серверы с десятками и сотнями ядер, но требует от программ учитывать топологию памяти для эффективного размещения данных.
- На чиповом уровне NUMA-принципы реализуются в **многоядерных процессорах (CMP — Chip Multi-Processing)**, где ядра объединены на одном кристалле или в одном корпусе, а в современных **chiplet-архитектурах** (например, AMD с CCD+IOD или Intel с Foveros/EMIB) вычислительные кристаллы соединяются через стандарты вроде **UCIe (Universal Chiplet Interconnect Express)**, создавая внутрипроцессорную NUMA-подобную структуру.

Классификация По организации многопроцессорности

- На уровне систем масштаба центра обработки данных доминируют **массово-параллельные системы (MPP)** и **кластеры**, где каждый узел — это независимый компьютер с собственным процессором, памятью и ОС, связанные высокоскоростной сетью (InfiniBand, Ethernet с RoCE).
- Такие системы масштабируются практически неограниченно и лежат в основе современных **грид- и облачных вычислений**, где ресурсы виртуализованы и распределяются динамически по запросу.
- В отличие от SMP/NUMA, в MPP используется **распределённая память**, и взаимодействие между узлами осуществляется через передачу сообщений (MPI), а не через разделяемую память.
- Однако для упрощения программирования разрабатываются гибридные подходы, такие как **DSM (Distributed Shared Memory)** — программные или аппаратные системы, эмулирующие единое адресное пространство поверх распределённой памяти.

Классификация По организации многопроцессорности

- Современная многопроцессорность всё чаще становится **гетерогенной**: в одном узле сочетаются CPU, GPU, FPGA, NPU (нейропроцессоры) и DPU (ускорители данных), каждый из которых оптимизирован под свой класс задач.
- Такие гибридные системы требуют новых моделей программирования (OpenCL, SYCL, CUDA) и эффективных межкомпонентных связей.
- Эта тенденция усиливается благодаря технологиям **дезагрегации (disaggregation),** где ресурсы памяти, хранилища и ускорителей выносятся за пределы отдельного сервера и объединяются в пулы через высокоскоростные фабрики на базе **CXL**.
- В результате архитектура многопроцессорности эволюционирует от простой SMP-модели к сложной, многоуровневой, гетерогенной и гибко конфигурируемой экосистеме, где вычислительные, памятные и сетевые ресурсы динамически распределяются в зависимости от рабочей нагрузки.

По организации многопроцессорности (1)

Класс	Описание	Тип памяти / связь	Примеры	Преимущества	Недостатки
Однопроцессорные системы	Одно CPU выполняет все задачи.	Локальная память	Ранние ПК, микроконтроллеры	Простота, низкое энергопотребление	Ограниченная производительность
SMP (Symmetric Multiprocessing)	Несколько одинаковых CPU с **общей памятью**, равноправный доступ.	Общая память (UMA), общая шина	Старые серверы (до 2000-х)	Простота программирования	Ограниченная масштабируемость, узкое место — шина
CMP (Chip Multiprocessor / многоядерные)	Несколько ядер на одном кристалле, общие L3/память.	Общая память, когерентные кэши	Современные CPU (Intel Core, AMD Ryzen)	Высокая производительность, энергоэффективность	Тепловые ограничения, interference за ресурсы
NUMA / CC-NUMA (Non-Uniform / Cache-Coherent NUMA)	Многопроцессорные системы с **неоднородным доступом** к памятии когерентностью кэшей.	Разделяемая, но неоднородная память	AMD EPYC, Intel Xeon Scalable	Масштабируемость до сотен ядер	Сложность управления локальностью (numa-aware)
MPP / Кластеры (Massively Parallel Processor)	Тысячи процессоров с **локальной памятью**, объединённых сетью.	Распределённая память (NORMA)	Суперкомпьютеры (Frontier, Fugaku)	Высокая масштабируемость, производительность	Сложное программирование (MPI), высокие задержки
Грид / Облака	Виртуализованные кластеры, ресурсы объединены сетью, доступ по требованию.	Распределённая, виртуализированная	AWS, Azure, Google Cloud, BOINC	Гибкость, масштабируемость, отказоустойчивость	Задержки, безопасность, зависимость от сети
Гетерогенные узлы (CPU + GPU + FPGA + NPU + DPU)	Разные типы вычислителей в одном узле, оптимизированные под разные задачи.	Разделяемая (UMA/hUMA) или распределённая	Серверы с GPU (NVIDIA), DPU (NVIDIA BlueField), FPGA (Intel)	Высокая эффективность для специфических задач	Сложность программирования и интеграции
Chiplet-архитектуры (CCD+IOD, Foveros, EMIB, CoWoS, SoIC)	Процессор состоит из нескольких **чиплетов**(малых кристаллов), соединённых на подложке.	Высокоскоростные межчиплетные интерконнекты	AMD EPYC (chiplets), Intel Foveros, Apple M-series	Лучший выход годных, гибкость, масштабирование	Сложность проектирования, тепловые зоны
UCIe (Universal Chiplet Interconnect Express)	Открытый стандарт соединения чиплетов (аналог PCIe для chiplets).	Электрический интерфейс между чиплетами	Intel, AMD, TSMC, Samsung (с 2023)	Интероперабельность между вендорами	Новые инструменты проектирования, тестирования
Disaggregated / Pooled Resources (CXL-fabric: memory, storage, network)	Ресурсы (память, вычисления, хранилище) **разделены** и могут выделяться по требованию через CXL.	CXL-сеть (CXL.mem, CXL.io, CXL.cache)	Современные дата-центры (Intel, Microsoft)	Гибкость, эффективное использование ресурсов	Новые протоколы, задержки, сложность управления

По организации многопроцессорности (2)

Класс	Описание	Тип памяти	Связь процессоров	Примеры	Преимущества	Недостатки
SMP (Symmetric Multiprocessing)	Несколько одинаковых процессоров, равноправнодоступных к общей памяти и устройствам.	Единая разделяемая память (UMA)	Общая шина или кроссбар	Старые десктопы/серверы с 2–4 CPU	Простота программирования, единое адресное пространство	Ограниченная масштабируемость (до ~8 ядер), конкуренция за память
NUMA (Non-Uniform Memory Access)	Многопроцессорная система с распределённой разделяемой памятью. Доступ к "локальной" памяти быстрее, чем к "удалённой".	Разделяемая, но неоднородная память	Межсокетные интерконнекты (UPI, Infinity Fabric)	Современные серверы (Intel Xeon, AMD EPYC)	Масштабируемость (до 128+ ядер), высокая пропускная способность	Неоднородные задержки, сложность оптимизации (привязка памяти/потоков)
CC-NUMA (Cache-Coherent NUMA)	NUMA скогерентностью кэшеймежду узлами.	Разделяемая, когерентная	Directory-based или snoop-based	SGI Origin, HP Superdome, современные EPYC	Поддержка единого адресного пространства при масштабируемости	Высокая сложность, накладные расходы на когерентность
UMA (Uniform Memory Access)	Все процессоры имеютодинаковый доступко всей памяти.	Разделяемая, однородная	Общая шина или кроссбар	Многоядерные CPU, старые SMP-системы	Простота, предсказуемость задержек	Ограниченная масштабируемость
MPP (Massively Parallel Processor)	Масштово-параллельная система: тысячи процессоров ссобственной локальной памятью. Нет единой памяти.	Распределённая (локальная на узел)	Высокоскоростная сеть (InfiniBand, proprietary)	Cray T3E, IBM Blue Gene, современные AI-кластеры	Высокая масштабируемость, производительность	Сложное программирование (MPI), коммуникации — узкое место
Cluster (кластер)	Связка независимых компьютеров (нод), объединённых сетью. Каждый — отдельный компьютер.	Распределённая (нет общей памяти)	Ethernet, InfiniBand, RoCE	HPC-кластеры, облачные дата-центры	Дешёвые компоненты, высокая масштабируемость	Высокие задержки межузловой связи, сложность управления
COMA (Cache-Only Memory Architecture)	Вся физическая память рассматривается как кэш; реальная память — на диске или в удалённых узлах.	Распределённая, кэш-центричная	Сеть с когерентностью	Экспериментальные системы (MIT Alewife)	Эффективное использование памяти	Сложность реализации, редко используется
NORMA (No Remote Memory Access)	Процессорыне могут напрямую обращатьсяк памяти других узлов. Обмен — только через сообщения.	Полностью распределённая	Сеть передачи сообщений	Распределённые системы, кластеры с MPI	Максимальная масштабируемость	Нет единого адресного пространства, сложность синхронизации
DSM (Distributed Shared Memory)	Программная/аппаратная абстракция, создающая иллюзию общей памяти в системе с распределённой памятью.	Виртуально разделяемая	Сеть + протоколы согласованности	Software DSM (TreadMarks), hUMA, CXL.mem	Упрощает программирование (как shared memory)	Накладные расходы, ложные разделяемые доступы

Краткие пояснения ключевых моделей

- **1. SMP (Symmetric Multiprocessing)**
  - Все процессоры равноправны, используют общую шину и память.
  - Программы могут выполняться на любом ядре.
  - Использовалась в 1990–2000-х, сейчас заменена NUMA.
- **2. NUMA**
  - Современный стандарт для многопроцессорных серверов.
  - Каждый CPU-сокет имеет локальную память.
  - Доступ к удалённой памяти — через межсокетный интерконнект (например, UPI или Infinity Fabric).
  - Требует привязки потоков и памяти (numactl, first-touch policy).
- **3. CC-NUMA**
  - NUMA с поддержкой когерентности кэшей между узлами.
  - Позволяет программам работать с единой памятью, как в SMP.
  - Реализуется через directory-based coherence (а не snooping).
- **4. MPP (Massively Parallel Processor)**
  - Тысячи процессоров, каждый со своей памятью.
  - Обмен — через передачу сообщений (MPI).
  - Пример: суперкомпьютеры, AI-тренировочные фермы.
- **5. Кластер**
  - Связка стандартных серверов (нод).
  - Каждая нода — самостоятельный компьютер.
  - Коммуникация — по сети (часто InfiniBand с RDMA).
  - Основа облачных платформ и HPC.
- **6. COMA**
  - Все локальные памяти узлов — кэши глобального адресного пространства.
  - Реальные данные могут храниться на диске или в "хозяйской" ноде.
  - Редко реализуется на практике.
- **7. NORMA**
  - Полная изоляция памяти.
  - Общение — только через очереди сообщений.
  - Используется в распределённых системах (например, микросервисы).
- **8. DSM (Distributed Shared Memory)**
  - Программист видит единую память, но она физически распределена.
  - Может быть реализовано:
    - Программно (TreadMarks)
    - Аппаратно (CXL.mem)
    - Гибридно (hUMA в AMD APU)

Краткие пояснения ключевых моделей

- **9. Грид / Облака**
  - Грид: распределённые ресурсы для научных задач (BOINC).
  - Облако: коммерческие виртуализованные кластеры (AWS, Azure).
  - Поддержка мультитенантности, автомасштабирования.
- **10. Гетерогенные вычисления**
  - Комбинация CPU, GPU (для параллелизма), FPGA (гибкость), NPU (ИИ), DPU (оффлоад сети).
  - Пример: NVIDIA Grace Hopper — CPU + GPU через NVLink-C2C.
- **11. Chiplet-архитектуры**
  - Вместо одного большого кристалла — несколько малых (chiplets).
  - Преимущества:
    - Лучший выход годных (yield).
    - Возможность комбинировать технологии (например, 5 нм для CPU, 6 нм для I/O).
  - Пример: AMD EPYC — CCD (Compute Chiplet Die) + IOD (I/O Die).
- **12. UCIe (Universal Chiplet Interconnect Express)**
  - Открытый стандарт соединения чиплетов.
  - Поддерживает coherent и non-coherent обмен.
  - Цель — создать экосистему взаимозаменяемых чиплетов от разных вендоров.
- **13. Disaggregated / Pooled Resources**
  - Разделение ресурсов: память, CPU, GPU, хранилище — в отдельных "пулах".
  - Выделяются по требованию через CXL-fabric.
  - Пример: сервер запрашивает дополнительную память с другого узла через CXL.mem.

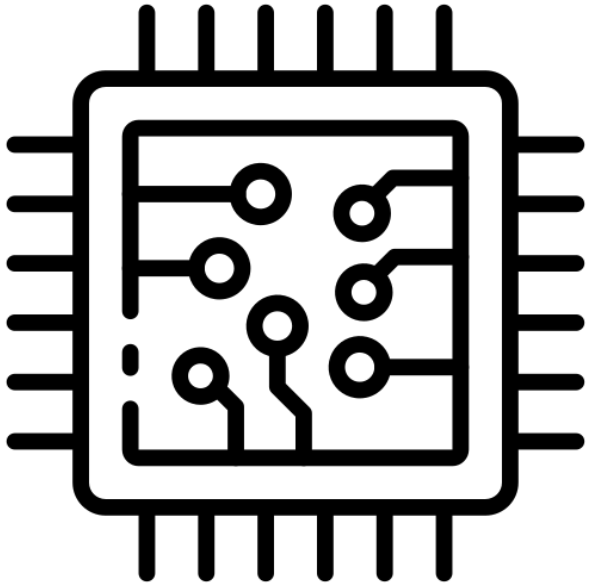
Эволюция многопроцессорности: от SMP к disaggregation

Этап	Архитектура	Ключевая идея
1980–1990	**SMP**	Общая память, несколько CPU
1990–2000	**NUMA**	Масштабирование с неоднородной памятью
2000–2010	**MPP / Кластеры**	Распределённые системы, MPI
2010–2020	**CMP / Гетерогенные узлы**	Многоядерность, GPU/FPGA
2020+	**Chiplet, CXL, Disaggregation**	Модульность, пулинг ресурсов

Модель	Лучше всего подходит для
SMP / UMA	Многоядерные CPU, десктопы
NUMA / CC-NUMA	Серверы, базы данных, виртуализация
Cluster / MPP	HPC, ИИ, big data
DSM / hUMA / CXL	Гетерогенные системы, disaggregated computing
NORMA	Высоко-масштабируемые распределённые системы

Современная многопроцессорность эволюционирует от жёстко связанных систем к гибким, модульным, распределённым архитектурам:

- **Chiplet** → модульность на уровне кристалла.
- **CXL** → когерентность и пулинг памяти/устройств.
- **Disaggregation** → ресурсы как "сервис".
- **Гетерогенность** → правильный вычислитель под задачу.



Архитектура ЭВМ. По способу адресации и доступу к памяти

Классификация По способу адресации и доступу к памяти

- Классификация архитектур ЭВМ по способу адресации и доступу к памяти охватывает как форматы инструкций, взаимодействующих с памятью, так и механизмы формирования адресов и интерпретации данных.
- Одно из фундаментальных различий — это **модель доступа к памяти: load/store, регистр-память (reg-mem)** и **память-память (mem-mem)**.
- В архитектурах **load/store** (характерных для RISC, например ARM, RISC-V) арифметические операции работают только с регистрами, а доступ к памяти осуществляется отдельными командами загрузки (LOAD) и сохранения (STORE).
- Это упрощает конвейер и повышает предсказуемость.
- В **регистр-память'ных** архитектурах (например, x86) одна инструкция может читать операнд из памяти, выполнять операцию и записывать результат обратно.
- **Память-память'ные архитектуры** (встречаются в некоторых CISC и DSP) позволяют оперировать непосредственно двумя ячейками памяти за одну команду, но они редки в современных универсальных процессорах из-за сложности реализации.

Классификация По способу адресации и доступу к памяти

- Способы формирования адресов также варьируются.
- Наиболее распространены **прямая** (адрес задан в инструкции), **косвенная** (адрес хранится в регистре или памяти), **индексная** (база + индекс), **базовая со смещением и относительная по программному счётчику** (PC-relative) адресация, последняя особенно важна для позиционно-независимого кода (PIC).
- Кроме того, архитектуры различаются по **порядку байтов** (endianness): **little-endian** (младший байт по младшему адресу — x86, RISC-V по умолчанию), **big-endian** (старший байт по младшему адресу — классический PowerPC, сетевые протоколы) и **bi-endian** (поддержка обоих режимов, как в ARM и современных RISC-V).
- Выбор endianness влияет на совместимость, сериализацию данных и производительность при работе с мультибайтовыми значениями.

Классификация По способу адресации и доступу к памяти

- Современные архитектуры всё чаще включают продвинутые механизмы управления доступом к памяти для повышения безопасности и надёжности.
- Яркий пример — **capability-based addressing**, реализованный в проекте **CHERI** (Capability Hardware Enhanced RISC Instructions).
- Вместо обычных указателей используются «капабилити» — защищённые токены, содержащие не только адрес, но и права доступа (чтение, запись, выполнение) и границы памяти.
- Это аппаратно предотвращает переполнения буферов, использование после освобождения и другие уязвимости памяти.
- Другой важный механизм — **транзакционная память**, позволяющая выполнять блоки кода как атомарные транзакции.
- **Аппаратная транзакционная память** (**HTM**), как в **Intel TSX** (с режимами RTM и HLE), использует кэш и специальные буферы для отслеживания конфликтов, а **программная транзакционная память** (STM) реализуется на уровне библиотек.
- Оба подхода упрощают параллельное программирование, заменяя блокировки на оптимистичный параллелизм.

Классификация По способу адресации и доступу к памяти

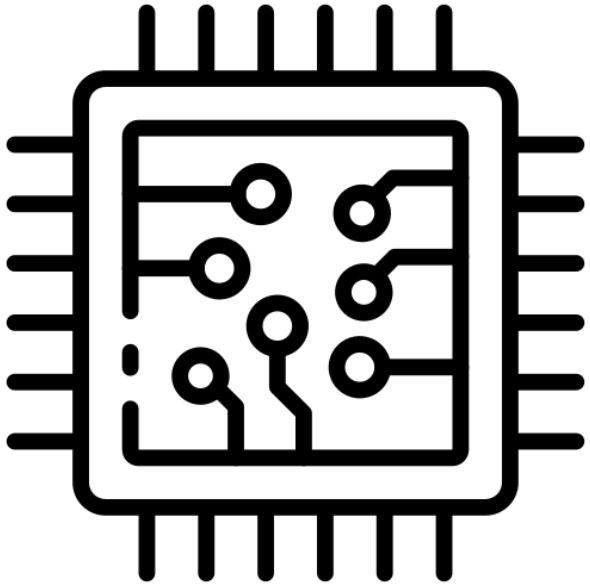
- Наконец, архитектуры различаются по **модели согласованности памяти** (memory ordering) — правилам, определяющим, в каком порядке операции с памятью становятся видимыми другим потокам.
- **x86** следует строгой модели **TSO** (Total Store Order), где все записи упорядочены, что упрощает программирование, но ограничивает оптимизации.
- **ARM** и **RISC-V** используют **слабоупорядоченную (weakly-ordered)** модель, допускающую перестановку операций для повышения производительности, но требующую явных барьеров памяти (DMB, FENCE) для корректной синхронизации.
- Эти различия критически важны при разработке многопоточных приложений и системного ПО, поскольку напрямую влияют на корректность, производительность и переносимость кода.
- Таким образом, способ адресации и доступа к памяти — это не просто техническая деталь, а глубокая архитектурная основа, определяющая безопасность, параллелизм и эффективность всей вычислительной системы.

По способу адресации и доступу к памяти

Категория	Подход / модель	Описание	Примеры	Преимущества	Недостатки
Тип доступа к памяти	**Load/Store**	Только специальные инструкцииload/storeработают с памятью. Остальные — с регистрами.	ARM, RISC-V, MIPS, Power	Упрощает конвейер, повышает ILP	Требует больше инструкций
	Register-Memory (reg-mem)	Операции могут напрямую использовать регистр и ячейку памяти.	x86 (часть инструкций)	Меньше инструкций в коде	Сложное декодирование, разная длина инструкций
	Memory-to-Memory (mem-mem)	Операции работают напрямую между ячейками памяти.	VAX, CISC (некоторые)	Очень плотный код	Очень сложное выполнение, медленно
Адресация	**Прямая**	Адрес операнда указан в инструкции.	MOV R1, [0x1000]	Простота	Жёсткая привязка к адресу
	Косвенная	Адрес хранится в регистре.	MOV R1, [R2]	Гибкость (указатели)	Дополнительная косвенность
	Индексная / база + смещение	Адрес = базовый регистр + смещение.	MOV R1, [R2 + 8]	Удобно для массивов и структур	Требует вычисления адреса
	Относительная от PC	Адрес = PC + смещение.	Ветвления, вызовы функций	Независимость от абсолютного адреса (позиционно-независимый код)	Ограниченный диапазон
Порядок байтов (Endianness)	**Little-endian**	Младший байт — по младшему адресу.	x86, x86-64, RISC-V (по умолч.)	Удобно для арифметики (младший байт первым)	Может путать при отладке
	Big-endian	Старший байт — по младшему адресу.	IBM z/Architecture, старые Power, сетевые протоколы	Соответствует записи чисел слева направо	Менее эффективно в некоторых случаях
	Bi-endian	Поддержка обоих режимов (на уровне ISA или ОС).	ARM, Power, RISC-V, MIPS	Гибкость, совместимость	Сложность реализации
Специализированные модели адресации	**Capability-based addressing**	Указатели — "возможности" (capabilities) с метаданными: адрес, длина, права.	CHERI (на базе MIPS, ARM, RISC-V)	Аппаратная защита от переполнения буфера, утечек	Требует изменений в ПО и ОС
Транзакционная память	**HTM (Hardware Transactional Memory)**	Аппаратная поддержка транзакций: блок инструкций выполняется атомарно или откатывается.	Intel TSX (HLE/RTM), IBM Blue Gene/Q	Высокая производительность в многопоточных структурах данных	Уязвимости (TSX отключён в некоторых CPU), ограниченная длина
	STM (Software Transactional Memory)	Транзакции реализованы на уровне ПО (библиотеки, компилятор).	Haskell, C++ STM библиотеки	Портабельность	Высокие накладные расходы
Модели памяти (memory ordering)	**TSO (Total Store Order)**	Почти строгая последовательность, но разрешает store-buffer forwarding.	x86/x86-64	Просто для программиста	Ограничивает оптимизации
	Weakly-ordered	Очень слабые гарантии. Порядок операций может меняться без барьеров.	ARM, RISC-V (по умолчанию)	Высокая производительность, гибкость	Сложно писать корректный параллельный код
	RC (Release Consistency)	Явное разделение на acquire/release операции.	Используется в моделях для C11/C++11, RISC-V (RVWMO)	Баланс между производительностью и понятностью	Требует явного управления барьерами

Краткие пояснения ключевых понятий

- **1. Load/Store vs Reg-Mem vs Mem-Mem**
  - **Load/Store (RISC)** — только load и store обращаются к памяти. Все операции — между регистрами.
  - **Reg-Mem (CISC)** — операции вроде ADD R1, [mem] разрешены.
  - **Mem-Mem** — ADD [mem1], [mem2] — редко, в тяжёлых CISC.
- **2. Адресация**
  - Современные архитектуры используют комбинации: база + смещение, косвенная, PC-relative.
  - Особенно важно для доступа к массивам, структурам, позиционно-независимому коду (PIE).
- **3. Endianness**
  - **Little-endian:** 0x12345678 → в памяти: 78 56 34 12
  - **Big-endian:** 12 34 56 78
  - **Bi-endian:** можно переключить режим (например, в ARM: BE8, BE32).
- **4. Capability-based addressing (CHERI)**
  - Указатель — это **capability**: (база, длина, права).
  - Защищает от:
    - Переполнения буфера
    - Использования после освобождения (use-after-free)
    - Несанкционированного доступа
  - Реализовано в **CHERI-RISC-V, CHERI-ARM.**
- **5. Транзакционная память**
  - Позволяет выполнять блок инструкций как атомарную транзакцию.
  - **HTM**: аппаратная поддержка (Intel TSX — HLE и RTM).
  - **STM**: программная реализация.
  - Применяется в lock-free структурах данных, базах данных.
- **6. Модели памяти**
  - Определяют, **в каком порядке** другие потоки видят операции с памятью.
  - **x86 (TSO)** — почти строгая, удобна, но менее гибкая.
  - **ARM/RISC-V (слабоупорядоченная)** — позволяет больше переупорядочивания → выше производительность, но требует **барьеры памяти** (dmb, fence).



Архитектура ЭВМ. По безопасности и изоляции

Классификация По безопасности и изоляции

- Классификация архитектур ЭВМ по безопасности и изоляции отражает эволюцию аппаратных механизмов, предназначенных для защиты данных, кода и системных ресурсов от вредоносных программ, утечек и атак на уровне ОС или гипервизора.
- Фундаментальной основой остаются **режимы привилегий**: в x86 используются **кольца защиты (rings 0–3)**, в ARM — **исполнительные уровни (EL0–EL3)**, а в RISC-V — **привилегированные режимы (U/M/S/H)**.
- Эти уровни разделяют код по степени доверия: ядро ОС работает в наиболее привилегированном режиме, пользовательские приложения — в наименее.
- Дополнительно в RISC-V реализованы **PMP** (Physical Memory Protection) и **PMA** (Physical Memory Attributes), позволяющие настраивать права доступа к регионам памяти даже в отсутствие MMU, что критично для встраиваемых и безопасных систем.

Классификация По безопасности и изоляции

- Для защиты от эксплуатации уязвимостей памяти (например, переполнения буфера или ROP-атак) разработаны механизмы **контроля целостности потока управления (CFI/CFP)**.
- **Intel CET (Control-flow Enforcement Technology)** включает **теневой стек (Shadow Stack)** для защиты возвратных адресов и **IBT (Indirect Branch Tracking)** для проверки целевых адресов косвенных переходов.
- Аналогичные функции в ARM — **PAC (Pointer Authentication)** и **BTI (Branch Target Identification)** — используют криптографические теги и метки для верификации указателей и целей ветвлений.
- Для повышения безопасности памяти применяются **тегированные архитектуры: ARM MTE (Memory Tagging Extension)** присваивает каждому блоку памяти 4-битный тег, проверяемый при доступе, что позволяет обнаруживать use-after-free и buffer overflows в реальном времени.
- Подобные подходы реализованы в **HWASAN** и ранее — в устаревшем **Intel MPX**.

Классификация По безопасности и изоляции

- Следующий уровень изоляции обеспечивается через **доверенные исполняемые среды (TEE — Trusted Execution Environments)** и **защищённые анклавы**.
- **Intel SGX** создаёт изолированные «энклавы» в памяти, защищённые даже от гипервизора и ядра ОС.
- **ARM TrustZone** разделяет систему на «нормальный» и «безопасный» миры, где критические операции (аутентификация, криптография) выполняются в изолированной среде.
- В экосистеме RISC-V развивается **Keystone**, а AMD и ARM предлагают собственные решения — **SEV/SEV-SNP** и **CCA/Realm**, соответственно.
- Эти технологии легли в основу парадигмы **Confidential Computing**, где данные и код защищаются не только при хранении и передаче, но и **во время обработки**.
- **Intel TDX, AMD SEV-SNP и ARM CCA** обеспечивают аппаратную изоляцию виртуальных машин, гарантируя, что даже облачный провайдер не может получить доступ к содержимому ВМ.

Классификация По безопасности и изоляции

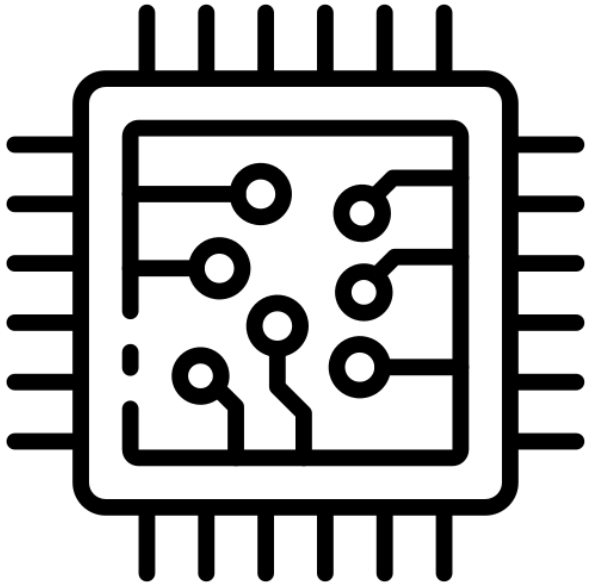
- Наконец, безопасность расширяется на периферию и ввод-вывод. **IOMMU (Intel VT-d, AMD-Vi)** изолирует устройства, предотвращая DMA-атаки, и поддерживает **DMA fencing**, **ATS (Address Translation Services)** и **PRI (Page Request Interface)** для безопасного и эффективного доступа к памяти из устройств.
- Криптографические ключи и чувствительные данные хранятся в **аппаратных защищённых хранилищах**, таких как **TPM**, **Secure Enclave** (Apple) или **TrustZone CryptoCell**.
- Все эти механизмы формируют многоуровневую, «глубокую» защиту, где изоляция обеспечивается на каждом уровне — от указателей и потоков управления до виртуальных машин и физических устройств.
- Таким образом, современная архитектура безопасности — это не отдельная функция, а сквозная, аппаратно-поддерживаемая стратегия, интегрированная в саму основу вычислительной платформы.

По безопасности и изоляции

Категория	Технология / механизм	Описание	Примеры	Преимущества	Недостатки
Режимы привилегий	**Кольца (rings) / Режимы выполнения (ELs)**	Иерархия привилегий: ядро (высокий уровень), пользователь (низкий).	x86 (rings 0–3), ARM (EL0–EL3), RISC-V (U/S/M/H)	Изоляция ОС и приложений	Большинство ОС использует только два уровня
	PMP / PMA (RISC-V) (Physical Memory Protection / Physical Memory Attributes)	Аппаратный контроль доступа к физической памяти.	RISC-V (PMP)	Гибкая настройка прав доступа	Сложность конфигурации
Виртуализация	**VT-x (Intel), AMD-V, ARM VHE, RISC-V H-extension**	Аппаратная поддержка виртуальных машин (VM).	Intel Core, AMD EPYC, ARM Neoverse	Высокая производительность виртуализации	Сложность управления, уязвимости (например, Spectre)
	VT-d / AMD-Vi / ARM SMMU	IOMMU для защиты DMA и виртуализации устройств.	Intel VT-d, AMD-Vi, ARM SMMU	Защита от вредоносных устройств (DMA-атаки)	Требует поддержки BIOS/UEFI
TEE (Trusted Execution Environment)	**Intel SGX (Software Guard Extensions)**	Защищённые "энклавы" — изолированные области памяти с шифрованием.	Intel Skylake+ (ограничено)	Высокая конфиденциальность данных даже от ОС	Уязвимости, ограниченный размер, сложность разработки
	AMD SEV / SEV-ES / SEV-SNP	Шифрование памяти виртуальной машины на уровне CPU.	AMD EPYC (Rome и новее)	Конфиденциальность VM в облаке	Требует поддержки гипервизора
	ARM TrustZone	Делит систему наSecure WorlduNormal World.	ARM Cortex-A, IoT-чипы	Поддержка безопасного ПО (keystore, DRM)	Ограниченная изоляция (все в одном CPU)
	ARM CCA / Realms	Новое поколение TEE: аппаратная изоляция "реальмов" с доверенной базой.	ARMv9+, Confidential Computing	Масштабируемость, совместимость с облаком	Новые системы, ограниченная экосистема
	Keystone (RISC-V)	Открытая TEE-платформа на базе RISC-V.	RISC-V с PMP/PCC	Открытый исходный код, безопасность по дизайну	Экспериментальная, не промышленная зрелость
Confidential Computing	**Intel TDX (Trust Domain Extensions)**	Аппаратная изоляция виртуальных машин с шифрованием памяти.	Intel Sapphire Rapids	Конфиденциальность в облаке, защита от администратора	Требует поддержки гипервизора и ОС
	AMD SEV-SNP	Расширение SEV с защитой от вредоносных гипервизоров.	AMD EPYC Genoa	Защита целостности кода и данных	Сложная настройка
	NVIDIA Confidential GPU	Расширение конфиденциальных вычислений на GPU.	NVIDIA H100 (с BlueField DPU)	Обработка конфиденциальных данных на GPU	Новая технология, требует CXL и DPU
Контроль потока выполнения (CFI/CFP)	**Intel CET (Control-flow Enforcement Technology) – Shadow Stack – IBT (Indirect Branch Tracking)**	Защита от ROP/JOP-атак. Shadow Stack — отдельный стек вызовов.	Intel Tiger Lake+	Аппаратная защита стека и переходов	Ограниченная поддержка ПО
	ARM PAC (Pointer Authentication), BTI (Branch Target Identification)	Криптографическая подпись указателей и защита целей переходов.	ARMv8.3+ (PAC), ARMv8.5+ (BTI)	Защита от переполнения стека, ROP	Требует компиляторной поддержки (LLVM, GCC)
Memory Safety	**ARM MTE (Memory Tagging Extension)**	Добавляет 4-битные теги к указателям и памяти. Обнаруживает утечки/переполнения.	ARMv8.5+ (MTE), Google Pixel 6+	Обнаружение ошибок в реальном времени	Накладные расходы, частичная поддержка
	Tagged memory / HWASAN	Аппаратная поддержка тегированной памяти (аналог AddressSanitizer).	MTE, Tagged Pointers (Apple)	Высокая эффективность детектирования	Требует ОС и ПО
	Intel MPX (устар.)	Расширенные указатели с границами.	Intel Skylake (отключено)	Попытка аппаратной безопасности	Низкая производительность, не поддерживается
Защита I/O	**IOMMU + ATS/PRI**	IOMMU изолирует DMA. ATS — кэширование TLB устройствами. PRI — запросы на ошибки страниц.	VT-d, AMD-Vi, PCIe ATS/PRI	Защита от DMA-атак, виртуализация устройств	Требует поддержки устройств
	DMA-fencing	Ограничение доступа устройств к памяти через IOMMU.	Включено по умолчанию в современных системах	Критически важна для безопасности	Может снижать производительность
Безопасные хранилища	**Secure Key Storage / Root of Trust**	Аппаратные модули для хранения ключей (eFuse, TPM, Secure Enclave).	Apple Secure Enclave, TPM 2.0, ARM CryptoCell	Защита ключей даже при компрометации ОС	Физические атаки возможны

Краткие пояснения ключевых понятий

- **1. Режимы привилегий**
  - **x86**: 4 кольца (0–3), ОС — ring 0, приложения — ring 3.
  - **ARM**: Execution Levels (EL0–EL3): EL0 — приложения, EL1 — ОС, EL2 — гипервизор, EL3 — монитор.
  - **RISC-V**: Machine (M), Supervisor (S), User (U), Hypervisor (H).
- **2. TEE (Trusted Execution Environment)**
  - **SGX**: изолированные энклавы в памяти (Intel).
  - **TrustZone**: разделение на Secure/Normal world (ARM).
  - **SEV/SNP**: шифрование всей VM (AMD).
  - **CCA/Realms**: следующее поколение TEE (ARMv9).
- **3. Confidential Computing**
  - Цель: обработка данных в зашифрованном виде, даже при доступе со стороны ОС, гипервизора или администратора.
  - Реализуется через аппаратное шифрование памяти и изоляцию доменов (TDX, SEV-SNP, CCA).
- **4. CFI / CFP (Control-Flow Integrity / Protection)**
  - Защита от атак типа ROP (Return-Oriented Programming).
  - **Intel CET**: Shadow Stack + IBT.
  - **ARM PAC/BTI**: криптографические подписи указателей и целей переходов.
- **5. Memory Safety**
  - **ARM MTE**: обнаруживает use-after-free, buffer overflow в реальном времени.
  - **HWASAN**: аппаратная версия AddressSanitizer.
  - **MPX** — устаревшая, неэффективная технология.
- **6. IOMMU и защита I/O**
  - Без IOMMU устройство (например, USB-контроллер) может читать всю память.
  - IOMMU ограничивает DMA только разрешёнными областями.
  - ATS/PRI — улучшают производительность и надёжность в виртуализированных средах.



Архитектура ЭВМ. По устойчивости и надежности

Классификация По устойчивости и надежности

- Классификация архитектур ЭВМ по устойчивости и надёжности охватывает аппаратные и программные методы, направленные на предотвращение, обнаружение и восстановление после ошибок — как случайных (например, вызванных космическими лучами), так и системных.
- Ключевым элементом в серверных и критически важных системах является **коррекция ошибок в памяти (ECC — Error Correcting Code)**.
- Наиболее распространённый стандарт — **SECDED (Single Error Correction, Double Error Detection)** — позволяет исправлять однобитовые и обнаруживать двухбитовые ошибки.
- Более продвинутые схемы, такие как **DEC-TED** (Double Error Correction, Triple Error Detection) или **Chipkill**, способны выдерживать отказ целого чипа памяти в модуле.
- Современные технологии, включая **DDR5**, внедряют **on-die ECC** прямо в микросхемы DRAM, а **патрулирующая очистка (patrol scrubbing)** регулярно сканирует память в фоне, исправляя ошибки до их накопления и превращения в неисправимые сбои.

Классификация По устойчивости и надежности

- Для защиты вычислительных блоков применяются методы **избыточности**.
- **Lockstep-архитектура** использует два (или более) идентичных ядра, выполняющих одну и ту же программу; их результаты постоянно сравниваются, и при расхождении система переходит в безопасное состояние.
- **TMR (Triple Modular Redundancy)** — тройное резервирование, при котором три копии вычисляют один результат, а голосованием выбирается правильный (даже при сбое одного блока).
- В менее критичных, но всё же важных системах применяется **горячее резервирование** (hot spare) и **механизмы контрольных точек** (checkpointing)
  с откатом (rollback): состояние системы периодически сохраняется, и при обнаружении ошибки выполнение возобновляется с последней корректной точки.
- Такие подходы широко используются в высоконадёжных серверах, телекоммуникационном оборудовании и распределённых системах.

Классификация По устойчивости и надежности

- В условиях повышенного радиационного фона — космос, авиация, ядерные установки — применяются **радиационно-стойкие (rad-hard) архитектуры**.
- К ним относятся специализированные процессоры, такие как **RAD750** и **RAD5500** на базе PowerPC, LEON/GR740 (SPARC), а также современные **rad-hard RISC-V** ядра.
- Они используют технологические (утолщённые оксиды, SOI-подложки) и архитектурные (дублирование триггеров, ECC в регистрах) меры для защиты от **одиночных событий (SEU — Single Event Upset)** и **защёлкивания (SEL — Single Event Latchup)**.
- Такие чипы проходят строгую сертификацию и работают в экстремальных условиях, где обычные процессоры быстро выходят из строя.

Классификация По устойчивости и надежности

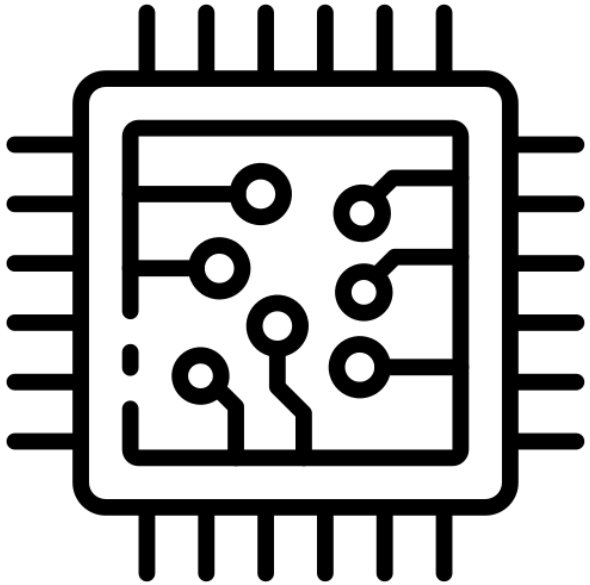
- Наконец, надёжность регулируется отраслевыми **стандартами функциональной безопасности.**
- В автомобильной промышленности — ISO 26262 с уровнями ASIL (A–D), в авионике — DO-178C (для ПО) и DO-254 (для аппаратуры), в медицинских устройствах — IEC 62304.
- Эти стандарты требуют не только аппаратной устойчивости, но и строгого процесса разработки, верификации и документирования. В соответствии с ними проектируются **fail-safe** (переход в безопасное состояние при сбое) и **fail-operational** (продолжение работы даже при частичном отказе) архитектуры.
- Например, автопилот в современном самолёте или система управления тормозами в автомобиле должны оставаться работоспособными даже при отказе одного из компонентов.
- Таким образом, устойчивость и надёжность в современных ЭВМ — это не просто «крепкий чип», а комплексная дисциплина, объединяющая аппаратный дизайн, системное ПО, процессы разработки и строгие нормативные требования.

По устойчивости и надёжности

Категория	Механизм / технология	Описание	Примеры	Преимущества	Недостатки
Коррекция ошибок в памяти	**ECC (Error Correcting Code) – SECDED (Single Error Correct, Double Error Detect) – DEC-TED (Double Error Correct, Triple Error Detect) – Chipkill (многочиповая коррекция) – On-die ECC (внутри чипа, DDR5)**	Обнаружение и исправление ошибок в ОЗУ.	Серверы, HPC, рабочие станции	Защита от одиночных и множественных битовых ошибок	Дополнительные затраты по площади и энергопотреблению
	Patrol Scrubbing	Периодическое чтение и корректирование памяти для предотвращения накопления ошибок.	IBM zSeries, Intel Xeon, AMD EPYC	Предотвращает "застарелые" битовые сбои	Небольшая нагрузка на память
Избыточность вычислений	**Lockstep (двойной/тройной режим)**	Два или три ядра выполняют одну программу синхронно; при расхождении — сбой.	Автомобильные MCU, авиационные CPU	Высокая надёжность	Удвоение/утроение аппаратных затрат
	TMR (Triple Modular Redundancy)	Три модуля выполняют одну задачу, результат выбирается по мажоритарному принципу.	Космические системы, ядерные установки	Устойчивость к одиночным сбоям	Высокая избыточность (3× аппаратура)
	Горячее резервирование (Hot Standby)	Резервный узел работает параллельно или готов к немедленному включению.	Телеком, медицинские системы	Минимальное время восстановления	Высокое энергопотребление
	Checkpoint/rollback (контрольные точки)	Периодическое сохранение состояния; при сбое — откат к последней контрольной точке.	HPC, кластеры, embedded-системы	Программная устойчивость без избыточного железа	Накладные расходы, потеря прогресса
Радиационная стойкость (rad-hard)	**RAD-серии процессоров (радиационно-стойкие)**	Устойчивы к космическому излучению (SEU, latch-up).	RAD750 (PowerPC), LEON3/GR740 (SPARC), rad-hard RISC-V	Работа в космосе, на орбите, в ускорителях	Низкая производительность, высокая стоимость
	Технологии защиты	Использование SOI, epitaxial substrates, ECC, TMR на кристалле.	BAE Systems, Cobham Gaisler, NASA	Устойчивость к одиночным событиям (SEU)	Сложное производство
Стандарты для критических систем	**ISO 26262 (ASIL) (автомобильная)**	Классификация уровней функциональной безопасности (ASIL A–D).	Автомобильные ECU, ADAS	Обеспечивает безопасность в автономных системах	Требует сложной верификации
	DO-178C / DO-254 (авиационные)	Стандарты разработки ПО и аппаратуры для авиации.	Самолёты (Boeing, Airbus)	Высокая достоверность, сертифицируемость	Длительный цикл разработки
	IEC 62304 (медицинские)	Стандарт разработки ПО для медицинских устройств.	Имплантируемые устройства, сканеры	Безопасность для жизни	Жёсткие требования к тестированию
Архитектуры отказоустойчивости	**Fail-safe (отказо-безопасные)**	При сбое система переходит в безопасное состояние (остановка).	Тормоза в поездах, лифты	Защита от опасных последствий	Полная остановка функции
	Fail-operational (отказоустойчивые)	При сбое система продолжает работать (например, дублированные каналы).	Автопилоты, автономные автомобили	Высокая доступность	Сложность и стоимость

Краткие пояснения ключевых понятий

- **1. ECC и коррекция ошибок**
  - **SECDED:** исправляет 1 бит, обнаруживает 2.
  - **DEC-TED:** исправляет 2 бита, обнаруживает 3.
  - **Chipkill:** может исправить целый DRAM-чип (до 4 бит).
  - **On-die ECC (DDR5):** встроена в модуль — защищает данные внутри чипа.
  - **Scrubbing:** фоновое сканирование памяти для исправления ошибок до их накопления.
- **2. Избыточность**
  - **TMR** — классический подход в космосе: 3 вычислителя, 1 голосующее устройство.
  - **Lockstep** — два ядра выполняют одинаковые инструкции синхронно; при расхождении — ошибка.
  - **Checkpoint/rollback** — используется в длительных вычислениях (например, симуляции).
- **3. Радиационная стойкость**
  - В космосе и на высоте — космические лучи вызывают SEU (Single Event Upset) — переворот бита.
  - RAD-процессоры используют:
    - Утолщённые диэлектрики
    - SOI (Silicon-on-Insulator)
    - Встроенную ECC и TMR
  - Пример: RAD750 — на Марсианских роверах.
- **4. Стандарты безопасности**
  - **ASIL** (Automotive Safety Integrity Level): от A (низкий) до D (высокий). ASIL-D — например, тормозная система.
  - **DO-178C** — для ПО в авиации (уровни A–E, A — критичный).
  - **IEC 62304** — классификация ПО по риску (Class A–C).
- **5. Fail-safe vs Fail-operational**
  - **Fail-safe:** машина останавливается безопасно (например, поезд тормозит).
  - **Fail-operational:** система продолжает работать (например, автопилот переключается на резервный канал).



Архитектура ЭВМ. Исторические и учебные архитектуры

Классификация. Исторические и учебные архитектуры

- Классификация архитектур ЭВМ на исторические и учебные охватывает как первые реализации вычислительных машин, заложившие основы современной информатики, так и специально разработанные модели для обучения и исследований.
- Среди пионеров — **IAS-машина, EDVAC и EDSAC**, созданные в 1940–1950-х годах и воплотившие принципы **архитектуры фон Неймана**: единое адресное пространство для кода и данных, программируемость и последовательное выполнение инструкций.
- Эти машины стали прототипами всех универсальных компьютеров.
- В 1960-х IBM представила революционную **System/360** — первое семейство совместимых ЭВМ, введшее понятие единой архитектуры для разных моделей, что позволило клиентам масштабировать системы без переписывания ПО.
- Эта концепция до сих пор лежит в основе современных процессорных платформ.

Классификация. Исторические и учебные архитектуры

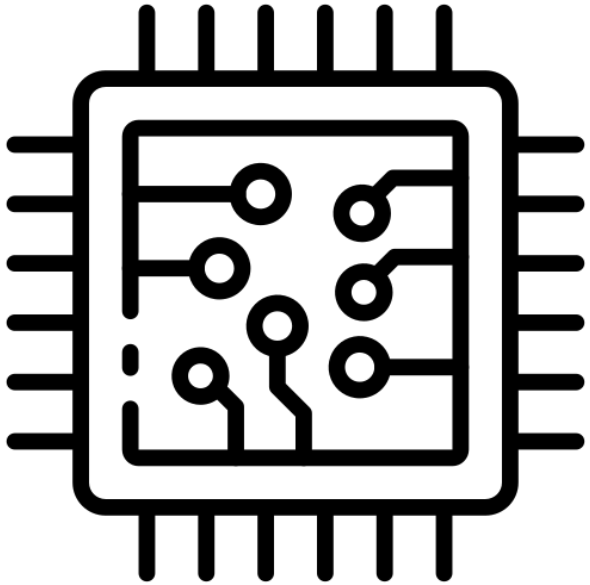
- В 1970–1980-х годах доминировали мини- и микрокомпьютеры, ставшие доступными для университетов и предприятий.
- **DEC PDP-8 и PDP-11** (а позже **VAX**) оказали огромное влияние на развитие операционных систем (например, Unix) и языков программирования (C).
- В СССР развивались собственные линейки: **МЭСМ** и **БЭСМ** — одни из первых советских ЭВМ, а **ЕС ЭВМ** (Единая система) была клоном IBM System/360.
- Особое место занимает отечественная архитектура «**Эльбрус**», начавшаяся с суперкомпьютеров на базе стековой архитектуры, а затем перешедшая к **VLIW** (очень длинному машинному слову) — подходу, опередившему своё время и предвосхитившему идеи Itanium.
- Эти системы демонстрируют, как исторические архитектуры не только решали практические задачи, но и исследовали альтернативные пути развития вычислений.

Классификация. Исторические и учебные архитектуры

- Эпоха персональных компьютеров была определена массовыми 8- и 16-битными микропроцессорами: **MOS 6502** (Apple II, Commodore 64), **Zilog Z80** (ZX Spectrum, CP/M-машины) и **Motorola 68000** (Macintosh, Amiga).
- Эти чипы отличались простотой, доступностью и ярко выраженной CISC-архитектурой, что сделало их идеальными как для коммерческого использования, так и для обучения.
- Их изучение до сих пор помогает понять основы работы процессора, адресации и взаимодействия с периферией.
- В то же время появились и чисто **учебные архитектуры**, такие как **DLX** (разработанная Джоном Хеннесси и Дэвидом Паттерсоном) и **LC-3**, созданные специально для преподавания основ компьютерной архитектуры без излишней сложности реальных ISA.

Классификация. Исторические и учебные архитектуры

- Современный этап характеризуется возрождением открытых и исследовательских платформ.
- **RISC-V** — не только промышленный стандарт, но и мощная учебная база: его модульность позволяет создавать как ультрапростые ядра (**PicoRV32**), так и высокопроизводительные (**BOOM, Rocket**).
- **Проекты вроде OpenTitan** (открытый корень доверия) и симуляторы (**gem5, QEMU, Spike, SimH**) позволяют не только изучать, но и экспериментировать с архитектурами в виртуальной среде.
- Кроме того, **виртуальные машины — JVM, .NET CLR, WebAssembly** — сами по себе являются «архитектурами» с собственными ISA, изолированными средами выполнения и моделями памяти, и широко используются в обучении компиляторам, безопасности и кроссплатформенной разработке.
- Таким образом, исторические и учебные архитектуры служат не только памятниками прошлому, но и живыми инструментами для понимания будущего вычислительных систем.



Архитектура ЭВМ. По специализированным областям применения

Классификация По специализированным областям применения

- Классификация архитектур ЭВМ по специализированным областям применения отражает переход от универсальных процессоров к вычислительным системам, оптимизированным под конкретные задачи, рабочие нагрузки и ограничения среды.
- В сфере **высокопроизводительных вычислений (HPC)** доминируют гибридные платформы: серверные CPU с векторными расширениями (**AVX-512** в x86, **SVE/SVE2** в ARM), соединённые с массивами **GPU** (NVIDIA Hopper/Blackwell, AMD CDNA) через высокоскоростные интерконнекты (**NVLink, InfiniBand**).
- Для максимальной пропускной способности используются 3D-стекированная память (HBM) и специализированные ускорители, такие как **Google TPU**.
- Такие системы проектируются для максимизации вычислительной плотности и пропускной способности памяти, часто в ущерб энергоэффективности или детерминизму.

Классификация По специализированным областям применения

- В области **искусственного интеллекта и машинного обучения (AI/ML)** архитектуры становятся всё более гетерогенными и специализированными.
- Помимо GPU, активно развиваются **нейропроцессоры (NPU)**: Apple Neural Engine, ARM Ethos-U, Qualcomm Hexagon Tensor Processor (HTP) — для edge-устройств, и **крупномасштабные ускорители**: NVIDIA Blackwell, AMD MI300X, Intel Gaudi и Google TPU v5 — для дата-центров.
- Эти чипы оптимизированы под операции с низкой точностью (INT4/INT8/FP8), матричные умножения и плотные потоки данных, часто интегрируя память ближе к вычислительным блокам (in-memory computing) для преодоления «узкого горлышка фон Неймана». Программные стеки (CUDA, ROCm, OpenVINO) и компиляторы (TVM, MLIR) играют ключевую роль в эффективном использовании этих архитектур.

Классификация По специализированным областям применения

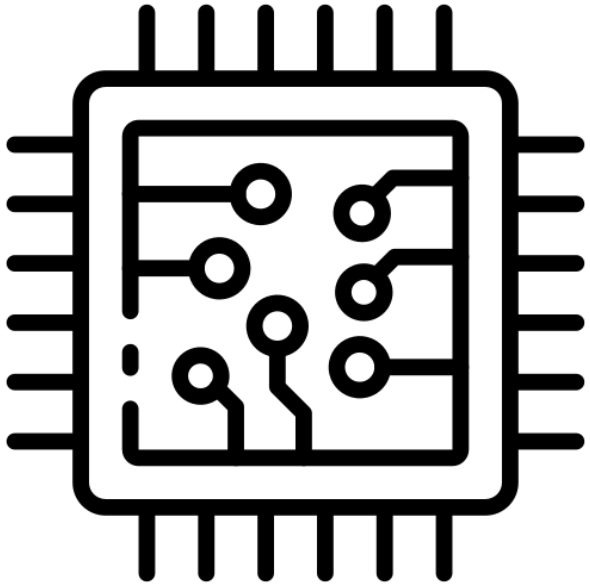
- **Для встраиваемых систем, edge-устройств и задач реального времени (RT)** критичны **энергоэффективность**, **предсказуемость и детерминизм**.
- Здесь доминируют микроконтроллеры и процессоры с упрощённой архитектурой: **ARM Cortex-M/R/A** (в режимах low-power), **RISC-V RV32/64** с расширениями для DSP или векторных операций, а также классические **DSP** (например, TI C6000) с **гарвардской архитектурой**, **MAC-блоками** и поддержкой **фиксированной точки**.
- Вместо сложных кэшей часто используются scratchpad SRAM — статическая память с гарантированным временем доступа.
- Архитектуры реального времени могут применять **TTA (Time-Triggered Architecture)** или **lockstep-режим** для обеспечения строгих временных гарантий, что необходимо в автомобильной электронике, промышленной автоматике и авионике.

Классификация По специализированным областям применения

- Наконец, в **инфраструктурных доменах — сетях, хранилищах и телекоммуникациях** — всё большую роль играют **специализированные процессоры данных: DPU (Data Processing Unit) и SmartNIC**.
- Примеры — NVIDIA BlueField, AMD Pensando, Intel IPU.
- Они разгружают CPU от задач виртуализации, шифрования, обработки сетевых пакетов и управления хранилищем, обеспечивая высокую пропускную способность и изоляцию.
- **В 5G и телекоммуникациях** применяются **FPGA, ACAP** (Adaptive Compute Acceleration Platform, как Xilinx Versal) и **inline-ускорители**, способные обрабатывать трафик на лету с минимальными задержками.
- Таким образом, современная архитектура ЭВМ всё чаще представляет собой не единый процессор, а **целевую вычислительную платформу**, где CPU, ускорители, память и I/O тесно интегрированы и оптимизированы под конкретную предметную область — от облака до датчика в поле.

По специализированным областям применения

Область применения	Ключевые требования	Архитектурные особенности	Примеры / технологии	Преимущества	Особенности
DSP (Digital Signal Processing)	Обработка сигналов в реальном времени, высокая вычислительная плотность.	**Гарвардская архитектура, MAC-блоки (умножение-накопление), фиксированная точка, VLIW.**	TI TMS320C6x, Analog Devices SHARC, ARM Cortex-M с DSP-расширениями	Высокая эффективность при обработке аудио, видео, радио.	Оптимизированы под повторяющиеся вычисления (фильтры, FFT).
Системы реального времени (RT)	Детерминизм, предсказуемые задержки, отсутствие джиттера.	**Scratchpad SRAM (вместо кэша), предсказуемые шины, без сложного OoO.**	ARM Cortex-R, Infineon AURIX, NXP S32, TTArch (Time-Triggered)	Гарантированное время отклика.	Кэши заменяются на управляемую память (SPRAM).
Встраиваемые системы (Embedded)	Низкое энергопотребление, малый размер, надёжность.	**Простые конвейеры, энергосберегающие режимы, предсказуемые тайминги.**	ARM Cortex-M, RISC-V RV32, AVR, PIC	Долгая автономная работа, низкая стоимость.	Часто используются без ОС или с RTOS.
HPC (High Performance Computing)	Максимальная производительность, масштабируемость, пропускная способность памяти.	**Векторные ISA (AVX-512, SVE), GPU, HBM, NVLink, InfiniBand.**	Intel Xeon + AVX-512, AMD EPYC + MI200, NVIDIA A100/H100, Cray, Frontier (суперкомпьютер)	Высокая FLOPS, эффективность на задачах моделирования.	Используются кластеры с тысячами узлов.
AI/ML (искусственный интеллект)	Ускорение матричных операций (GEMM), энергоэффективность.	**Специализированные ускорители: TPU, NPU, матричные блоки (Tensor Cores, AMX).**	Google TPU, NVIDIA Hopper/Blackwell, AMD MI300X, Intel Gaudi, Apple Neural Engine, ARM Ethos-U	Высокая производительность в ИИ при низком энергопотреблении.	Часто используются систолические массивы.
Edge / IoT / Embedded AI	Низкое энергопотребление, малый форм-фактор, локальная обработка.	**Микроконтроллеры, microNPU, DSP, RISC-V.**	ARM Cortex-M55 + Ethos-U55, Apple Watch, ESP32, microNPU (Glow, Arm)	Обработка данных на устройстве (без облака).	Поддержка TinyML, TensorFlow Lite.
Сетевые и хранилищные ускорители (SmartNIC / DPU / IPU)	Оффлоад сетевых и хранилищных функций с CPU.	**Встроенные процессоры, I/O-ускорители, поддержка виртуализации (SR-IOV).**	NVIDIA BlueField DPU, AMD Pensando, Intel IPU (Infrastructure Processing Unit)	Освобождает CPU, повышает безопасность и эффективность.	Используется в облаках и дата-центрах.
Телеком / 5G / RAN	Обработка потоковых данных, низкие задержки, гибкость.	**FPGA, ACAP (Adaptable Compute Acceleration Platform), DSP-ускорители.**	Xilinx Versal (ACAP), Intel Agilex, 5G базовые станции	Быстрая перенастройка под протоколы (5G NR, Open RAN).	Inline-ускорение пакетной обработки.
Safety-critical (безопасность-критичные)	Высокая надёжность, отказоустойчивость, соответствие стандартам.	**Lockstep-ядра, TTA (Time-Triggered Architecture), ECC, контрольные точки.**	Автомобильные (ASIL-D), авиационные (DO-178C), медицинские (IEC 62304).	Гарантированная безопасность при сбоях.	Используется в автопилотах, тормозных системах.



Архитектура ЭВМ. Межкристальные и упаковочные технологии

Межкристальные и упаковочные технологии

- Современные вычислительные системы всё больше полагаются не только на микроархитектуру ядра, но и на **физическое расположение и соединение кристаллов** (чиплетов).
- Упаковочные и межкристальные технологии определяют **пропускную способность, задержки, энергоэффективность и масштабируемость чипов**.

- **Основные цели упаковочных технологий**
  - Увеличение плотности интеграции.
  - Снижение задержек между блоками.
  - Повышение пропускной способности (особенно между CPU, GPU, памятью).
  - Гибкость проектирования (chiplet-подход).
  - Улучшение тепловыделения и выхода годных (yield).

Межкристальные и упаковочные технологии

- Классификация архитектур ЭВМ по межкристальным и упаковочным технологиям отражает фундаментальный сдвиг в проектировании процессоров: от единых монолитных кристаллов к модульным, гетерогенным системам на основе множества соединённых чиплетов.
- **Монолитные кристаллы** — традиционный подход, при котором все компоненты процессора (ядра, кэш, контроллеры памяти, I/O) изготавливаются на одном кремниевом кристалле.
- Такая архитектура обеспечивает минимальные задержки и высокую плотность соединений, но сталкивается с растущими проблемами: снижением выхода годных изделий при увеличении площади кристалла, ростом стоимости и физическими ограничениями масштабирования.
- В эпоху перехода к суб-5-нм техпроцессам монолитный дизайн становится всё менее экономически и технологически целесообразным.

Межкристальные и упаковочные технологии

- В ответ на эти вызовы возникла парадигма **чиплетных систем (chiplet-based design)**, при которой процессор разбивается на функциональные блоки — чиплеты, изготавливаемые независимо и соединяемые в одном корпусе.
- Яркий пример — архитектура **AMD Zen**, где вычислительные кристаллы (**CCD**) с ядрами и кэшем соединяются с центральным I/O-кристаллом (**IOD**) через высокоскоростную шину **Infinity Fabric**.
- Intel использует собственные технологии упаковки: **EMIB (Embedded Multi-die Interconnect Bridge)** для 2D-соединений и **Foveros** — для 3D-стекирования чиплетов. TSMC предлагает решения **CoWoS (Chip-on-Wafer-on-Substrate)** и **SoIC (System on Integrated Chips)**, позволяющие создавать сложные 2.5D и 3D-сборки с высокой плотностью межсоединений.
- Такой подход снижает стоимость, повышает гибкость (можно комбинировать чиплеты из разных техпроцессов) и улучшает масштабируемость.

Межкристальные и упаковочные технологии

- Ключевым элементом чиплетных архитектур являются **стандартизированные межкристальные интерфейсы**, обеспечивающие совместимость и высокую пропускную способность между разнородными блоками.
- Лидером в этой области стал **UCIe (Universal Chiplet Interconnect Express)** — открытый стандарт, основанный на физическом уровне PCIe и поддерживающий как **дешёвые 2D-соединения**, так и **высокопроизводительные 3D-стеки**.
- UCIe обеспечивает когерентность кэшей, низкие задержки и высокую энергоэффективность (до 0.5 пДж/бит), что делает его основой для будущих гетерогенных SoC.
- Другие инициативы включают **BoW (Bunch of Wires)** от Intel — упрощённый интерфейс для коротких соединений, и **OpenHBI (Open High Bandwidth Interconnect)** — открытая альтернатива для высокоскоростных межсоединений, ориентированная на HPC и ИИ.

Межкристальные и упаковочные технологии

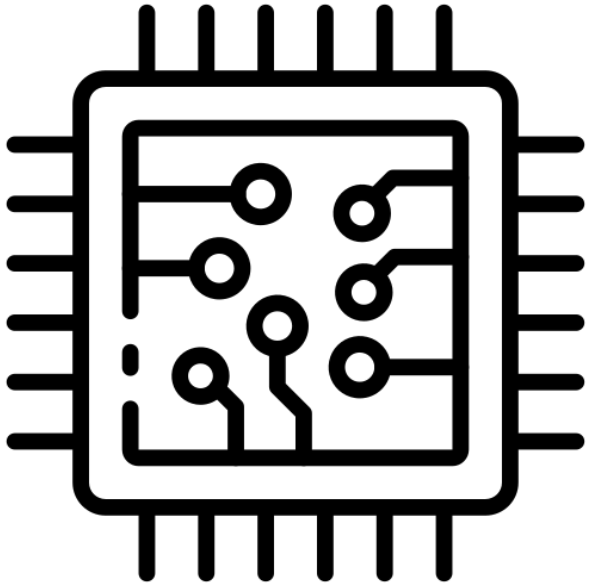
- Таким образом, межкристальные и упаковочные технологии перестали быть просто способом сборки чипа — они стали **архитектурным уровнем проектирования**.
- Современный процессор всё чаще представляет собой «систему в корпусе» (SiP), где CPU, GPU, NPU, HBM и I/O-контроллеры — это независимые чиплеты, соединённые через стандартизированные, высокоскоростные фабрики.
- Это позволяет создавать гибкие, масштабируемые и экономически эффективные платформы, адаптированные под конкретные рабочие нагрузки — от мобильных устройств до суперкомпьютеров.
- Будущее архитектуры ЭВМ лежит не в увеличении размера кристалла, а в интеллектуальной интеграции множества специализированных кристаллов в единую, согласованную вычислительную систему.

Классификация межкристальных и упаковочных технологий

Технология	Описание	Особенности	Примеры	Преимущества	Недостатки
Monolithic (монолитный кристалл)	Все компоненты (CPU, GPU, кэш, I/O) — на одном кремниевом кристалле.	Традиционный подход.	Intel Core (до 12-го поколения), Apple A14	Простота, низкие задержки	Ограниченный размер, низкий выход годных при больших диезах
2.5D Packaging (2.5D-упаковка)	Несколько кристаллов (чиплетов) размещаются наинтерпосере(passive silicon interposer), соединённом с подложкой.	Интерпосер обеспечивает плотные межсоединения (TSV — Through-Silicon Vias).	NVIDIA GPU (HBM), AMD Instinct, Intel Foveros (частично)	Высокая пропускная способность, поддержка HBM	Дорогая технология, сложное производство
3D Stacking (3D-наращивание)	Кристаллы соединяютсявертикальночере з микробычки (microbumps) и TSV.	Позволяет "складывать" память над логикой.	Samsung HBM, Intel Foveros, AMD 3D V-Cache	Максимальная плотность, минимальные задержки, высокая пропускная способность	Тепловые проблемы, сложность охлаждения
EMIB (Embedded Multi-die Interconnect Bridge)	Активная межсоединительная "дорожка" встраивается в подложку.	Гибкая, масштабируемая альтернатива интерпосеру.	Intel FPGAs, Ponte Vecchio, Lunar Lake	Дешевле 2.5D, поддержка гетерогенных чиплетов	Ограниченная длина моста
Foveros (Intel)	3D-стекинг логики на логику. Позволяет размещать CPU/GPU поверх "базового чиплета".	Использует TSV и микробычки.	Intel Arrow Lake, Lunar Lake	Компактность, disaggregation	Сложное управление теплом
CoWoS (Chip-on-Wafer-on-Substrate)	Технология TSMC: чипы и HBM размещаются на кремниевом интерпосере.	Поддерживает HBM и несколько чиплетов.	NVIDIA H100, AMD Instinct MI300, Apple M4 (ожидается)	Высокая производительность для AI/HPC	Высокая стоимость, ограниченная доступность
SoIC (System-on-Integrated-Chips, TSMC)	3D-стекинг без микробычков — прямое соединение кристаллов (hybrid bonding).	Наиболее плотное соединение.	MediaTek, будущие GPU и CPU	Наилучшая плотность и энергоэффективность	На ранней стадии массового внедрения
UCIe (Universal Chiplet Interconnect Express)	Открытый стандарт соединения чиплетов. Поддерживает coherent-обмен.	Объединяет chiplets от разных вендоров.	Intel, AMD, TSMC, Qualcomm (с 2023)	Интероперабельность, модульность, disaggregation	Требует новых инструментов проектирования

Межкристальные и упаковочные технологии

- **Тренды и будущее**
  - **Chiplet-экономика:** вместо "один большой кристалл" — модульная сборка из специализированных чиплетов.
  - **UCIe** — попытка создать экосистему совместимых чиплетов от разных производителей.
  - **3D-интеграция** — путь к memory-centric computing (память над процессором).
  - **Disaggregation**: разделение CPU, GPU, памяти, хранилища на отдельные пулы, соединённые через CXL и UCIe.



Архитектура ЭВМ. Энергетическая модель и управление питанием

Энергетическая модель и управление питанием

- **Современные вычислительные системы** — от смартфонов до суперкомпьютеров — сталкиваются с ограничениями по энергопотреблению, тепловыделению и автономности.
- Управление питанием стало ключевой частью архитектуры ЭВМ.

- **Основные цели энергетической модели**
  - Минимизация потребления энергии.
  - Управление тепловыделением (TDP, температура).
  - Продление времени автономной работы (в мобильных устройствах).
  - Повышение энергоэффективности (производительность на ватт).
  - Поддержка динамической адаптации под нагрузку.

Энергетическая модель и управление питанием

- Классификация архитектур ЭВМ по энергетической модели и управлению питанием отражает переход от простого максимизации производительности к балансу между вычислительной мощностью, энергопотреблением и тепловыделением.
- Современные процессоры используют иерархию аппаратных механизмов для динамического контроля энергии.
- Ключевым из них является **DVFS (Dynamic Voltage and Frequency Scaling)** — одновременное изменение тактовой частоты и напряжения питания в зависимости от нагрузки: при низкой активности частота и напряжение снижаются, что экспоненциально уменьшает потребление (поскольку мощность пропорциональна $(C \cdot V^2 \cdot f)$.
- Дополнительно применяются **clock gating** (отключение тактирования неиспользуемых блоков) и **power gating** (полное отключение питания от «спящих» модулей), что позволяет устранять утечки тока в статическом режиме.
- На уровне чипа реализуются **power islands** — изолированные энергетические домены, каждый из которых может управляться независимо, что особенно важно в гетерогенных SoC с CPU, GPU и NPU.

Энергетическая модель и управление питанием

- Эволюция энергетических моделей привела к концепции **энергетически пропорциональных систем (energy-proportional computing)**, впервые предложенная Лукасом Церни и Лукасом Барроузом.
- Идея заключается в том, что система должна потреблять энергию, строго пропорциональную выполняемой работе: при 10% загрузке — 10% мощности, а не 50%, как в традиционных серверах.
- Хотя идеал недостижим, современные архитектуры стремятся к нему за счёт глубоких состояний простоя (C-states), тонкой гранулярности управления питанием и специализированных ускорителей, которые выполняют задачи с гораздо большей энергоэффективностью, чем универсальные ядра.
- Например, обработка видео на медиа-движке потребляет в разы меньше энергии, чем на CPU, что делает систему в целом более пропорциональной.

Энергетическая модель и управление питанием

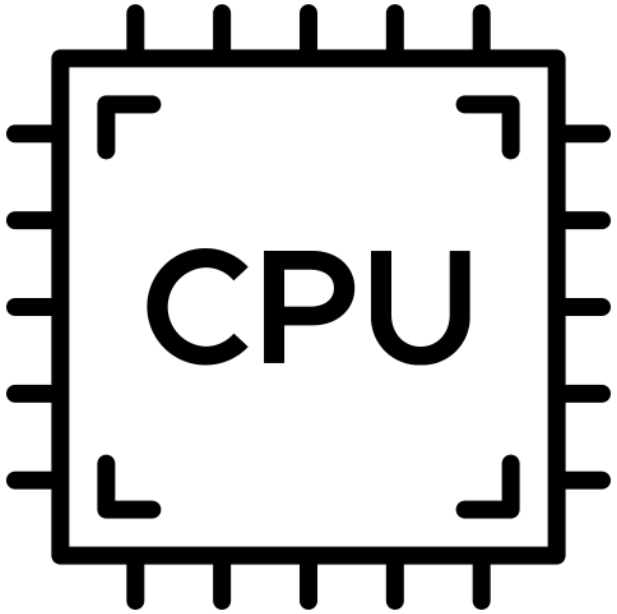
- **Управление питанием сегодня** — это не только аппаратная функция, но и результат тесного взаимодействия между **аппаратным планировщиком** и **операционной системой**.
- В гибридных архитектурах, таких как Intel Alder Lake и Arrow Lake, аппаратный модуль **Thread Director** в реальном времени анализирует характеристики потоков (интенсивность, задержки, IPC) и рекомендует ОС, на каком типе ядра (производительном P-core или энергоэффективном E-core) запускать каждый поток.
- Со стороны ОС используются продвинутые алгоритмы: **EAS (Energy-Aware Scheduling)** в Linux учитывает энергетические характеристики CPU при распределении задач, а **CPPC (Collaborative Processor Performance Control)** позволяет процессору и ОС совместно управлять частотой и напряжением на основе метрик производительности и энергопотребления.
- Это обеспечивает оптимальный баланс между отзывчивостью и энергоэффективностью, особенно в мобильных и ноутбучных сценариях.

Энергетическая модель и управление питанием

- Наконец, энергетическая модель влияет на проектирование всей вычислительной платформы — от кристалла до дата-центра.
- В мобильных SoC (Apple, Qualcomm) энергопотребление является главным ограничением, поэтому там активно используются ультраэффективные ядра (Cortex-M, RISC-V), специализированные NPU и агрессивное управление питанием.
- В серверах же, несмотря на высокую общую мощность, критична **производительность на ватт**, что стимулирует переход на ARM и RISC-V, использование жидкостного охлаждения и разработку «зелёных» ЦОД.
- Таким образом, современная энергетическая модель — это комплексная, многоуровневая стратегия, объединяющая аппаратные инновации, интеллектуальное планирование и системное ПО для достижения максимальной эффективности в условиях всё более жёстких требований к устойчивости и автономности.

Классификация по энергетической модели и механизмам управления питанием

Механизм / технология	Описание	Уровень применения	Примеры	Преимущества	Недостатки
Динамическое масштабирование напряжения и частоты (DVFS) (Dynamic Voltage and Frequency Scaling)	**Изменение частоты и напряжения процессора в зависимости от нагрузки.**	Ядро, SoC, CPU, GPU	Intel SpeedStep, AMD Cool'n'Quiet, ARM DVFS	Экономия энергии при низкой нагрузке	Задержки при изменении режима
P-States (Performance States)	**Режимы производительности: от высокой (P0) до энергосберегающей (Pn).**	ACPI, ОС, BIOS	x86, ARM, RISC-V	Поддержка ОС и драйверами	Требует сложной политики управления
C-States (Idle States)	**Режимы простоя: от лёгкого (C1) до глубокого сна (C6/C7).**	Ядро, кэш, кристалл	Intel C-states, AMD C-nap, ARM WFI/WFE	Снижение энергопотребления в простое	Большие задержки выхода из глубоких состояний
D-States (Device Power States)	**Управление питанием периферийных устройств (PCIe, USB, дисплей).**	Устройства, шины	PCIe D0–D3, USB Suspend	Экономия на I/O	Требует координации с драйверами
Heterogeneous Power Management (big.LITTLE, DynamIQ)	**Разные ядра с разным энергопотреблением. Задачи переносятся на "лёгкие" ядра при низкой нагрузке.**	SoC (мобильные, embedded)	ARM big.LITTLE, DynamIQ, Apple M-series E-cores	Высокая энергоэффективность	Сложность планировщика ОС
Power Gating (отключение питания)	**Полное отключение питания блоков (ядер, кэшей, функциональных узлов).**	Логические блоки, кластеры	ARM Power Domains, Intel Unused Core Power Down	Нулевое потребление в простое	Требует сохранения состояния
Clock Gating (отключение тактирования)	**Остановка тактового сигнала для неактивных блоков.**	На уровне логики	Встроено в большинство современных CPU	Низкие накладные расходы, мгновенное включение	Не устраняет утечки (leakage)
Adaptive Voltage Scaling (AVS)	**Динамическая подстройка напряжения на основе температуры и PVT (Process-Voltage-Temperature).**	SoC, high-end CPU	Qualcomm AVS, Intel Adaptive Boost	Оптимизация напряжения "по ходу"	Требует сенсоров и контроля в реальном времени
Fine-Grained Power Management	**Управление питанием на уровне отдельных функциональных блоков (ALU, FPU, SIMD).**	Внутри ядра	ARM Neoverse, RISC-V с PMP	Высокая точность управления	Сложность проектирования
Energy-Aware Scheduling (EAS)	**Планировщик ОС учитывает энергоэффективность ядер при распределении задач.**	ОС (Linux, Android)	Linux EAS, Android HMP	Максимизация производительности на ватт	Требует поддержки ядра и SoC
Thermal Throttling (тепловое ограничение)	**Снижение частоты при превышении температуры.**	CPU, GPU, SoC	Intel Thermal Velocity Boost, AMD Precision Boost Overdrive	Защита от перегрева	Падение производительности
Near-Threshold Computing (NTC)	**Работа при напряжении близком к пороговому — для минимального энергопотребления.**	Исследовательские и ultra-low-power системы	Intel Claremont, MC13224V	Экстремально низкое энергопотребление	Очень низкая производительность, нестабильность
Approximate Computing	**Сознательная жертва точности ради энергосбережения (в ИИ, мультимедиа).**	DSP, NPU, GPU	8-битные вычисления в ИИ, lossy prefetch	До 10× энергоэффективности	Неприемлемо для критических задач



Примеры современных архитектур и их применение

Примеры современных архитектур и их применение

- К 2025 году **ландшафт современных архитектур ЭВМ характеризуется глубокой гетерогенностью, специализацией и переходом от универсальных процессоров к вычислительным платформам, оптимизированным под конкретные домены, задачи**.
- В сегменте клиентских устройств доминируют гибридные архитектуры: Intel **Arrow Lake** и **Lunar Lake** с комбинацией P- и E-ядер, отказом от SMT в настольных версиях и мощными ИИ-ускорителями NPU; **AMD Zen 5** с рекордным IPC, поддержкой AVX-512 и 3D V-Cache для игр; а также **Apple Silicon** (M3/M4), где единая SoC-платформа объединяет CPU, GPU, Neural Engine и память в едином кристалле с ультравысокой энергоэффективностью.
- Эти архитектуры активно используют чиплеты, **UCIe-интерконнекты и аппаратные механизмы безопасности** (SEV-SNP, TDX, CCA), что делает их основой не только для ПК и ноутбуков, но и для edge-устройств с локальным ИИ.

Примеры современных архитектур и их применение

- В дата-центрах и HPC наблюдается поляризация: с одной стороны, масштабируемые **x86-платформы** (Intel Sapphire Rapids/Granite Rapids, AMD EPYC Turin) с поддержкой **CXL, HBM и Confidential Computing**; с другой — рост экосистемы **ARM** (Ampere Altra, AWS Graviton4) и **RISC-V** (встраиваемые ускорители, DPU).
- Ключевую роль играют специализированные ускорители: **NVIDIA Blackwell** и **AMD MI300X** для ИИ и научных вычислений, **Google TPU v5e, Intel Gaudi 3**, а также **DPU/SmartNIC** (NVIDIA BlueField-3, AMD Pensando) для разгрузки сетевых и инфраструктурных задач.
- Архитектуры всё чаще проектируются как пулы ресурсов, где память, вычисления и хранилище дезагрегированы и объединены через **CXL.fabric**, что позволяет динамически выделять ресурсы под рабочую нагрузку — тренд, который будет только усиливаться к 2030 году.

Примеры современных архитектур и их применение

- В области искусственного **интеллекта формируется новая парадигма — нейроморфные и квантово-вдохновлённые архитектуры**.
- Хотя **Intel Loihi 2** и **IBM NorthPole** пока находятся в исследовательской фазе, они демонстрируют потенциал спайковых нейросетей для ультраэнергоэффективной обработки сенсорных данных в реальном времени.
- Параллельно развиваются **оптические вычисления** и **in-memory computing,** направленные на преодоление «узкого горлышка фон Неймана».
- К 2030 году ожидается появление гибридных систем, где традиционные CPU/GPU работают в связке с нейроморфными чипами для задач восприятия, а квантовые процессоры (IBM, Google, IonQ) — для специфических алгоритмов (криптография, оптимизация). Эти архитектуры будут применяться в автономных транспортных средствах, робототехнике и персонализированной медицине.

Примеры современных архитектур и их применение

- **Прогноз до 2030 года указывает на дальнейшую дезагрегацию, открытость и адаптивность.**
- **RISC-V станет доминирующей** ISA в IoT, edge и даже в серверных ускорителях благодаря модульности и отсутствию лицензионных ограничений.
- **UCIe и CXL** создадут единый стандарт для сборки гетерогенных систем из чиплетов разных производителей.
- Энергоэффективность станет главным ограничением: архитектуры будут проектироваться по принципу «**вычисления там, где данные**», с интеграцией памяти и логики (3D-stacking, chiplets с HBM).
- Наконец, **безопасность и конфиденциальность** станут неотъемлемой частью архитектуры на всех уровнях — от защищённых анклавов до квантово-устойчивой криптографии.
- Таким образом, **будущее архитектур ЭВМ — не в «одном чипе для всего», а в гибких, безопасных, энергоэффективных и специализированных вычислительных тканях, адаптирующихся под меняющиеся потребности цифрового мира.**

ЭВМ, периферийные устройства и контроллеры

Тема: Архитектурные особенности организации ЭВМ различных классов

Благодарю за внимание

КУТУЗОВ Виктор Владимирович

Белорусско-Российский университет, Кафедра «Программное обеспечение информационных технологий»
Республика Беларусь, Могилев, 2025