УТВЕРЖДАЮ
Зав. Кафедры программной инженерии БГТУ
к.т.н., доцент_____Н.В. Пацей
протокол № 4 от 30 ноября 2021 г.

Экзаменационные вопросы

Дисциплина: «Арифметико-логические основы вычислительных систем»

Специальности: 1-40 01 01 «Программное обеспечение информационных технологий»
1-40 05 01-03 «Информационные системы и технологии»
1-98 01 03 «Программное обеспечение информационной безопасности мобильных систем»

Экзамен устно-письменный.

1. Системы счисления. Перевод чисел из одной системы счисления в другую. Метод преобразования с использованием весов разрядов.
2. Перевод чисел из одной системы счисления в другую. Метод деления (умножения) на новое основание.
3. Перевод чисел из одной системы счисления в другую. Метод с использованием особого соотношения оснований исходной и искомой систем счисления.
4. Арифметические операции над двоичными числами. Операция сложения и вычитания в двоичной системе исчисления.
5. Операция умножения в двоичной системе исчисления
6. IEEE754. Специальные числа. Зачем нулю знак.
7. Деление двоичных чисел (общие правила).
8. Деление двоичных чисел с восстановлением остатка.
9. Деление двоичных чисел без восстановления остатка.
10. Двоично-десятичная арифметика. Сложение и вычитание двоично-десятичных чисел .
11. Кодирование алгебраических чисел. Дополнительный и обратный коды двоичных чисел.
12. Операции с двоичными числами в дополнительном и обратном кодах.
13. Модифицированные коды
14. Логические операции с двоичными кодами: логическое суммирование, логическое умножение, логическое отрицание, суммирование по модулю два, логические сдвиги .
15. Арифметические сдвиги положительных двоичных чисел, представленных в прямом коде. Арифметические сдвиги двоичных чисел, представленных в обратном коде.
16. Арифметические сдвиги двоичных чисел, представленных в дополнительном коде. Сдвиг отрицательных чисел с переполнением.
17. Представление чисел с фиксированной точкой. Арифметические операции над числами, представленными с фиксированной точкой.
18. Представление чисел с плавающей точкой. Сложение чисел, представленных в формате с плавающей точкой
19. Умножение чисел, представленных в формате с плавающей точкой. Деление чисел, представленных в формате с плавающей точкой.
20. Неосновные арифметические операции. Вычисление квадратного корня
21. Методы вычисления элементарных функций.
22. Денормализованные числа. Подводные камни в арифметике с плавающей запятой.
23. Погрешности обусловленные форматом с плавающей точкой
24. Основные понятия алгебры логики. Способы задания логической функции.

25. Понятие о принципе двойственности. Суперпозиция логических функций.
26. Нормальная и совершенные нормальные логических функций.
27. Минимизация булевых функций. Основные понятия. Наиболее известные методы минимизации. Минимизация системы логических функций. Минимизация частично определенных функций.
28. Минимизация логических выражений методом Квайна.
29. Минимизация логических выражений с использованием Карт Карно (диаграммами Вейча).
30. Синтез логических схем по логическим выражениям в булевом базисе. Логический базис И-НЕ. Логический базис ИЛИ-НЕ.
31. Законы и правила алгебры Буля
32. Параллелизм. Виды, организация.
33. Устройства ЭВМ. Состав АЛУ.
34. Типы памяти.
35. Код Грея.
36. Обратная польская запись.
37. АЦП и ЦАП. Предназначение. Параметры сравнения и выбора.
38. Корректирующие коды. Код Хэ́мминга. Область применения.
39. Языки описания аппаратуры. ПЛИС (FPGA) модули.
40. Сумматор. Многоразрядный сумматор. Ускорение выполнения математических операций.
41. Полная система логических функций.
42. Искусство управления сложностью. Цифровая абстракция.
43. Логические элементы. Таблицы истинности. Обозначения элементов в разных представления.
44. За пределами цифровой абстракции. Напряжение питания. Логические уровни. Допускаемые уровни шумов.
45. Передаточная характеристика. Статическая дисциплина.
46. Биполярные и КМОП транзисторы. Полупроводники. Конденсаторы. n-МОП и p-МОП-транзисторы
47. Логический вентиль НЕ и другие на КМОП-транзисторах. Псевдо n-МОП-Логика Потребляемая мощность
48. Проектирование комбинационной логики. От логики к логическим элементам, Что такое X и Z: способы сопряжения микросхем в ЭВМ.
49. Временные характеристики цифровых микросхем. Задержка распространения и задержка реакции. Импульсные помехи.
50. Базовые комбинационные блоки. Мультиплексоры. Логика на мультиплексорах. Дешифраторы
51. Проектирование последовательностной логики. Защелки и триггеры. RS-триггер. D-защелка. D-Триггер. Регистр.
52. Триггер с функцией разрешения. Триггер с функцией сброса. Проектирование синхронных логических схем. Синхронные последовательностные схемы. Синхронные и асинхронные схемы.
53. Конечные автоматы. Пример проектирования конечного автомата
54. Конечные автоматы. Кодирование состояний. Автоматы Мура и Мили.
55. Декомпозиция конечных автоматов. Восстановление конечных автоматов по электрической схеме.
56. Синхронизация последовательностных схем. Временные характеристики системы. Расфазировка тактовых сигналов. Метастабильность. Синхронизаторы.
57. Типы триггеров. Классификация триггеров. RS-триггер на элементах И-НЕ и ИЛИ–НЕ. T-, JK-, D-триггеры.
58. Параллельные и последовательные регистры. Отличия в обозначения цифровых элементов в разных стандартах.

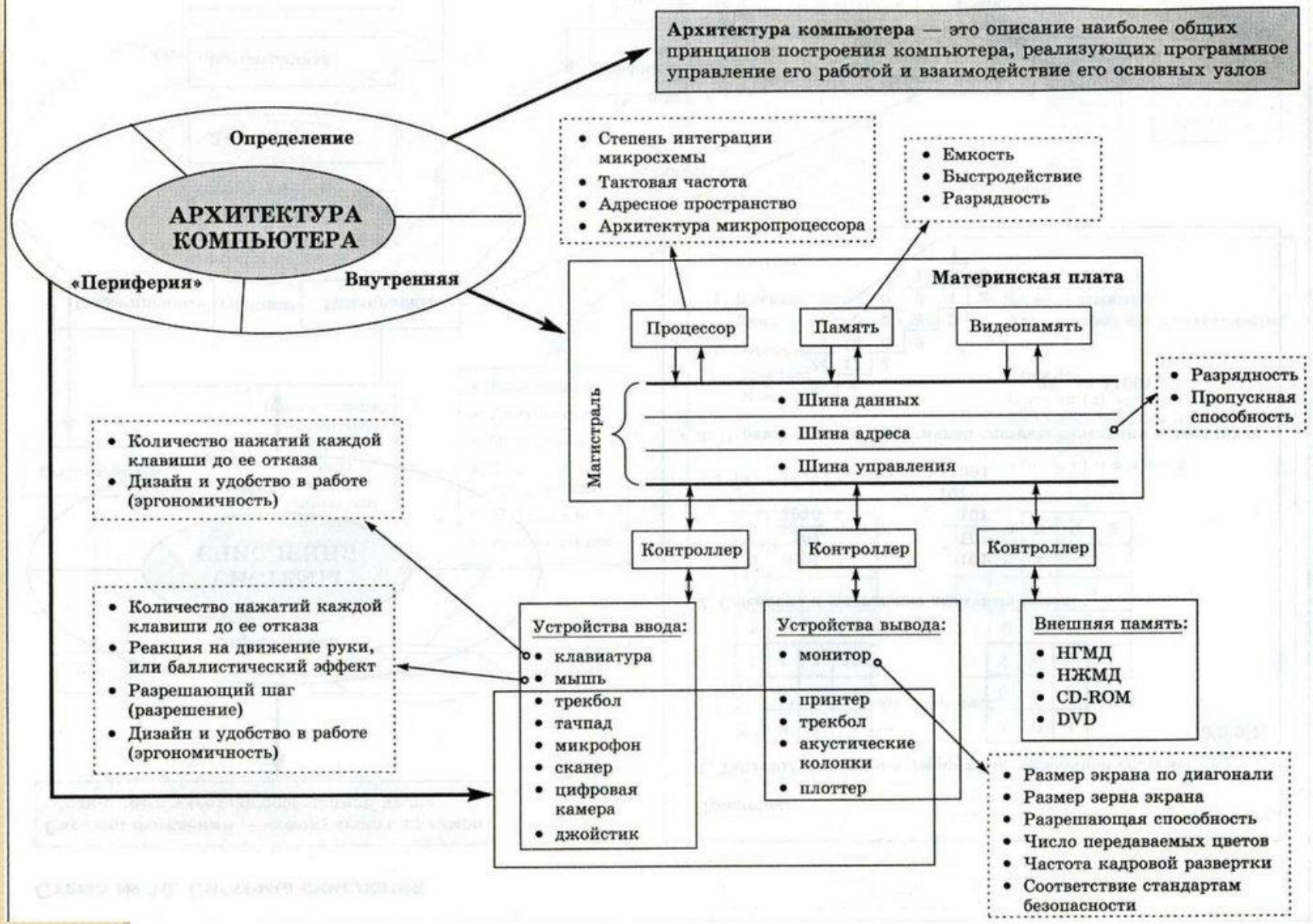
59. Мультиплексоры и демультиплексоры. Отличия в обозначения цифровых элементов в разных стандартах.
60. Погрешность метаматематических операций в цифровых системах. Способы оценки. Округление.
61. Архитектура процессора. Основные компоненты. Способы классификации. Много уровневая организация. Контроллеры вввода-вывола.
62. RISK, CISK, MISC, VLIW. Отличительные особенности, сфера применения. Что такое Spectre и Meltdown.
63. Сравнительная характеристика архитектур. В чем преимущества. Преимущества RISK. Какова проблема лицензирования архитектур.
64. Виртуальные архитектуры. Команды (инструкции), предназначение, виды. Тактирование процессоров. Выполнение инструкций. Поток инструкций
65. Регистр процессора: предназначение, виды. Шины:предназначение, виды.. Кэш: предназначение, виды.
66. Что такое суперскалярная архитектура. Ее особенности. Предсказатели переходов. Иерархия памяти. Ветвление
67. Что такое гетерагенные вычисления. FPGA-акселератор? Сфера применения. Перспективные направления развития вычислительных систем.

Лектор Гринюк Д.А.

Архитектура процессора

- **Архитекту́ра проце́ссора** — количественная составляющая компонентов микроархитектуры вычислительной машины (процессора компьютера) (например, регистр флагов или регистры процессора), рассматриваемая IT-специалистами в аспекте прикладной деятельности.
- С точки зрения:
- программиста — совместимость с определённым набором команд (например, процессоры, совместимые с командами Intel x86), их структуры (например, систем адресации или организации регистровой памяти) и способа исполнения (например, счётчик команд).
- аппаратной составляющей вычислительной системы — это некий набор свойств и качеств, присущий целому семейству процессоров (иначе говоря — «внутренняя конструкция», «организация» этих процессоров).
- Имеются различные классификации архитектур процессоров как по организации (например, по количеству и сложности отдельных команд: RISC, CISC; по возможности доступа команд к памяти[1]), так и по назначению (например, специализированные графические, математические или предназначенные для цифровой обработки сигналов).

Архитектура компьютера

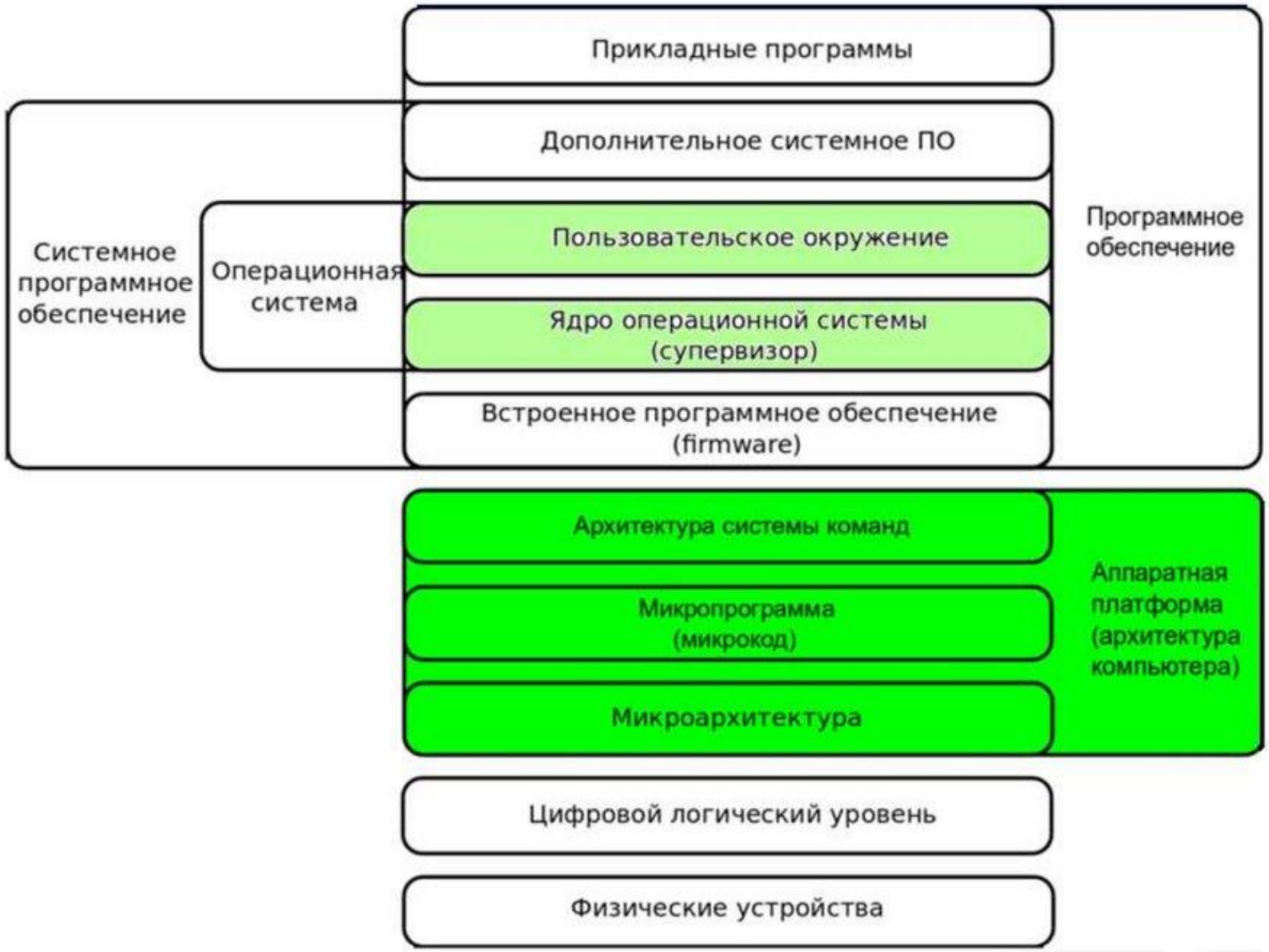


Контроллеры ввода-вывода

- **Super I/O** (англ. *Super Input/output)* — название класса сопроцессоров, которые начали использоваться после 1980-х годов на материнских платах IBM PC-совместимых компьютеров путём сочетания функций многих контроллеров, сперва одной платой, устанавливаемой в слот расширения, а затем и одной микросхемой, тем самым достигая уменьшения числа микросхем контроллеров, и таким образом привели к снижению сложности и стоимости компьютера в целом. Super I/O объединяет интерфейсы различных низкоскоростных устройств.
- Фактически, на рынке материнских плат массового сегмента предоставлена продукция всего четырёх фирм: ITE Tech, Nuvoton, Microchip Technology и Fintek. Производители брендовой продукции (IBM, HP, Dell, FSC) как правило используют проприетарные разработки схемотехники материнских плат и не публикуют спецификаций применяемых контроллеров.
- Как правило, включает в себя следующие функции:
- контроллер дисковода гибких дисков (floppy);
- контроллер параллельного порта (LPT-порт);
- контроллер последовательных (COM) портов и портов клавиатуры и мыши (PS/2).
- Super I/O также может включать в себя и другие интерфейсы, такие как игровой (MIDI или джойстик) или инфракрасный порты.
- контроллер Ethernet
- Изначально Super I/O связывались через шину ISA. Одновременно с развитием IBM PC-совместимых компьютеров происходило смещение Super I/O, сперва на шины VLB, затем стала использоваться шина PCI. Современные Super I/O используют шину LPC (интерфейс которой предоставляет южный мост материнской платы) и часто реализованы в составе чипсета.

- Схемотехника материнской платы предполагает наличие цепей измерения, которые производятся с помощью аналогово-цифровых преобразователей, преобразующих измеряемый параметр в цифровые значения, после чего они могут быть переданы в другое вычислительное устройство на плате для дальнейшей обработки. Измерения, производимые на материнской плате, в основном касаются трёх групп параметров: обороты вентиляторов, температура и напряжения.
- Скорость вращения вентиляторов, применяемых для охлаждения блоков и отдельных частей материнской платы, обычно контролируется при помощи тахометров, встроенных в вентилятор — обычно для этого используется датчик Холла. Такой вентилятор отличается дополнительными (кроме двух питания) проводами, одним (устанавливаемый в корпус, блок питания, на охлаждение радиаторов микросхем чипсепа материнской платы/видеокарты) или двумя (процессорный).
- Для мониторинга температуры используется три типа датчиков: терморезисторы, транзисторы (например, 2N3904) и датчики интегрированные в процессор.
- Логически аппаратный мониторинг выглядит как набор регистров, значение которых изменяется при изменении состояния на входах.
- **USB-контроллер** в составе платформы персонального компьютера обеспечивает коммуникацию с периферийными устройствами, подключенными к USB. USB-контроллер является устройством, способным взаимодействовать с оперативной памятью в обход центрального процессора в режиме прямого доступа к памяти.
- По способу интеграции контроллер для USB-шины может быть задействован в составе системной логики или в виде дискретного чипа как на самой системной плате, так и на плате расширения. По способу подключения USB-контроллер может быть выполнен для PCI-шины, либо для шины PCI Express.

Многоуровневая организация компьютера



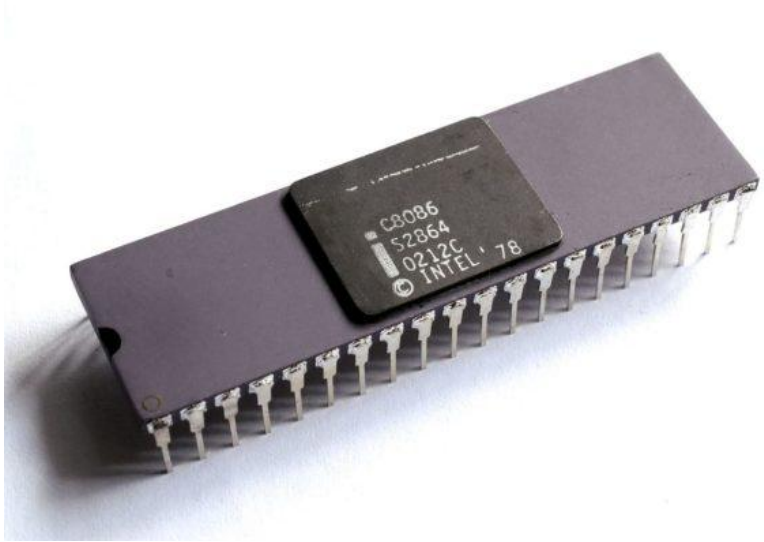
Виды архитектур

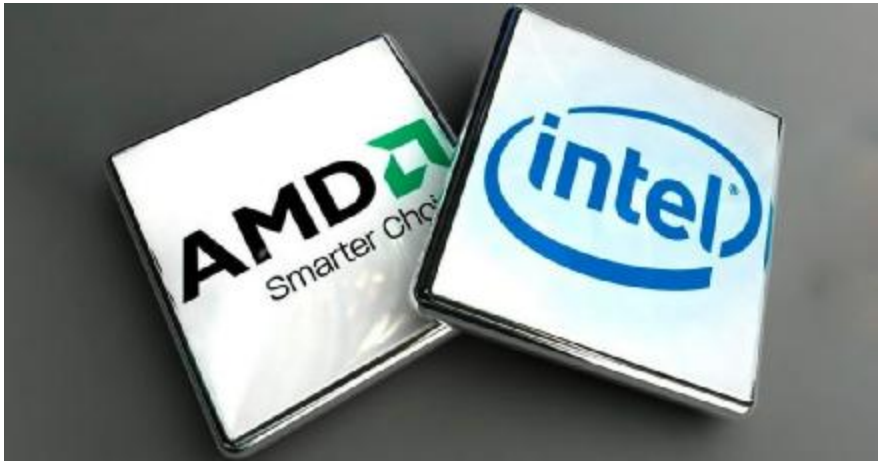
RISC

- RISC (англ. Reduced Instruction Set Computer — «компьютер с сокращённым набором команд») — архитектура процессора, в котором быстродействие увеличивается за счёт упрощения инструкций: их декодирование становится более простым, а время выполнения — меньшим. Первые RISC-процессоры не имели даже инструкций умножения и деления и не поддерживали работу с числами с плавающей запятой.
- По сравнению с CISC эта архитектура имеет константную длину команды, а также меньшее количество схожих инструкций, позволяя уменьшить итоговую цену процессора и энергопотребление, что критично для мобильного сегмента. У RISC также большее количество регистров.
- Примеры RISC-архитектур: PowerPC, серия архитектур ARM (ARM7, ARM9, ARM11, Cortex).
- В общем случае RISC быстрее CISC. Даже если системе RISC приходится выполнять 4 или 5 команд вместо одной, которую выполняет CISC, RISC все равно выигрывает в скорости, так как RISC-команды выполняются в 10 раз быстрее.
- Отсюда возникает закономерный вопрос: почему многие всё ещё используют CISC, когда есть RISC? Всё дело в совместимости. x86_64 всё ещё лидер в desktop-сегменте только по историческим причинам. Так как старые программы работают только на x86, то и новые desktop-системы должны быть x86(_64), чтобы все старые программы и игры могли работать на новой машине.
- Для Open Source это по большей части не является проблемой, так как пользователь может найти в интернете версию программы под другую архитектуру. Сделать же версию проприетарной программы под другую архитектуру может только владелец исходного кода программы.

CISC

- CISC (англ. Complex Instruction Set Computer — «компьютер с полным набором команд») — тип процессорной архитектуры, в первую очередь, с нефиксированной длиной команд, а также с кодированием арифметических действий в одной команде и небольшим числом регистров, многие из которых выполняют строго определенную функцию.
- Самый яркий пример CISC архитектуры — это x86 (он же IA-32) и x86_64 (он же AMD64).
- В CISC процессорах одна команда может быть заменена ей аналогичной, либо группой команд, выполняющих ту же функцию. Отсюда вытекают плюсы и минусы архитектуры: высокая производительность благодаря тому, что несколько команд могут быть заменены одной аналогичной, но большая цена по сравнению с RISC процессорами из-за более сложной архитектуры, в которой многие команды сложнее раскодировать.





MISC

- MISC (англ. Minimal Instruction Set Computer — «компьютер с минимальным набором команд»).
- Ещё более простая архитектура, используемая в первую очередь для ещё большего уменьшения итоговой цены и энергопотребления процессора. Используется в IoT-сегменте и недорогих компьютерах, например, роутерах.
- Для увеличения производительности во всех вышеперечисленных архитектурах может использоваться “спекулятивное исполнение команд”. Это выполнение команды до того, как станет известно, понадобится эта команда или нет.

ARM

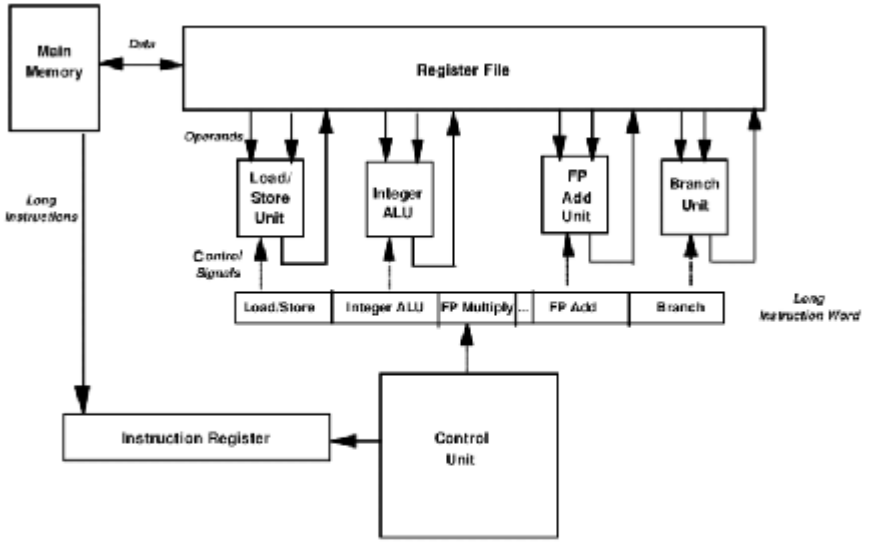
- **Архитектура** (от англ. *Advanced RISC Machine* — усовершенствованная RISC-машина; иногда — *Acorn RISC Machine*) — система команд и семейство описаний и готовых топологий 32-битных и 64-битных микропроцессорных/микроконтроллерных ядер, разрабатываемых компанией ARM Limited
- Среди лицензиатов готовых топологий ядер ARM — компании AMD, Apple, Analog Devices, Atmel, Xilinx, Cirrus Logic[en], Intel (до 27 июня 2006 года), Marvell, NXP, STMicroelectronics, Samsung, LG, MediaTek, Qualcomm, Sony, Texas Instruments, Nvidia, Freescale, Миландр, ЭЛВИС[2], HiSilicon, Байкал электроникс.
- Значимые семейства процессоров: ARM7, ARM9, ARM11 и Cortex.
- В 2006 году около 98 % из более чем миллиарда мобильных телефонов, продававшихся ежегодно, были оснащены, по крайней мере, одним процессором ARM[5]. По состоянию на 2009, на процессоры ARM приходилось до 90 % всех встроенных 32-разрядных процессоров[6]. Процессоры ARM широко используются в потребительской электронике — в том числе смартфонах, мобильных телефонах и плеерах, портативных игровых консолях, калькуляторах, умных часах и компьютерных периферийных устройствах, таких, как жесткие диски или маршрутизаторы.
- Многие лицензиаты проектируют собственные топологии ядер на базе системы команд ARM

- Эти процессоры имеют низкое энергопотребление, поэтому находят широкое применение во встраиваемых системах и преобладают на рынке мобильных устройств, для которых данный фактор немаловажен.
- В основном процессоры семейства завоевали сегмент массовых мобильных продуктов (сотовые телефоны, карманные компьютеры) и встраиваемых систем средней и высокой производительности (от сетевых маршрутизаторов и точек доступа до телевизоров). Отдельные компании заявляют о разработках эффективных серверов на базе кластеров ARM-процессоров, но пока это только экспериментальные проекты с 32-битной архитектурой
- Уже давно существует справочное руководство по архитектуре ARM, которое разграничивает все типы интерфейсов, которые поддерживает ARM, так как детали реализации каждого типа процессора могут различаться. Архитектура развивалась с течением времени и, начиная с ARMv7, были определены 3 профиля:
- A (application) — для устройств, требующих высокой производительности (смартфоны, планшеты);
- R (real time) — для приложений, работающих в реальном времени;
- M (microcontroller) — для микроконтроллеров и недорогих встраиваемых устройств.
- Профили могут поддерживать меньшее количество команд (команды определенного типа).
- Набор команд
- Чтобы сохранить устройство чистым, простым и быстрым, оригинальное изготовление ARM было исполнено без микрокода, как и более простой 8-разрядный процессор 6502, используемый в предыдущих микрокомпьютерах от Acorn Computers.
- Набор команд ARM. Режим, в котором исполняется 32-битный набор команд. ADC, ADD, AND, B/BL, BIC, CMN, CMP, EOR, LDM, LDR/LDRB, MLA, MOV, MUL, MVN, ORR, RSB, RSC, SBC, STM, STR/STRB, SUB, SWI, SWP, TEQ, TST
- Набор команд Thumb. Для улучшения плотности кода процессоры, начиная с ARM7TDMI, снабжены режимом «thumb». В этом режиме процессор выполняет альтернативный набор 16-битных команд. Большинство из этих 16-разрядных команд переводится в нормальные команды ARM. Уменьшение длины команды достигается за счёт сокрытия некоторых операндов и ограничения возможностей адресации по сравнению с режимом полного набора команд ARM.
- В режиме Thumb меньшие коды операций обладают меньшей функциональностью. Например, только ветвления могут быть условными, и многие коды операций имеют ограничение в виде доступа только к половине главных регистров процессора.

VLIW

- VLIW (англ. Very Long Instruction Word — «очень длинная машинная команда») — архитектура процессоров с несколькими вычислительными устройствами. Характеризуется тем, что одна инструкция процессора содержит несколько операций, которые должны выполняться параллельно. По сути является архитектурой CISC со своим аналогом спекулятивного исполнения команд, только сама спекуляция выполняется во время компиляции, а не во время работы программы, из-за чего уязвимости Meltdown и Spectre невозможны для этих процессоров.

Компиляторы для процессоров этой архитектуры сильно привязаны к конкретным процессорам. Например, в следующем поколении максимальная длина «очень длинной команды» может из условных 256 бит стать 512 бит, и тут приходится выбирать между увеличением производительности путём компиляции под новый процессор и обратной совместимостью со старым процессором. Опять же, Open Source позволяет простой перекомпиляцией получить программу под конкретный процессор.
Примеры архитектуры: Intel Itanium, Эльбрус-3.



- **Spectre** — группа аппаратных уязвимостей, ошибка в большинстве современных процессоров, имеющих спекулятивное выполнение команд (англ.)рус. и развитое предсказание ветвлений, позволяющих проводить чтение данных через сторонний канал в виде общей иерархии кэш-памяти. Затрагивает большинство современных микропроцессоров, в частности, архитектур x86/x86_64 (Intel и AMD) и некоторые процессорные ядра ARM.
- Уязвимость потенциально позволяет локальным приложениям (локальному атакующему, при запуске специальной программы) получить доступ к содержимому виртуальной памяти текущего приложения или других программ

Ошибка Spectre позволяет злонамеренным пользовательским приложениям, работающим на данном компьютере, получить доступ на чтение к произвольным местам компьютерной памяти, используемой процессом-жертвой, например другими приложениями (то есть нарушить изоляцию памяти между программами). Атаке Spectre подвержено большинство компьютерных систем, использующих высокопроизводительные микропроцессоры, в том числе персональные компьютеры, серверы, ноутбуки и ряд мобильных устройств[7]. В частности, атака Spectre была продемонстрирована на процессорах производства корпораций Intel, AMD и на чипах, использующих процессорные ядра ARM. Имеется вариант атаки Spectre, использующий JavaScript-программы для получения доступа к памяти браузеров (чтение данных других сайтов или данных, сохраненных в браузере)

- **Meltdown** — аппаратная уязвимость категории утечка по стороннему каналу, обнаруженная в ряде микропроцессоров, в частности, производства Intel и архитектуры ARM. Meltdown использует ошибку реализации спекулятивного выполнения команд (англ.)рус. в некоторых процессорах Intel и ARM (но не AMD[1][2]), из-за которой при спекулятивном выполнении инструкций чтения из памяти процессор игнорирует права доступа к страницам.
- Уязвимость позволяет локальному атакующему (при запуске специальной программы) получить несанкционированный доступ на чтение к привилегированной памяти (памяти, используемой ядром операционной системы).

Почему RISC победил CISC

Большинство современных процессоров - это RISC, либо “CISC-поверх-RISC”.

1. Реализация системы команд RISC-процессора требует меньше транзисторов

- можно снизить энергопотребление
- можно повысить тактовую частоту
- можно увеличить размер кэш-памяти

2. На практике оказалось, что при использовании CISC-компьютеров доля сложных интеллектуальных команд при выполнении программы не превышает 10-20%, а остальные команды вполне сопоставимы с RISC-аналогами.

3. RISC-программы лучше приспособлены для упреждающего выполнения, конвейерной обработки и других видов оптимизации.

Сравнение архитектур CISC, RISC, EPIC (VLIW)

Пусть в С-программе написано выражение «a = b + c + d + e;». С помощью каких команд процессора компьютер рассчитает значение a?

RISC	MISC	CISC	EPIC (VLIW)
Обнулить r2	Push b	Сложить b, c, a	Сложить b, c, r1, d, e, r2
Прочитать r1, b	Push c	Инкремент a, d	Сложить r1, r2, a, nop, nop, nop
Сложить r2, r1	Push d	Инкремент a, e	
Прочитать r1, c	Push e		
Сложить r2, r1	Add		
Прочитать r1, d	Add		
Сложить r2, r1	Add		
Прочитать r1, e	Pop a		
Сложить r2, r1			
Записать a, r1			

Отличительные признаки архитектур команд

- число, разновидности и сложность команд
- число и разновидности операндов
- совмещение выполняемой операции с обращением в
- память (или явные операции чтения/записи в память)
- длина команды (постоянная, плавающая)
- количество доступных регистров (это косвенный признак!)

Сравнительная оценка CISC-, RISC- и VLIW-архитетур

Характеристика	CISC	RISC	VLIW
Длина команды	Варьируется	Единая	Единая
Расположение полей в команде	Варьируется	Неизменное	Неизменное
Количество регистров	Несколько (часто специализированных)	Много регистров общего назначения	Много регистров общего назначения
Доступ к памяти	Может выполняться как часть команд различных типов	Выполняется только специальными командами	Выполняется только специальными командами

Преимущества RISC

- С одной стороны писать на Assembler'e под RISC процессоры не очень-то удобно. Если в лоб сравнивать код, написанный под CISC и RISC процессоры, очевидно преимущество первого.

  Так выглядит код одной и той же операции для x86 и ARM.

  x86

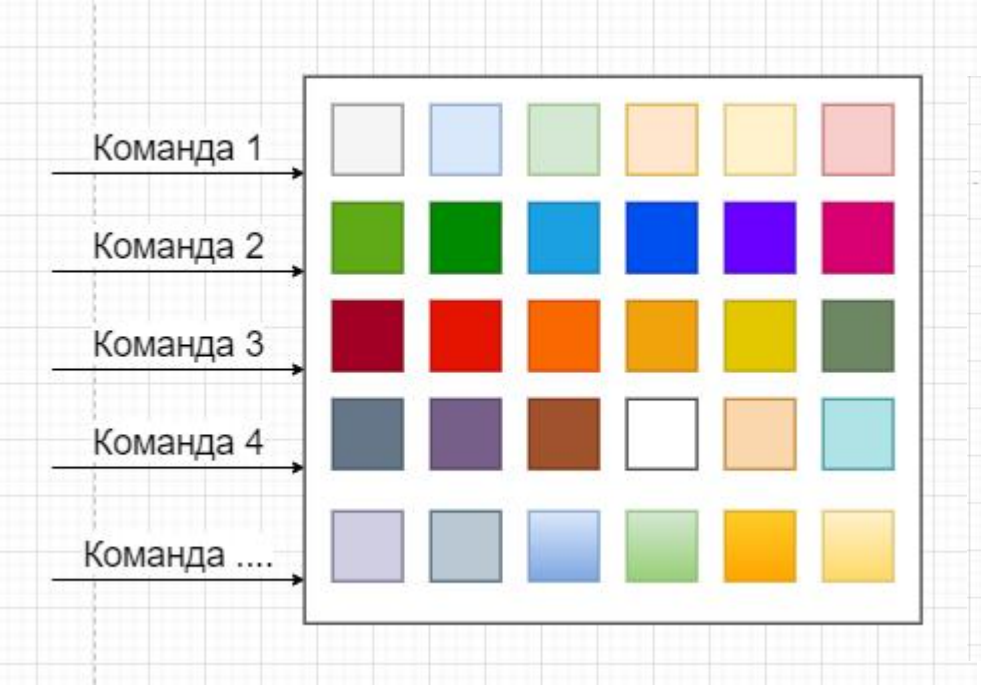
- MOV AX, 15; AH = 00, AL = 0Fh
- AAA; AH = 01, AL = 05
- RET
-

  ARM

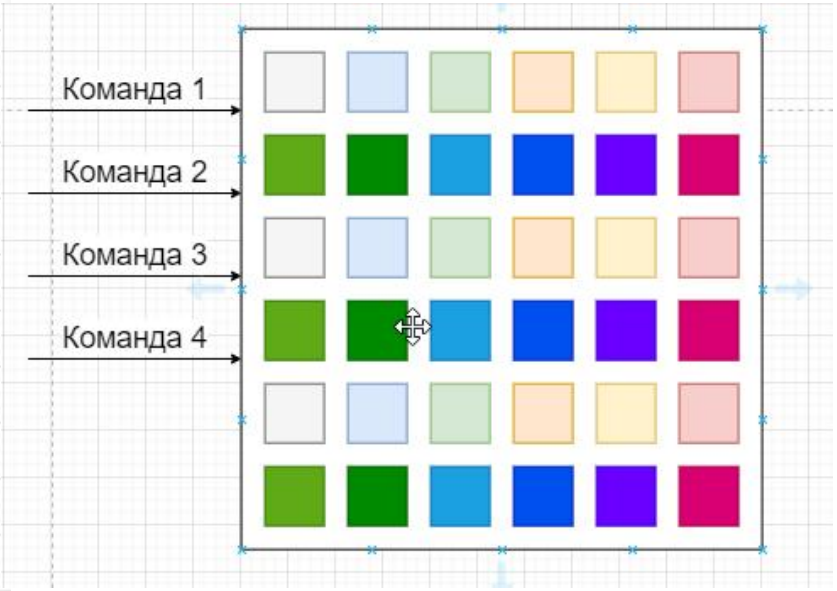
- MOV R3, #10
- AND R2, R0, #0xF
- CMP R2, R3
- IT LT
- BLT elsebranch
- ADD R2. #6
- ADD R1. #1
- elsebranch:
- END
-

- Но так было раньше. На ассемблере уже давно никто не пишет. Сейчас за программистов всё это делают компиляторы, поэтому никаких сложностей с написанием кода под RISC-процессоры нет. Зато есть преимущества. Представьте, что вы проектируете процессор. Расположение блоков на x86 выглядело бы так.

- Каждый цветной квадрат — это отдельные команды. Их много и они разные. Как вы поняли, здесь мы уже говорим про микроархитектуру, которая вытекает из набора команд. А вот ARM-процессор скорее выглядит так.



- Ему не нужны блоки, созданные для функций, написанных 50 лет назад. По сути, тут блоки только для самых востребованных команд. Зато таких блоков много. А это значит, что можно одновременно выполнять больше базовых команд. А раритетные не занимают место.



Лицензирование

- Но это все отличия технические. Есть отличия и организационные. Вы не задумывались почему для смартфонов так много производителей процессоров, а в мире ПК на x86 только AMD и Intel? Все просто — ARM это компания которая занимается лицензированием, а не производством.

  Даже Apple приложила руку к развитию ARM. Вместе с Acorn Computers и VLSI Technology. Apple присоединился к альянсу из-за их грядущего устройства — Newton. Устройства, главной функцией которого было распознавание текста.

  Даже вы можете начать производить свои процессоры, купив лицензию. А вот производить процессоры на x86 не может никто кроме синей и красной компании. А это значит что? Правильно, меньше конкуренции, медленнее развитие. Как же так произошло?

- Ну окей. Допустим ARM прекрасно справляется со смартфонами и планшетами, но как насчет компьютеров и серверов, где вся поляна исторически поделена? И зачем Apple вообще ломанулась туда со своим Apple Silicon.

Что сейчас?

- Допустим мы решили, что архитектура ARM более эффективная и универсальная. Что теперь? x86 похоронен?

  На самом деле, в Intel и AMD не дураки сидят. И сейчас под капотом современные CISC-процессоры очень похожи на RISC. Постепенно разработчики CISC-процессоров все-таки пришли к этому и начали делать гибридные процессоры, но старый хвост так просто нельзя сбросить.
- Но уже достаточно давно процессоры Intel и AMD разбивают входные инструкции на более мелкие микро инструкции (micro-ops), которые в дальнейшем — сейчас вы удивитесь — исполняются RISC ядром.

- **Те самые 4-8 ядер в вашем ПК — это тоже RISC-ядра!**

- Надеюсь, тут вы окончательно запутались. Но суть в том, что разница между RISC и CISC-дизайнами уже сейчас минимальна.

  А что остается важным — так это микроархитектура. То есть то, насколько эффективно все организовано на самом камне.

  Ну вы уже наверное знаете, что Современные iPad практически не уступают 15-дюймовым MacBook Pro с процессорами Core i7 и Core i9.

- Еще один бонус сокращенного набора RISC: меньше места на чипе занимает блок по декодированию команд. Да, для этого тоже нужно место. Архитектура RISC проще и удобнее, загибайте пальцы:

- проще работа с памятью,
- более богатая регистровая архитектура,
- легче делать 32/64/128 разряды,
- легче оптимизировать,
- меньше энергопотребление,
- проще масштабировать и делать отладку.
-

  Для примера вот два процессора одного поколения. ARM1 и Intel 386. При схожей производительности ARM вдвое меньше по площади. А транзисторов на нем в 10 раз меньше: 25 тысяч против 275 тысяч. Энергопотребление тоже отличается на порядок: 0.1 Ватт против 2 Ватт у Intel. Шок.

  Поэтому наши смартфоны, которые работают на ARM процессорах с архитектурой RISC, долго живут, не требуют активного охлаждения и такие быстрые.

Виртуальные архитектуры

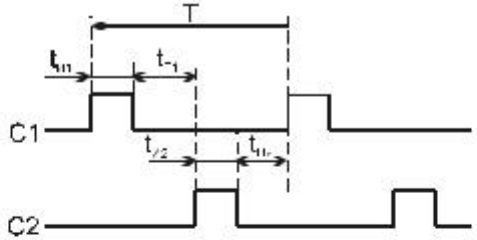
- Но раз нельзя запустить программу одной архитектуры на другой, то откуда берутся магические JAR-файлы, которые можно запустить на любой машине? Это пример виртуальной JVM-архитектуры, которая, по сути, эмулируется на целевой реальной машине. Поэтому достаточно JVM-машины для целевой архитектуры для запуска на ней любой Java-программы. Другим примером виртуальной архитектуры является .NET CIL.
- Из минусов виртуальных архитектур можно выделить меньшую производительность по сравнению с реальными архитектурами. Этот минус нивелируется с помощью JIT- и AOT-компиляции. Однако большим плюсом будет кроссплатформенность.
- Дальнейшим развитием этих архитектур стали гибридные архитектуры. Например современные x86_64 процессоры хотя и CISC-совместимы, но являются процессорами с RISC-ядром. В таких гибридных CISC-процессорах CISC-инструкции преобразовываются в набор внутренних RISC-команд. Какое дальнейшее развитие получат архитектуры процессора, покажет только время.

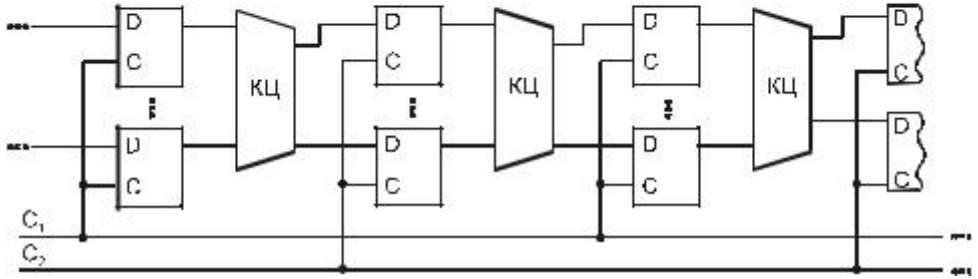
Команды (инструкции)

- **Что такое инструкция (**Команды **)?**
- Инструкция - это не что иное, как действие, которое мы отправляем процессору. Инструкции могут быть арифметическими операциями с различными типами данных, такими как с плавающей запятой, целыми числами, вектором, скаляром, логическими операциями, операциями перемещения данных, операциями перемещения битов (где бит изменяет положение), операциями перехода и т. Д.
- Эти же инструкции делятся на другие подтипы в зависимости от того, где находятся данные. Например, некоторые инструкции позволяют работать с данными, находящимися в регистрах в этот момент, в то время как в других случаях мы должны отметить адрес памяти, в котором находятся данные (прямой режим), или адрес адреса памяти (косвенный режим).
- Они бывают нескольких типов:
- **Арифметические**: сложение, вычитание, умножение и т. д.
- **Логические**: И (логическое умножение/конъюнкция), ИЛИ (логическое суммирование/дизъюнкция), отрицание и т. д.
- **Информационные**: move, input, outptut, load и store.
- **Команды перехода**: goto, if ... goto, call и return.
- **Команда останова**: halt.
- Прим. перев. На самом деле все арифметические операции в АЛУ могут быть созданы на основе всего двух: сложение и сдвиг. Однако чем больше базовых операций поддерживает АЛУ, тем оно быстрее.
- Инструкции предоставляются компьютеру на языке ассемблера или генерируются компилятором высокоуровневых языков.
- В процессоре инструкции реализуются на аппаратном уровне. За один такт одноядерный процессор может выполнить одну элементарную (базовую) инструкцию.
- Группу инструкций принято называть набором команд (англ. instruction set).

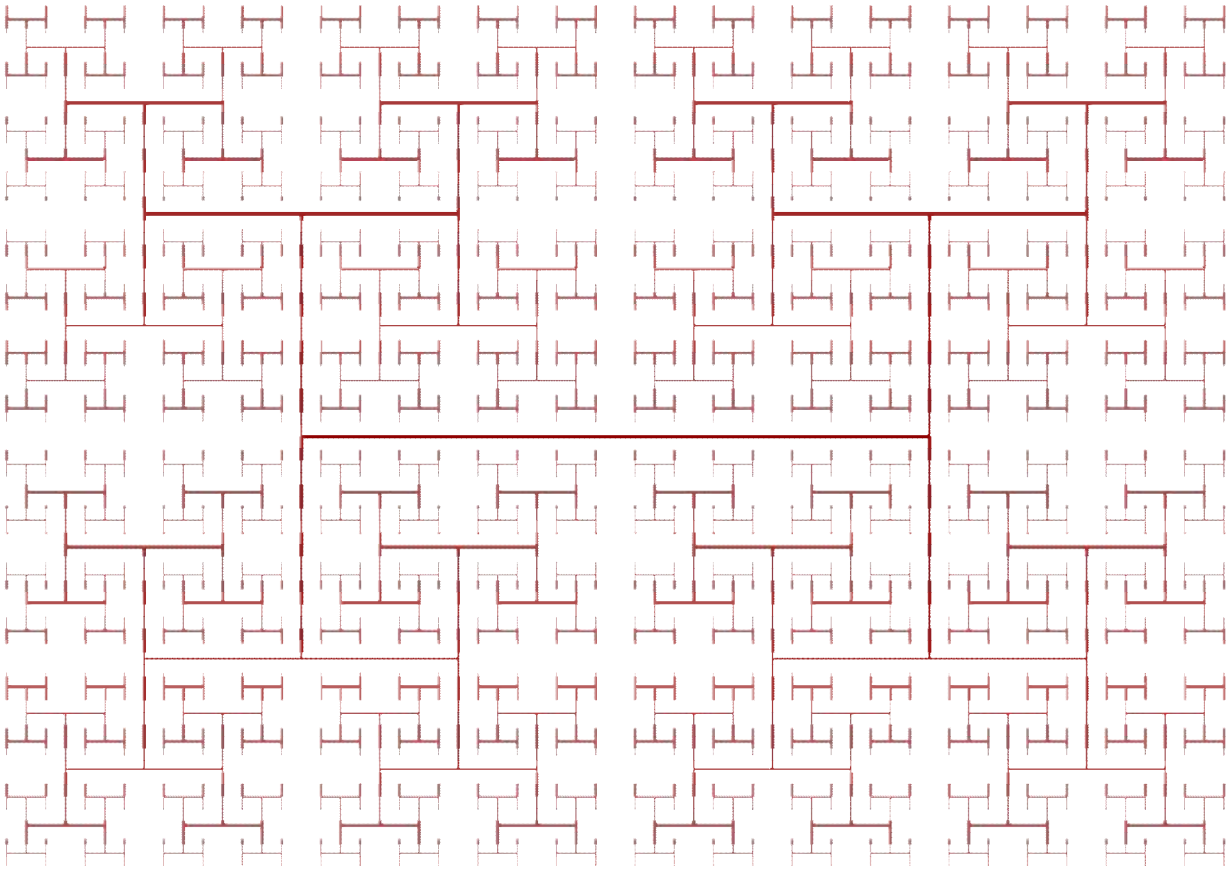
Тактирование процессора

- Быстродействие компьютера определяется тактовой частотой его процессора. Тактовая частота — количество тактов (соответственно и исполняемых команд) за секунду.
- Частота нынешних процессоров измеряется в ГГц (Гигагерцы). 1 ГГц = 10^9 Гц — миллиард операций в секунду.
- Чтобы уменьшить время выполнения программы, нужно либо оптимизировать (уменьшить) её, либо увеличить тактовую частоту. У части процессоров есть возможность увеличить частоту (разогнать процессор), однако такие действия физически влияют на процессор и нередко вызывают перегрев и выход из строя.





- Тактовые сигналы создают ещё одну сложность при проектировании процессора: поскольку их частоты постоянно растут, то на работу начинают влиять законы физики. Даже несмотря на чрезвычайно высокую скорость света, она недостаточно велика для высокопроизводительных процессоров. Если подключить тактовый сигнал к одному концу чипа, то ко времени, когда сигнал достигнет другого конца, он будет рассинхронизован на значительную величину. Чтобы синхронизировать все части чипа, тактовый сигнал распределяется при помощи так называемого H-Tree. Это структура, гарантирующая, что все конечные точки находятся на совершенно одинаковом расстоянии от центра.



- Может показаться, что проектирование каждого отдельного транзистора, тактового сигнала и контакта питания в чипе — чрезвычайно монотонная и сложная задача, и это в самом деле так. Даже несмотря на то, что в таких компаниях, как Intel, Qualcomm и AMD, работают тысячи инженеров, они не смогли бы вручную спроектировать каждый аспект чипа. Для проектирования чипов такого масштаба они используют множество сложных инструментов, автоматически генерирующих конструкции и электрические схемы. Такие инструменты обычно получают высокоуровневое описание того, что должен делать компонент, и определяют наилучшую аппаратную конфигурацию, удовлетворяющую этим требованиям. Недавно возникло направление развития под названием *High Level Synthesis*, которое позволяет разработчикам указывать необходимую функциональность в коде, после чего компьютеры определяют, как оптимальнее достичь её в оборудовании.
- Точно так же, как вы можете описывать компьютерные программы через код, проектировщики могут описывать кодом аппаратные устройства. Такие языки, как Verilog и VHDL позволяют проектировщикам оборудования выражать функциональность любой создаваемой ими электрической схемы. После выполнения симуляций и верификации таких проектов их можно синтезировать в конкретные транзисторы, из которых будет состоять электрическая схема. Хоть этап верификации может и не кажется таким увлекательным, как проектирование нового кэша или ядра, он значительно важнее их. На каждого нанимаемого компанией инженера-проектировщика может приходиться пять или более инженеров по верификации.

- Такие компании, как Intel, AMD и Nvidia, не публикуют схем работы своих процессоров, поэтому невозможно показать подобных полных электрических схем для современных процессоров. Однако этот простой сумматор позволит вам получить представление о том, что даже самые сложные части процессора можно разбить на логические и запоминающие элементы, а затем и на транзисторы. Теперь, когда мы знаем, как производятся некоторые компоненты процессора, нам нужно разобраться, как соединить всё вместе и синхронизировать. Все ключевые компоненты процессора подключены к *синхронизирующему (тактовому) сигналу (clock signal)*. Он попеременно имеет высокое и низкое напряжение, меняя его с заданным интервалом, называемым *частотой (frequency)*. Логика внутри процессора обычно переключает значения и выполняет вычисления, когда синхронизирующий сигнал меняет напряжение с низкого на высокое. Синхронизируя все части, мы можем гарантировать, что данные всегда поступают в правильное время, чтобы в процессоре не возникали «глюки».

  Вы могли слышать, что для повышения производительности процессора можно увеличить частоту тактовых сигналов. Это повышение производительности происходит благодаря тому, что переключение транзисторов и логики внутри процессора начинает происходить чаще, чем предусмотрено. Поскольку в секунду происходит больше циклов, то можно выполнить больше работы и процессор будет иметь повышенную производительность. Однако это справедливо до определённого предела. Современные процессоры обычно работают с частотой от 3,0 ГГц до 4,5 ГГц, и эта величина почти не изменилась за последние десять лет. Точно так же, как металлическая цепь не прочнее её самого слабого звена, процессор может работать не быстрее его самой медленной части. К концу каждого тактового цикла каждый элемент процессора должен завершить свою работу. Если какие-то части ещё её не завершили, то тактовый сигнал слишком быстрый и процессор не будет работать. Проектировщики называют эту самую медленную часть *критическим путём (Critical Path)* и именно он определяет максимальную частоту, с которой может работать процессор. Выше определённой частоты транзисторы просто не успевают достаточно быстро переключаться и начинают глючить или выдавать неверные выходные значения.

- Повысив напряжение питания процессора, мы можем ускорить переключение транзисторов, но это тоже срабатывает до определённого предела. Если подать слишком большое напряжение, то мы рискуем сжечь процессор. Когда мы повышаем частоту или напряжение процессора, он всегда начинает излучать больше тепла и потреблять бОльшую мощность. Так происходит потому, что мощность процессора прямо пропорциональна частоте и пропорциональна квадрату напряжения. Чтобы определить энергопотребление процессора, мы рассматриваем каждый транзистор как маленький конденсатор, который нужно заряжать или разряжать при изменении его значения.

  Подача питания — это настолько важная часть процессора, что в некоторых случаях до половины физических контактов на чипе может использоваться только для питания или заземления. Некоторые чипы при полной нагрузке могут потреблять больше 150 ампер, и со всем этим током нужно управляться чрезвычайно аккуратно. Для сравнения: центральный процессор генерирует больше тепла на единицу площади, чем ядерный реактор.

  Тактовый сигнал в современных процессорах отнимает примерно 30-40% от его общей мощности, потому что он очень сложен и должен управлять множеством различных устройств. Для сохранения энергии большинство процессоров с низким потреблением отключает части чипа, когда они не используются. Это можно реализовать отключением тактового сигнала (этот способ называется Clock Gating) или отключением питания (Power Gating).

- Верификация нового проекта часто занимает больше времени и денег, чем создание самого чипа. Компании тратят так много времени и средств на верификацию, потому что после отправки чипа в производство его невозможно исправить. В случае ошибки в ПО можно выпустить патч, но оборудование работает иначе. Например, компания Intel обнаружила баг в модуле деления с плавающей запятой некоторых чипов Pentium, и в результате это вылилось в потери, эквивалентные современным 2 миллиардам долларов.

  Сложно осмыслить то, что в одном чипе может быть несколько миллиардов транзисторов и понять, что все они делают. Если разбить чип на его отдельные внутренние компоненты, становится немного легче. Из транзисторов составляются логические элементы, логические элементы комбинируются в функциональные модули, выполняющие определённую задачу, а эти функциональные модули соединяются вместе, образуя архитектуру компьютера, которую мы обсуждали в первой части серии.

  БОльшая часть работ по проектированию автоматизирована, но изложенное выше позволяет нам осознать, насколько сложен только что купленный нами новый ЦП.

  Мы обсудили транзисторы, логические элементы, подачу питания и синхронизирующих сигналов, синтез конструкции и верификацию. В третьей части мы узнаем, что требуется для физического производства чипа. Все компании любят хвастаться тем, насколько современен их процесс изготовления (Intel — 10-нанометровый, Apple и AMD — 7-нанометровый, и т.д.), но что же на самом деле означают эти числа? Об этом мы расскажем в следующей части.

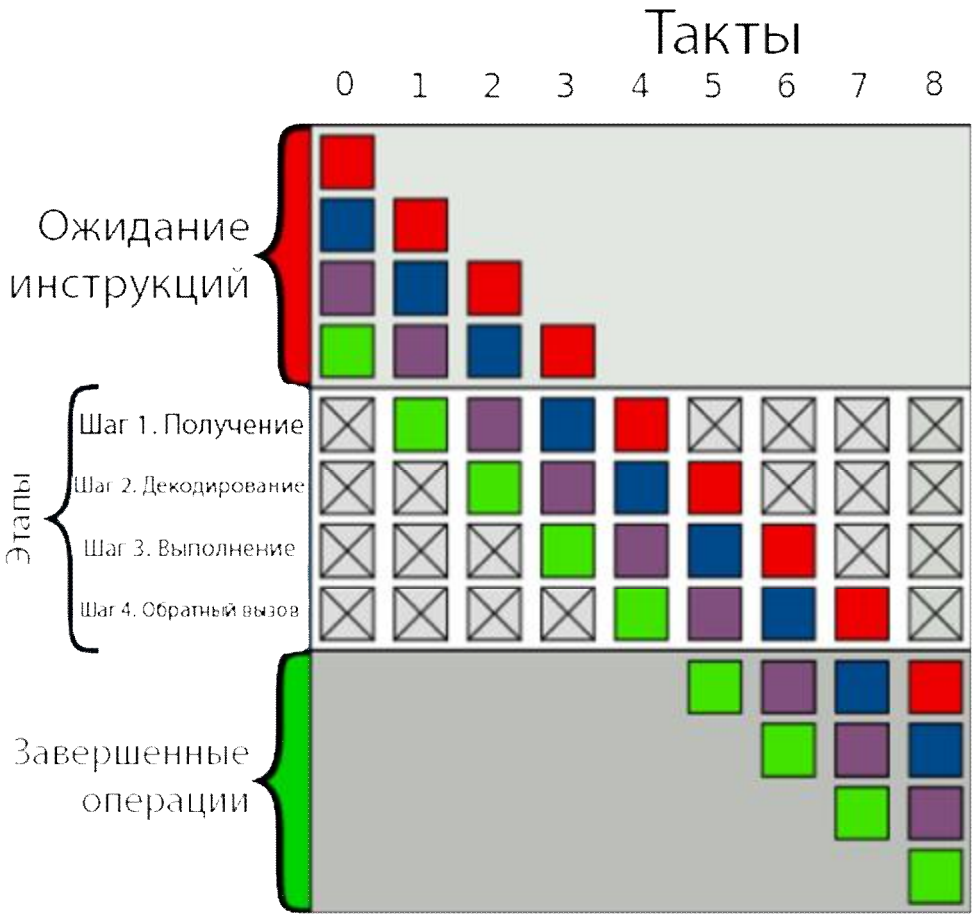
Выполнение инструкций

- Инструкции хранятся в ОЗУ в последовательном порядке. Для гипотетического процессора инструкция состоит из кода операции и адреса памяти/регистра. Внутри управляющего устройства есть два регистра инструкций, в которые загружается код команды и адрес текущей исполняемой команды.
- Т.Е., независимо от того, какой процессор использует наша система, все они читают двоичный код определенным образом, соответствующим своему семейству. Что они делают, так это берут определенное количество бит двоичного кода, который они выполняют, и интерпретируют их значение в соответствии с его расположением. Каждая инструкция кодируется следующим образом: первые цифры соответствуют коду инструкции и способу его выполнения, а последние биты - это сами данные или место, где находятся данные, на которых мы хотим выполнить инструкцию.
- Наборы регистров и инструкций ЦП называются ISA (Архитектура набора инструкций), и все в рамках одного ISA используют одинаковую кодировку инструкций и, следовательно, один и тот же двоичный код для них.
- Ещё в процессоре есть дополнительные регистры, которые хранят в себе последние 4 бита выполненных инструкций.
- Ниже рассмотрен пример набора команд, который суммирует два числа:
- LOAD_A 8. Это команда сохраняет в ОЗУ данные, скажем, <1100 1000>. Первые 4 бита — код операции. Именно он определяет инструкцию. Эти данные помещаются в регистры инструкций УУ. Команда декодируется в инструкцию load_A — поместить данные 1000 (последние 4 бита команды) в регистр А.
- LOAD_B 2. Ситуация, аналогичная прошлой. Здесь помещается число 2 (0010) в регистр В.
- ADD B A. Команда суммирует два числа (точнее прибавляет значение регистра В в регистр А). УУ сообщает АЛУ, что нужно выполнить операцию суммирования и поместить результат обратно в регистр А.
- STORE_A 23. Сохраняем значение регистра А в ячейку памяти с адресом 23.
- Вот такие операции нужны, чтобы сложить два числа.

Поток инструкций

- Современные процессоры могут параллельно обрабатывать несколько команд. Пока одна инструкция находится в стадии декодирования, процессор может успеть получить другую инструкцию.

Однако такое решение подходит только для тех инструкций, которые не зависят друг от друга.
Если процессор многоядерный, это означает, что фактически в нём находятся несколько отдельных процессоров с некоторыми общими ресурсами, например кэшем.



- **Связь набора инструкций с языком ассемблера**
- Все семейства процессоров имеют общий язык ассемблера, инструкции которого имеют соотношение 1: 1 с набором регистров и инструкций этого семейства процессоров. В приведенной выше таблице вы можете увидеть взаимосвязь между различными инструкциями языка ассемблера x86 и их кодом инструкций, который в таблице выражен в шестнадцатеричном формате.

Intel x86 Assembler Instruction Set Opcode Table

ADD Eb Gb *00*	**ADD** Ev Gv *01*	**ADD** Gb Eb *02*	**ADD** Gv Ev *03*	**ADD** AL Ib *04*	**ADD** eAX Iv *05*	**PUSH** ES *06*	**POP** ES *07*	**OR** Eb Gb *08*	**OR** Ev Gv *09*	**OR** Gb Eb *0A*	**OR** Gv Ev *0B*	**OR** AL Ib *0C*	**OR** eAX Iv *0D*	**PUSH** CS *0E*	**TWOBYTE** *0F*
ADC Eb Gb *10*	**ADC** Ev Gv *11*	**ADC** Gb Eb *12*	**ADC** Gv Ev *13*	**ADC** AL Ib *14*	**ADC** eAX Iv *15*	**PUSH** SS *16*	**POP** SS *17*	**SBB** Eb Gb *18*	**SBB** Ev Gv *19*	**SBB** Gb Eb *1A*	**SBB** Gv Ev *1B*	**SBB** AL Ib *1C*	**SBB** eAX Iv *1D*	**PUSH** DS *1E*	**POP** DS *1F*
AND Eb Gb *20*	**AND** Ev Gv *21*	**AND** Gb Eb *22*	**AND** Gv Ev *23*	**AND** AL Ib *24*	**AND** eAX Iv *25*	**ES:** *26*	**DAA** *27*	**SUB** Eb Gb *28*	**SUB** Ev Gv *29*	**SUB** Gb Eb *2A*	**SUB** Gv Ev *2B*	**SUB** AL Ib *2C*	**SUB** eAX Iv *2D*	**CS:** *2E*	**DAS** *2F*
XOR Eb Gb *30*	**XOR** Ev Gv *31*	**XOR** Gb Eb *32*	**XOR** Gv Ev *33*	**XOR** AL Ib *34*	**XOR** eAX Iv *35*	**SS:** *36*	**AAA** *37*	**CMP** Eb Gb *38*	**CMP** Ev Gv *39*	**CMP** Gb Eb *3A*	**CMP** Gv Ev *3B*	**CMP** AL Ib *3C*	**CMP** eAX Iv *3D*	**DS:** *3E*	**AAS** *3F*
INC eAX *40*	**INC** eCX *41*	**INC** eDX *42*	**INC** eBX *43*	**INC** eSP *44*	**INC** eBP *45*	**INC** eSI *46*	**INC** eDI *47*	**DEC** eAX *48*	**DEC** eCX *49*	**DEC** eDX *4A*	**DEC** eBX *4B*	**DEC** eSP *4C*	**DEC** eBP *4D*	**DEC** eSI *4E*	**DEC** eDI *4F*
PUSH eAX *50*	**PUSH** eCX *51*	**PUSH** eDX *52*	**PUSH** eBX *53*	**PUSH** eSP *54*	**PUSH** eBP *55*	**PUSH** eSI *56*	**PUSH** eDI *57*	**POP** eAX *58*	**POP** eCX *59*	**POP** eDX *5A*	**POP** eBX *5B*	**POP** eSP *5C*	**POP** eBP *5D*	**POP** eSI *5E*	**POP** eDI *5F*
PUSHA *60*	**POPA** *61*	**BOUND** Gv Ma *62*	**ARPL** Ew Gw *63*	**FS:** *64*	**GS:** *65*	**OPSIZE:** *66*	**ADSIZE:** *67*	**PUSH** Iv *68*	**IMUL** Gv Ev Iv *69*	**PUSH** Ib *6A*	**IMUL** Gv Ev Ib *6B*	**INSB** Yb DX *6C*	**INSW** Yz DX *6D*	**OUTSB** DX Xb *6E*	**OUTSW** DX Xv *6F*
JO Jb *70*	**JNO** Jb *71*	**JB** Jb *72*	**JNB** Jb *73*	**JZ** Jb *74*	**JNZ** Jb *75*	**JBE** Jb *76*	**JA** Jb *77*	**JS** Jb *78*	**JNS** Jb *79*	**JP** Jb *7A*	**JNP** Jb *7B*	**JL** Jb *7C*	**JNL** Jb *7D*	**JLE** Jb *7E*	**JNLE** Jb *7F*
ADD Eb Ib *80*	**ADD** Ev Iv *81*	**SUB** Eb Ib *82*	**SUB** Ev Ib *83*	**TEST** Eb Gb *84*	**TEST** Ev Gv *85*	**XCHG** Eb Gb *86*	**XCHG** Ev Gv *87*	**MOV** Eb Gb *88*	**MOV** Ev Gv *89*	**MOV** Gb Eb *8A*	**MOV** Gv Ev *8B*	**MOV** Ew Sw *8C*	**LEA** Gv M *8D*	**MOV** Sw Ew *8E*	**POP** Ev *8F*

- Имейте в виду, что в ISA постоянно добавляются новые инструкции, что приводит к появлению очень новых программ, которые явно используют эти новые инструкции, работают только на процессорах, которые их поддерживают. В общем, наборы инструкций стабильны во времени с небольшими изменениями, но время от времени вводятся инструкции для конкретных рынков, которые либо становятся частью стандарта, либо позже отбрасываются.
- Также есть случай, когда новые инструкции более эффективны, чем существующие, но в которых эти инструкции не исключаются из набора, потому что на рынке существует большое количество программного обеспечения, которое зависит от них.

RISC против CISC против Post-RISC

- У процессоров RISC мало инструкций, поэтому им нужно восполнить нехватку инструкций более сложными, но взамен они получают более высокую скорость при их выполнении из-за их легкости. С другой стороны, процессоры CISC имеют гораздо более сложные наборы инструкций, которые требуют более сложной конструкции оборудования, но вместо этого выполняют эти инструкции за меньшее количество циклов.
- Это различие, хотя и спорным в день, больше не так из-за того, что с момента появления из Pentium Pro на ПК мы пошли в эпоху пост-RISC, в котором, несмотря на то, что программы используют набор регистров и инструкций , они преобразуются в микрокод более простых инструкций в процессе, позволяя архитектурам CISC вести себя как архитектуры RISC и достигать высоких тактовых частот с использованием сложных инструкций.

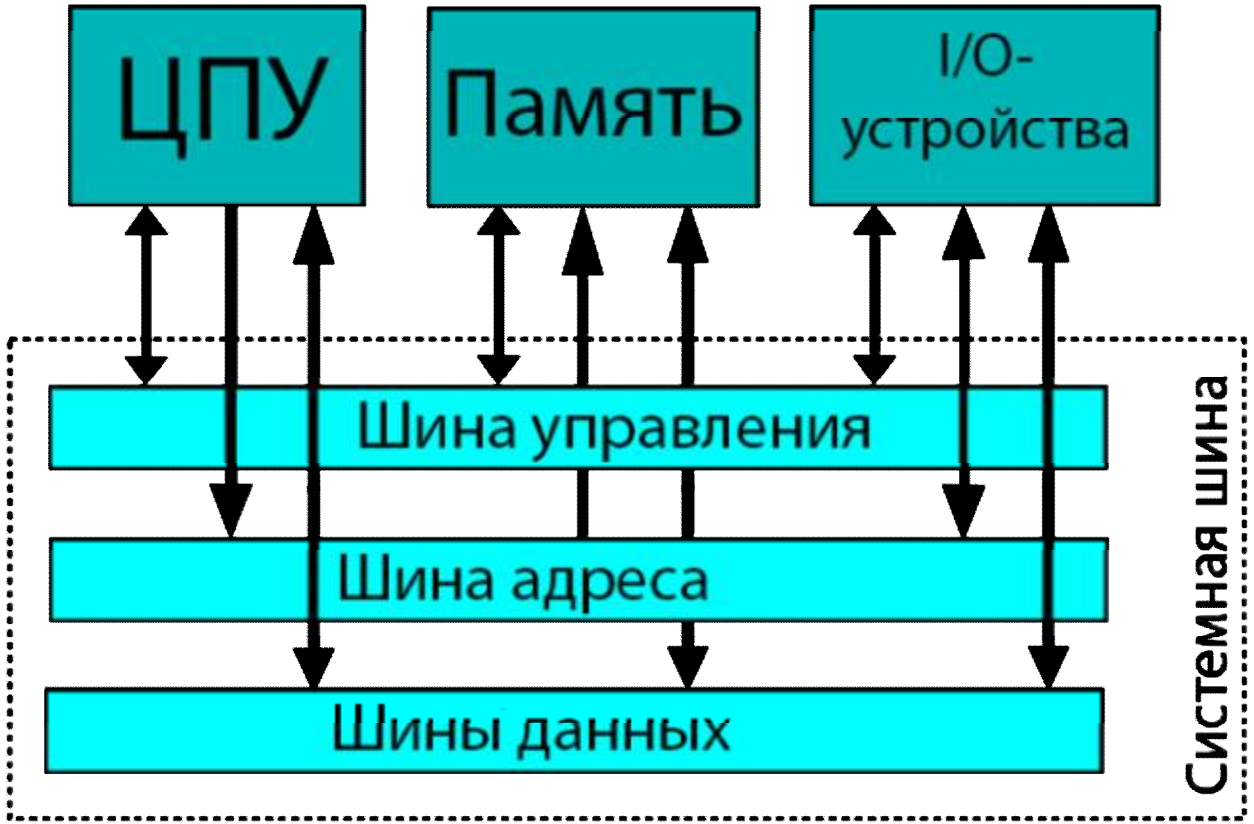
Регистры процессора

- Регистры - это память, ближайшая к существующему процессору и, следовательно, самая быстрая; Это очень маленькие блоки памяти, которыми можно управлять напрямую с помощью блока управления процессора. Они используются для выполнения всех видов общих задач, а не только для выполнения арифметических операций.
- Наиболее распространенные регистры в процессоре независимо от его ISA:
- **Регистры типа аккумулятора** : используется для арифметических операций. Каждое семейство имеет разное количество записей типа аккумулятора.
- **Регистры доступа к памяти** : содержат адрес памяти данных, к которым мы хотим получить доступ из ОЗУ.
- **Регистры данных в или из памяти** : Содержат данные, скопированные из памяти (чтение) или для записи по определенному адресу памяти (запись).
- **Регистры общего назначения** : это регистры памяти без специальной утилиты, которые служат для хранения данных, которые должны быть вызваны как можно быстрее.
- **Счетчик команд** : указывает следующую инструкцию для выполнения; Команды перехода изменяют их, когда вы хотите получить доступ не к следующей инструкции, а к другой части программы. В каждом полном командном цикле адрес памяти увеличивается на 1 и связывается с адресной шиной процессора.

- Некоторые из регистров ЦП, такие как регистр счетчика программ, который указывает, на какой следующий адрес памяти указывает процессор, находятся во всех ЦП и других типах процессоров с возможностью выполнения программ, в то время как другие записи уникальны для каждого набор записей и инструкций, делающий корреляцию 1: 1 между различными ISA практически невозможной.
- Даже если бы у нас был преобразователь кода инструкции 1: 1, у нас все равно были бы проблемы, потому что, хотя два процессора могут иметь одну и ту же инструкцию сложения, мы можем обнаружить, что способ использования регистров и регистров, которые они используют, различны и что есть даже регистры, которые есть в одной семье, а в других нет. Примером этих трудностей столкнулись оба **Microsoft** и Qualcomm при адаптации **Windows** 10 в ARM, чтобы все приложения x86 без проблем работали на процессоре ARM.
- Однако есть решения, такие как использование программного обеспечения для перевода инструкций. Указанное программное обеспечение переводит двоичный код в промежуточный код, а затем передает его в двоичный код целевого процессора, в котором мы хотим запустить приложение. Очевидно, что этот процесс намного медленнее, и рекомендуется запускать только очень старое программное обеспечение из семейств процессоров, которые не существуют на рынке.

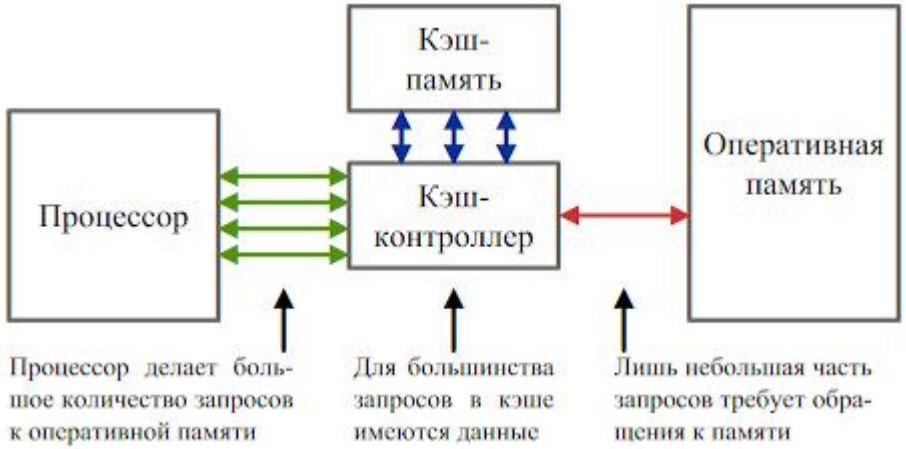
Шина

- Все данные между процессором, регистрами, памятью и I/O-устройствами (устройствами ввода-вывода) передаются по шинам. Чтобы загрузить в память только что обработанные данные, процессор помещает адрес в шину адреса и данные в шину данных. Потом нужно дать разрешение на запись на шине управления.

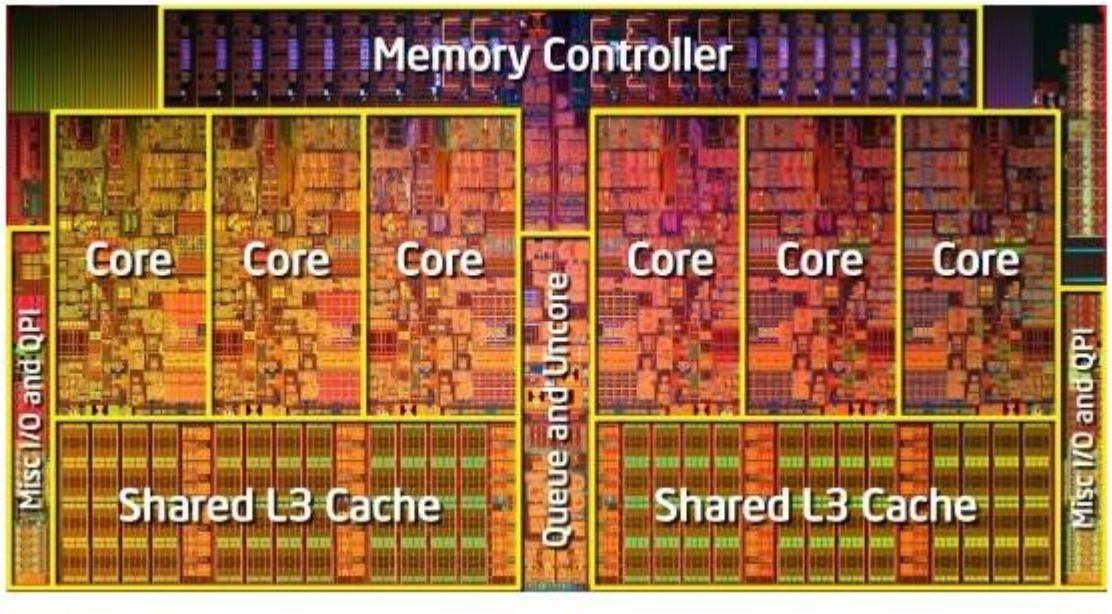


Кэш

- **Кэш процессора** — **кэш**, используемый **процессором** компьютера для уменьшения среднего времени доступа к компьютерной памяти. Является одним из верхних уровней иерархии памяти. **Кэш** использует небольшую, очень быструю память (обычно типа SRAM), которая хранит копии часто используемых данных из основной памяти. У процессора есть механизм сохранения инструкций в кэш. Как мы выяснили ранее, за секунду процессор может выполнить миллиарды инструкций. Поэтому если бы каждая инструкция хранилась в ОЗУ, то её изъятие оттуда занимало бы больше времени, чем её обработка. Поэтому для ускорения работы процессор хранит часть инструкций и данных в кэше.
- Если данные в кэше и памяти не совпадают, то они помечаются грязными битами (англ. dirty bit).



Как разрабатываются и производятся процессоры: основы архитектуры компьютеров

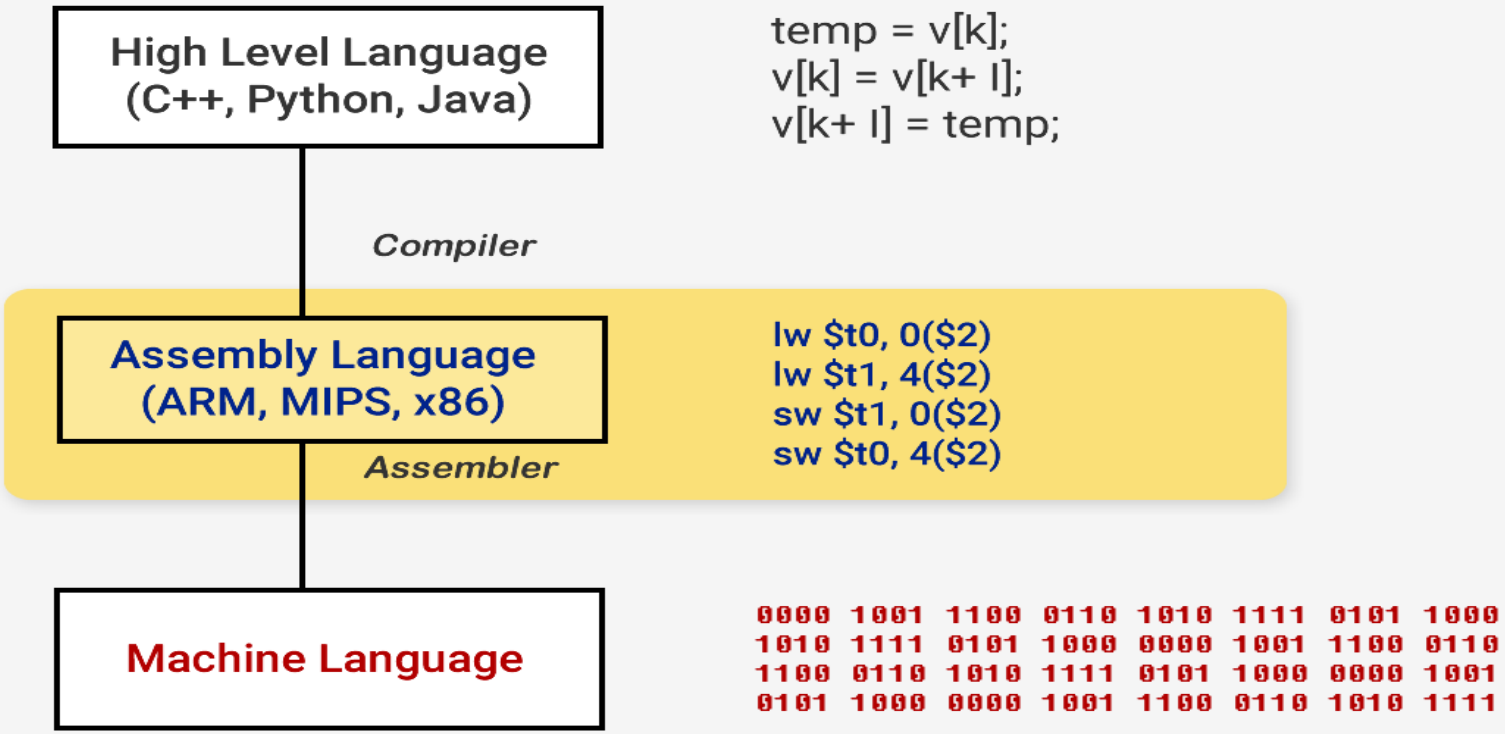


- Мы воспринимаем центральный процессор как «мозг» компьютера, но что это значит на самом деле? Что именно происходит внутри миллиардов транзисторов, благодаря которым работает компьютер? В нашей новой мини-серии из четырёх статей мы рассмотрим процесс создания архитектуры компьютерного оборудования и расскажем о принципах его работы.

- расскажем о компьютерной архитектуре, проектировании процессорных плат, VLSI (very-large-scale integration), производстве чипов и тенденциях будущего в области вычислительной техники. Если вам было интересно разобраться в подробностях работы процессоров, то начинать изучение лучше с этой серии статей.
- Мы начнём с очень высокоуровневого объяснения того, чем занимается процессор и как строительные блоки соединяются в функционирующую конструкцию. В том числе мы рассмотрим процессорные ядра, иерархию памяти, предсказание ветвлений и другое. Во-первых, нам нужно дать простое определение тому, что делает ЦП. Простейшее объяснение: процессор следует набору инструкций для выполнения определённой операции над множеством входящих данных. Например, это может быть считывание значения из памяти, затем прибавление его к другому значению, и наконец сохранение результата в память по другому адресу. Это может быть и нечто более сложное, например, деление двух чисел, если результат предыдущего вычисления больше нуля.

  Программы, например, операционная система или игра, сами по себе являются последовательностями инструкций, которые должен выполнять ЦП. Эти инструкции загружаются из памяти и в простом процессоре выполняются одна за другой, пока программа не завершится. Разработчики программного обеспечения пишут программы на высокоуровневых языках, например, на C++ или на Python, но процессор не может их понимать. Он понимает только единицы и нули, поэтому нам нужно каким-то образом представить код в этом формате.

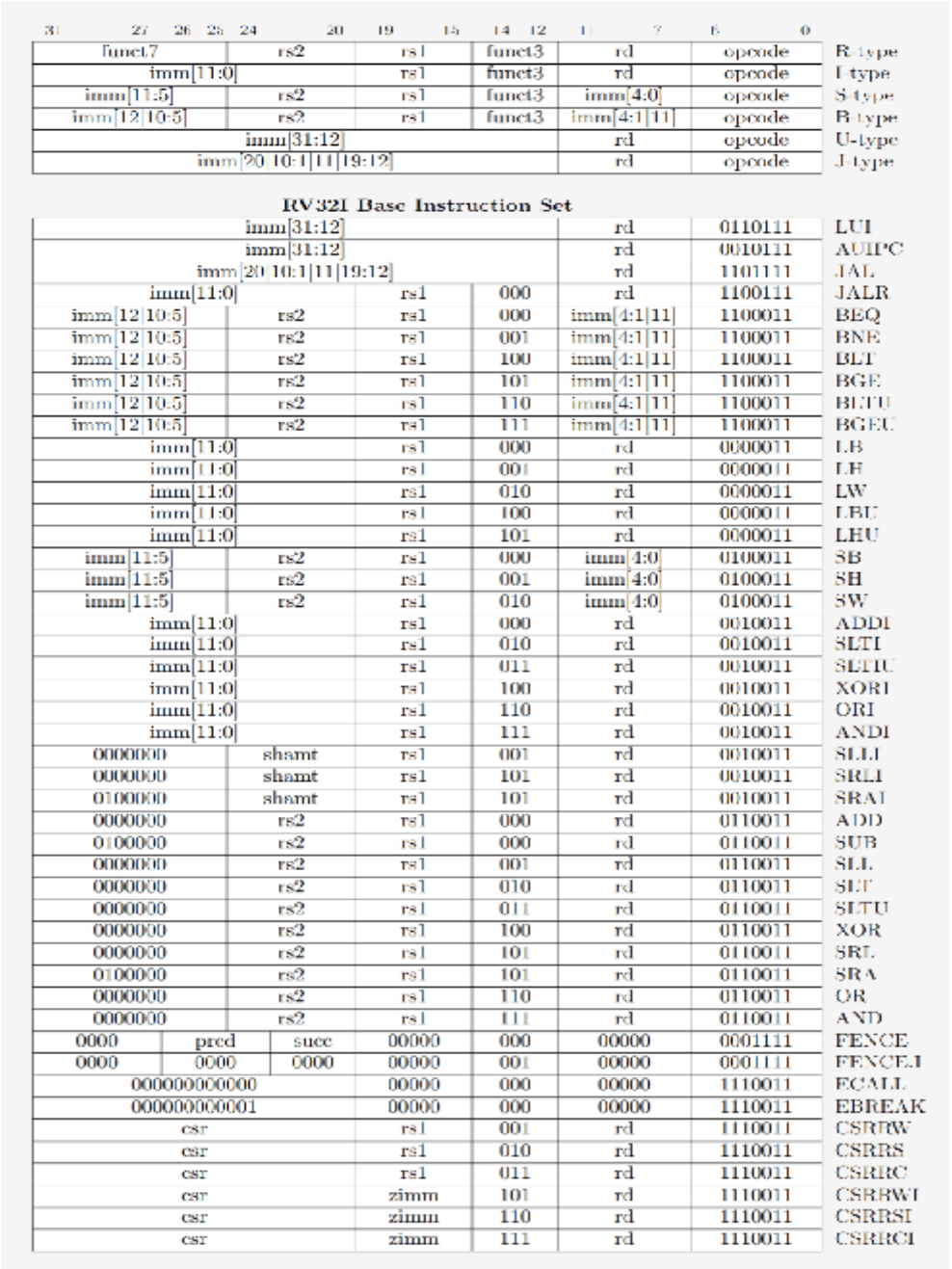
Levels of Program Code



- Программы компилируются в набор низкоуровневых инструкций, называемых *языком ассемблера*, который является частью архитектуры набора команд (Instruction Set Architecture, ISA). Это набор команд, которые должен понимать и выполнять ЦП. Одними из наиболее распространённых ISA являются x86, MIPS, ARM, RISC-V и PowerPC. Точно так же, как синтаксис написания функции на C++ отличается от функции, выполняющей то же действие в Python, у каждой ISA есть свой отличающийся синтаксис.

  Эти ISA можно разбить на две основных категории: с фиксированной и с переменной длиной. ISA RISC-V использует инструкции с фиксированной длиной, и это означает, что определённое заранее заданное количество битов в каждой инструкции определяет, какой тип имеет эта инструкция. В x86 всё иначе, в нём используются инструкции с переменной длиной. В x86 инструкции могут кодироваться различным способом с разным количеством битов для разных частей. Из-за такой сложности декодер инструкций в процессоре x86 обычно является самой сложной частью всего устройства.

  Инструкции с фиксированной длиной обеспечивают простое декодирование благодаря постоянной структуре, но ограничивают общее количество инструкций, которые могут поддерживаться ISA. В то время, как у популярных версий архитектуры RISC-V есть примерно 100 инструкций и все они имеют открытый исходный код, архитектура x86 проприетарна и никто не знает, сколько всего инструкций в ней есть. Обычно считается, что существует несколько тысяч инструкций x86, но точное число никто не публикует. Несмотря на различия между ISA, по сути все они имеют одинаковую базовую функциональность.



31 27	26 25	24 20	19 15	14 12	11 7	6 0	
funct7		rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]			rs1	funct3	rd	opcode	I-type
imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode	B-type
imm[31:12]					rd	opcode	U-type
imm[20|10:1|11|19:12]					rd	opcode	J-type

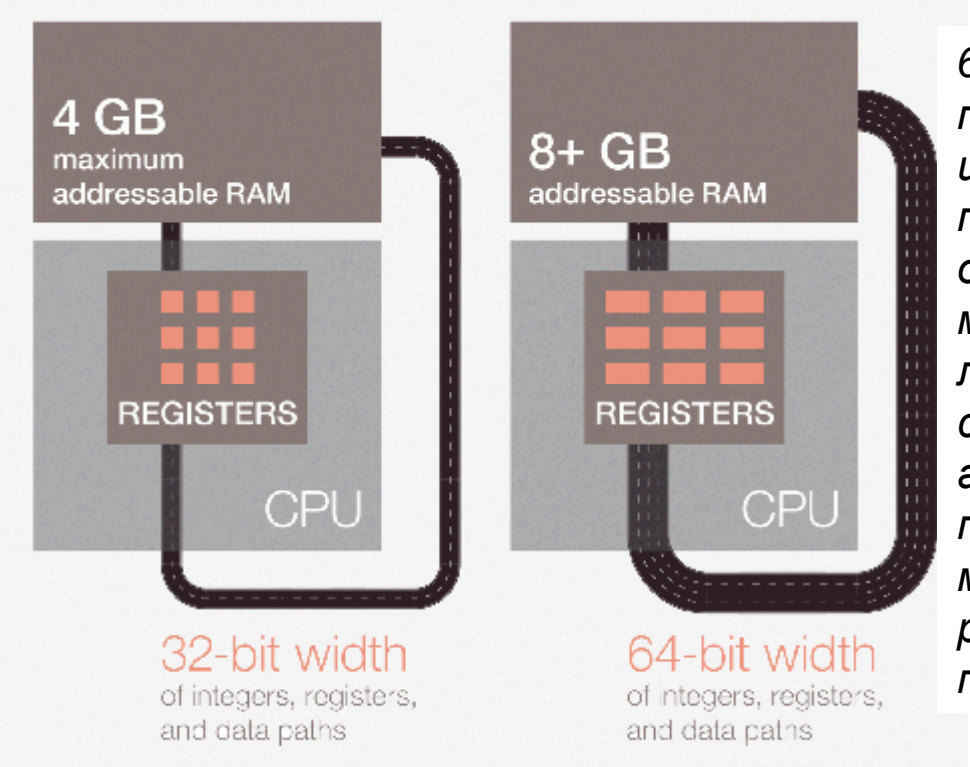
RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000	pred succ	00000	000	00000	0001111	FENCE
0000	0000 0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

- *Пример некоторых инструкций RISC-V. Опкод справа имеет длину 7 бит и определяет тип инструкции. Кроме того, каждая инструкция содержит биты, определяющие используемые регистры и выполняемые функции. Так ассемблерные инструкции разбиваются на двоичный код, чтобы его понимал процессор.*

- Теперь мы готовы включить компьютер и начать выполнять программы. Выполнение инструкции имеет несколько базовых частей, которые разбиты на множество этапов процессора.

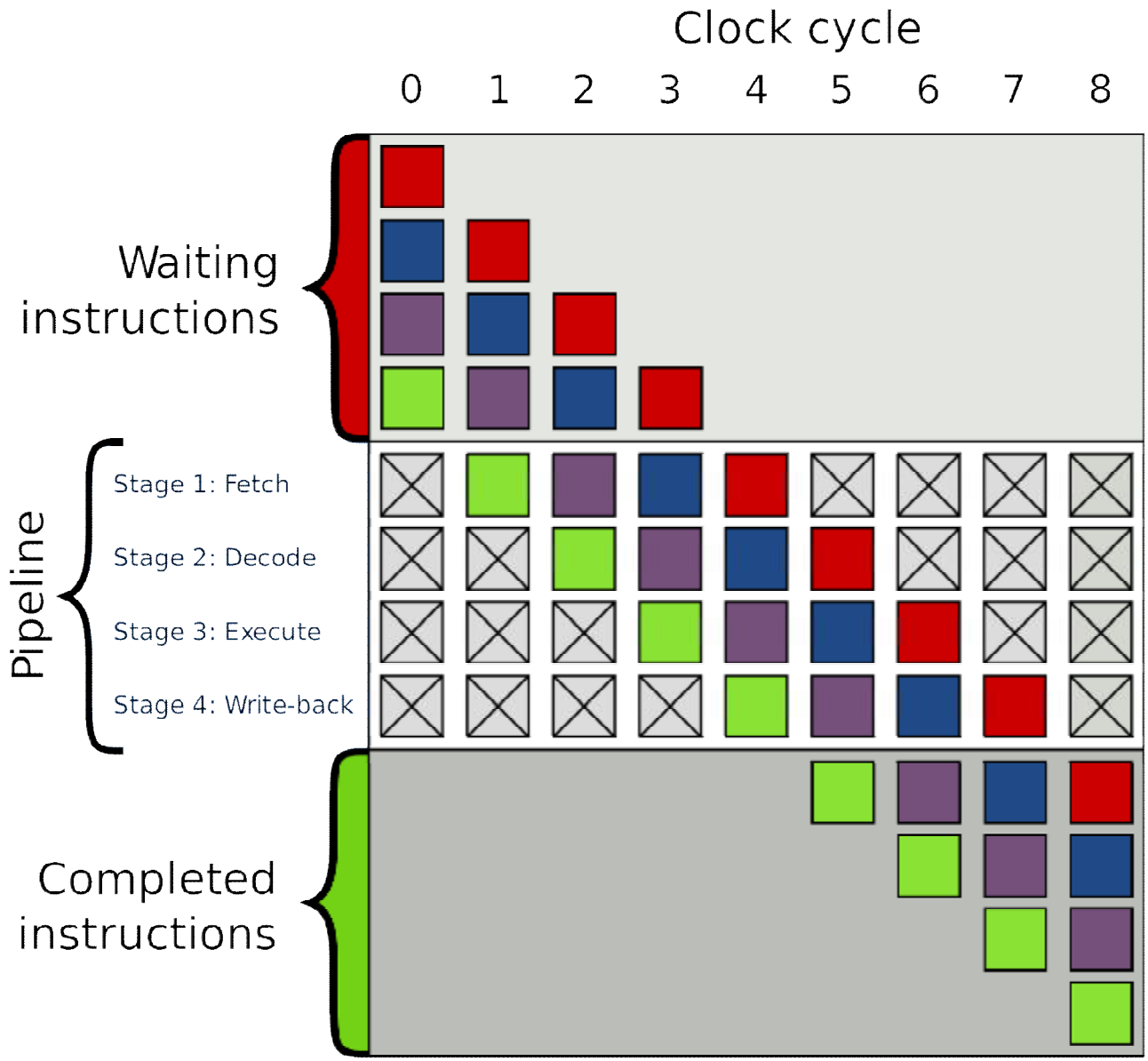
  Первый этап — передача инструкции из памяти в процессор для начала выполнения. На втором этапе инструкция декодируется, чтобы ЦП мог понять, какого типа эта инструкция. Существует множество типов, в том числе арифметические инструкции, инструкции ветвления и инструкции памяти. После того, как ЦП узнает, инструкцию какого типа он выполняет, операнды для инструкции берутся из памяти или внутренних регистров ЦП. Если вы хотите сложить число A и число B, то не можете выполнять сложение, пока не знаете значений A и B. Большинство современных процессоров являются 64-битными, то есть размер каждого значения данных составляет 64 бита.



64 бита — это ширина регистра процессора, канала передачи данных и/или адреса памяти. Для обычных пользователей это означает, какой объём информации компьютер может обработать за один раз, и лучше всего это понять в сравнении с младшим родственником по архитектуре — 32-битным процессором. 64-битная архитектура может обрабатывать за раз в два раза больше бит информации (64 бит против 32).

- Получив операнды для инструкции, процессор переносит их на этап выполнения, где производится операция над входящими данными. Это может быть сложение чисел, выполнение логических манипуляций с числами или просто передача чисел без их изменения. После вычисления результата может потребоваться доступ к памяти для его сохранения, или процессор может просто хранить значение в одном из своих внутренних регистров. После сохранения результата ЦП обновляет состояние различных элементов и переходит к следующей инструкции.

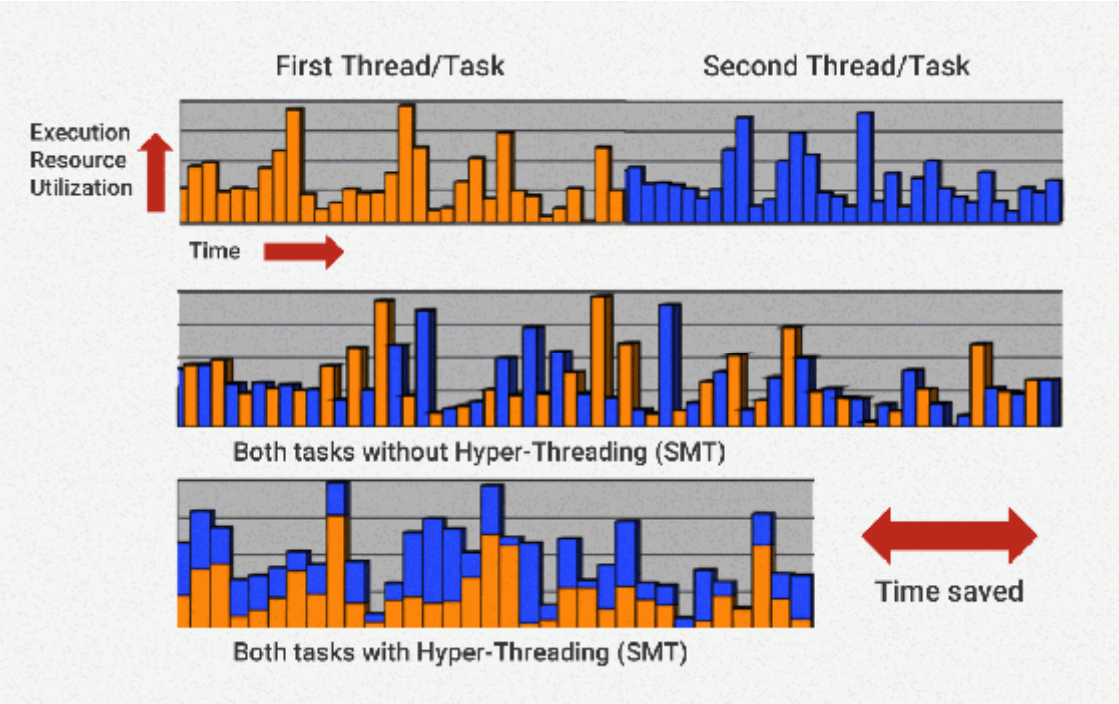
  Это объяснение, разумеется, сильно упрощено, и большинство современных процессоров для повышения эффективности разбивает эти несколько этапов на 20 или даже больше мелких этапов. Это означает, что хотя процессор начинает и завершает в каждом цикле несколько инструкций, может потребоваться 20 или больше циклов, чтобы выполнить одну инструкцию от начала до конца. Такая модель обычно называется pipeline («трубопровод», на русский обычно переводят как «конвейер»), потому что для заполнения трубопровода жидкостью и полного её прохождения требуется время, но после заполнения расход (вывод данных) будет постоянным.



Пример 4-этапного конвейера. Разноцветные прямоугольник и обозначают независящие друг от друга инструкции.

Весь проходимый инструкцией цикл — это очень тщательно скоординированный процесс, но не все инструкции могут завершаться одновременно. Например, сложение выполняется очень быстро, а деление или загрузка из памяти может занимать тысячи циклов. Вместо останова всего процессора до момента завершения одной медленной инструкции большинство современных процессоров выполняют их с изменением очерёдности. То есть они определяют, какую из инструкций выгоднее всего выполнить в текущий момент и буферизируют другие инструкции, которые пока не готовы. Если текущая инструкция ещё не готова, то процессор может перепрыгнуть вперёд по коду, чтобы посмотреть, готово ли что-то ещё.

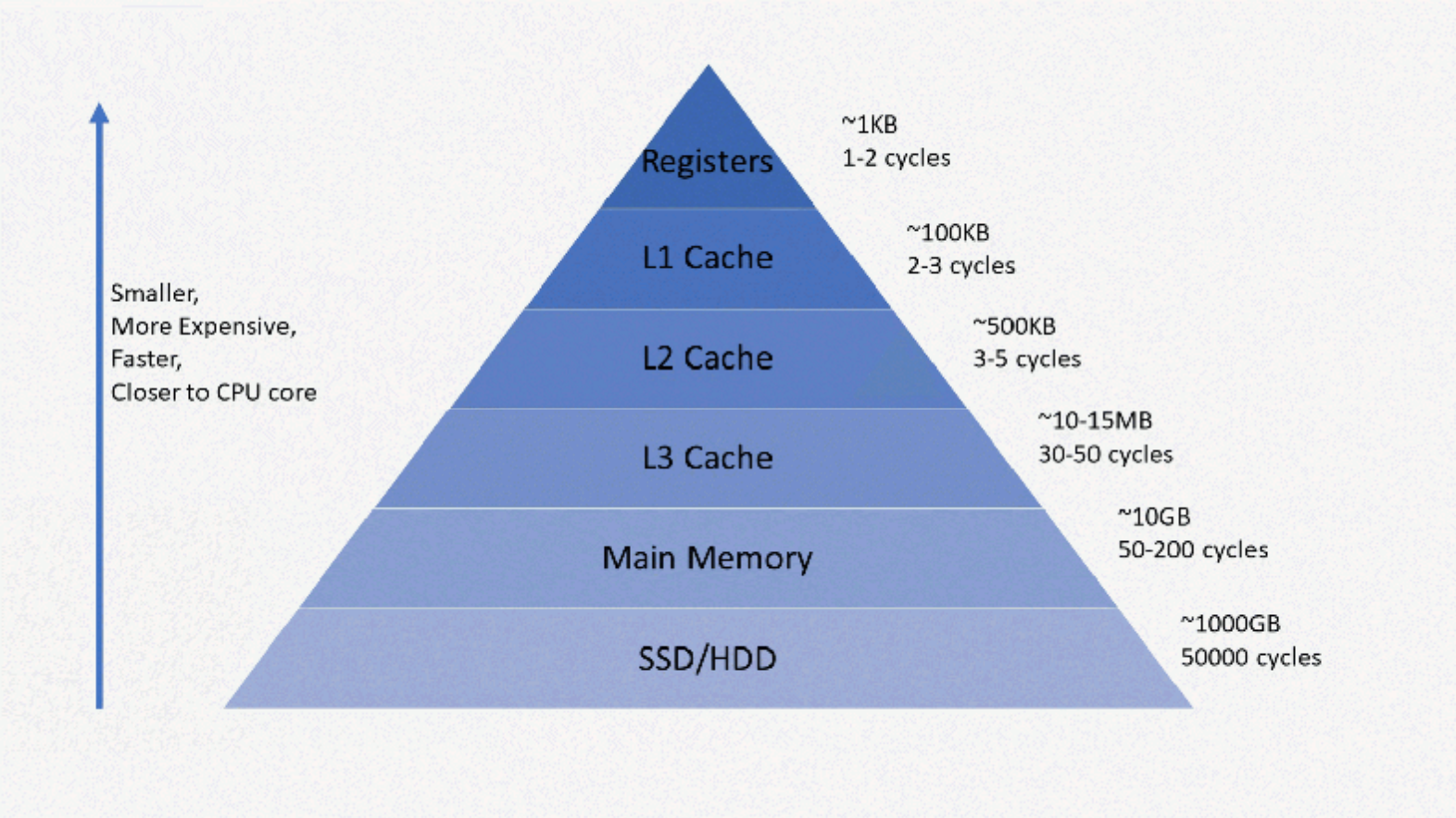
- Кроме выполнения с изменением очерёдности современные процессоры применяют технологию под названием ***суперскалярная архитектура***. Это означает, что в любой момент времени процессор одновременно выполняет на каждом этапе конвейера множество инструкций. Он может также ожидать ещё сотни других, чтобы начать их выполнение, и для того, чтобы иметь возможность одновременного выполнения нескольких инструкций внутри процессоров есть несколько копий каждого этапа конвейера. Если процессор видит, что к выполнению готовы две инструкции, и между ними нет зависимости, то он не ждёт, пока они завершатся по отдельности, а выполняет их одновременно. Одна из популярных реализаций такой архитектуры называется Simultaneous Multithreading (SMT) и также известна, как Hyper-Threading. Процессоры Intel и AMD сейчас поддерживают двухсторонний SMT, а IBM разработала чипы, поддерживающие до восьми SMT.



- Для завершения этого тщательно скоординированного выполнения процессор кроме базового ядра имеет множество дополнительных элементов. В процессоре есть сотни отдельных модулей, у каждого из которых есть специфическая функция, но мы рассмотрим только основы. Самыми важными и выгодными являются кэши и предсказатель переходов. Есть и другие дополнительные структуры, которые мы рассматривать не будем: буферы переупорядочивания, таблицы переименования регистров и станции резервирования.
- Необходимость кэшей иногда может сбивать с толку, ведь они хранят данные, как и ОЗУ или SSD. Но кэши отличаются задержкой и скоростью доступа. Даже несмотря на то, что память ОЗУ чрезвычайно быстра, она на порядки величин медленнее, чем нужно для ЦП. Для ответа с передачей данных ОЗУ может потребоваться сотни циклов, и процессору в это время будет нечем заняться. А если данных нет в ОЗУ, то могут потребоваться десятки тысяч циклов для получения доступа к ним с SSD. Без кэшей процессоры бы постоянно стопорились.

  Обычно процессоры имеют три уровня кэша, образующих так называемую *иерархию памяти*. Кэш L1 — самый маленький и быстрый, L2 находится посередине, а L3 — самый крупный и медленный из всех кэшей. Выше кэшей в иерархии находятся мелкие регистры, хранящие во время вычислений единственное значение данных. По порядку величин эти регистры являются самыми быстрыми устройствами хранения в системе. Когда компилятор преобразует высокоуровневую программу в язык ассемблера, он определяет наилучший способ использования этих регистров.

- Когда ЦП запрашивает данные из памяти, то сначала проверяет, хранятся ли эти данные уже в кэше L1. Если да, то можно всего за пару циклов получить к ним доступ. Если их там нет, то процессор проверяет L2, а затем и кэш L3. Кэши реализованы таким образом, что в общем случае они прозрачны для ядра. Ядро просто запрашивает данные по указанному адресу памяти, и тот уровень в иерархии, на котором они есть, отвечает ему. При переходе к последующим уровням в иерархии памяти размер и задержки обычно растут на порядки величин. В конце концов, если ЦП не находит данные ни в одном из кэшей, то обращается в основную память (ОЗУ).



- В обычном процессоре каждое ядро имеет два кэша L1: один для данных и другой для инструкций. Кэши L1 обычно имеют в целом объём порядка 100 килобайт и размер очень варьируется в зависимости от чипа и поколения процессора. Кроме того, обычно для каждого ядра есть свой кэш L2, хотя в некоторых архитектурах он может быть общим для двух ядер. Кэши L2 обычно имеют размер несколько сотен килобайт. Наконец, есть единственный кэш L3, общий для всех ядер, имеющий размер порядка десятков мегабайт.

  Когда процессор выполняет код, самые часто используемые инструкции и значения данных кэшируются. Это значительно ускоряет выполнение, потому что процессору не нужно постоянно обращаться за нужными данными в основную память. Во второй и третьей частях серии мы подробнее поговорим о том, как реализованы эти системы памяти.

  Кроме кэшей одним из самых важных строительных блоков современного процессора является точный *предсказатель переходов*. Инструкции переходов (ветвлений) схожи с конструкциями «if» для процессора. Один набор инструкций выполняется, если условие истинно, а другой — если оно ложно. Например, нам нужно сравнить два числа, и если они равны, выполнить одну функцию, а если не равны, то выполнить другую. Эти инструкции ветвления применяются чрезвычайно часто и могут составлять примерно 20% всех инструкций в программе.

- На первый взгляд кажется, что эти инструкции ветвления не должны вызывать проблем, но их правильное выполнение может оказаться очень сложным для процессора. В любой момент времени процессор может находиться в процессе одновременного выполнения десяти или двадцати инструкций, поэтому очень важно знать, *какие* инструкции выполнять. Может потребоваться 5 циклов, чтобы определить, что текущая инструкция — это переход и ещё 10 циклов, чтобы определить истинность условия. В это время процессор уже может начать выполнение десятков дополнительных инструкций, даже не зная, действительно ли это подходящие для выполнения инструкции.

  Чтобы обойти эту проблему, все современные высокопроизводительные процессоры используют методику под названием «упреждение» (speculation). Это означает, что процессор отслеживает инструкции ветвления и гадает, будет ли выполнен условный переход, или нет. Если предсказание верно, то процессор уже начал выполнять последующие инструкции, и это обеспечивает рост производительности. Если предсказание неверно, то процессор останавливает выполнение, удаляет все неверные инструкции, которые он начал выполнять, и начинает заново с правильной точки.

  Такие предсказатели перехода — одни из самых простейших разновидностей машинного обучения, потому что предсказатель изучает поведение ветвей в процессе выполнения. Если он предсказывает неверно слишком часто, то начинает обучаться правильному поведению. Десятилетия исследований методик предсказания переходов привели к тому, что в современных процессорах точность предсказаний превышает 90%.

- Хотя упреждение обеспечивает огромный рост производительности, потому что процессор может выполнять инструкции, которые уже готовы, вместо того, чтобы ожидать в очереди завершения выполняемых, оно в то же время создаёт уязвимости в защите. Знаменитая атака Spectre эксплуатирует баги в предсказании и упреждении переходов. Атакующий использует специально подобранный код, чтобы заставить процессор упреждающе выполнить код, благодаря чему происходит утечка значений из памяти. Для предотвращения утечки данных необходимо было переделать конструкцию отдельных аспектов упреждения, что привело к небольшому падению производительности.

  За последние десятилетия используемая в современных процессорах архитектура прошла долгий путь. Инновации и разработка продуманной структуры привели к повышению производительности и более оптимальному использованию аппаратных средств. Однако разработчики центральных процессоров тщательно хранят секреты их технологий, поэтому мы не можем точно узнать, что происходит у них внутри. Тем не менее, фундаментальные принципы работы процессоров стандартированы для всех архитектур и моделей. Intel может добавлять свои секретные ингредиенты, чтобы повысить долю попаданий кэша, а AMD может добавить улучшенный предсказатель переходов, но процессоры обеих компаний выполняют одинаковую задачу.

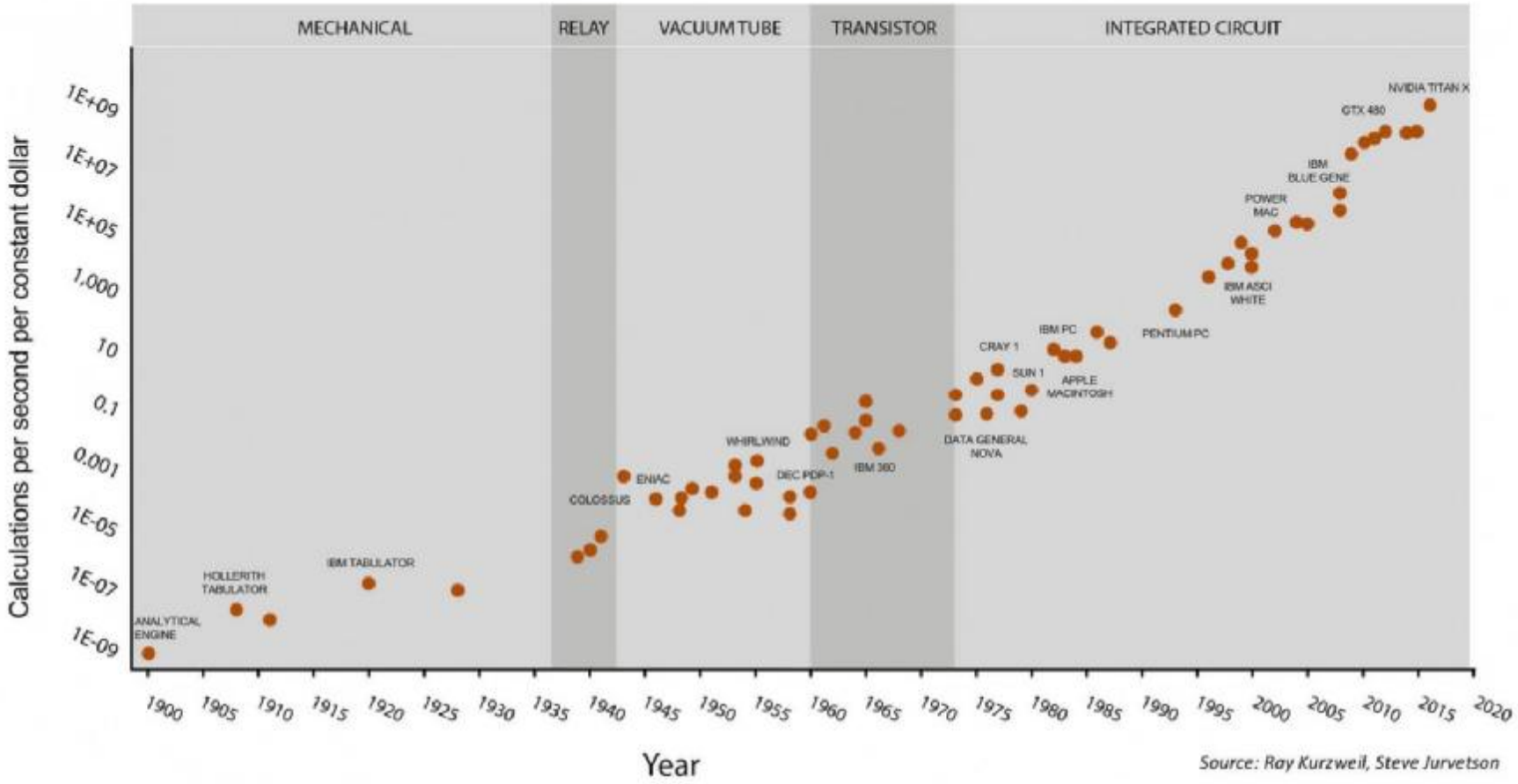
перспективы

- Несмотря на постоянные усовершенствования и постепенный прогресс в каждом новом поколении, в индустрии процессоров уже долгое время не происходит фундаментальных изменений. Огромным шагом вперёд стал переход от вакуума к транзисторам, а также переход от отдельных компонентов к интегральным схемам. Однако после них серьёзных сдвигов парадигмы такого же масштаба не происходило.

  Да, транзисторы стали меньше, чипы — быстрее, а производительность повысилась в сотни раз, но мы начинаем наблюдать стагнацию…

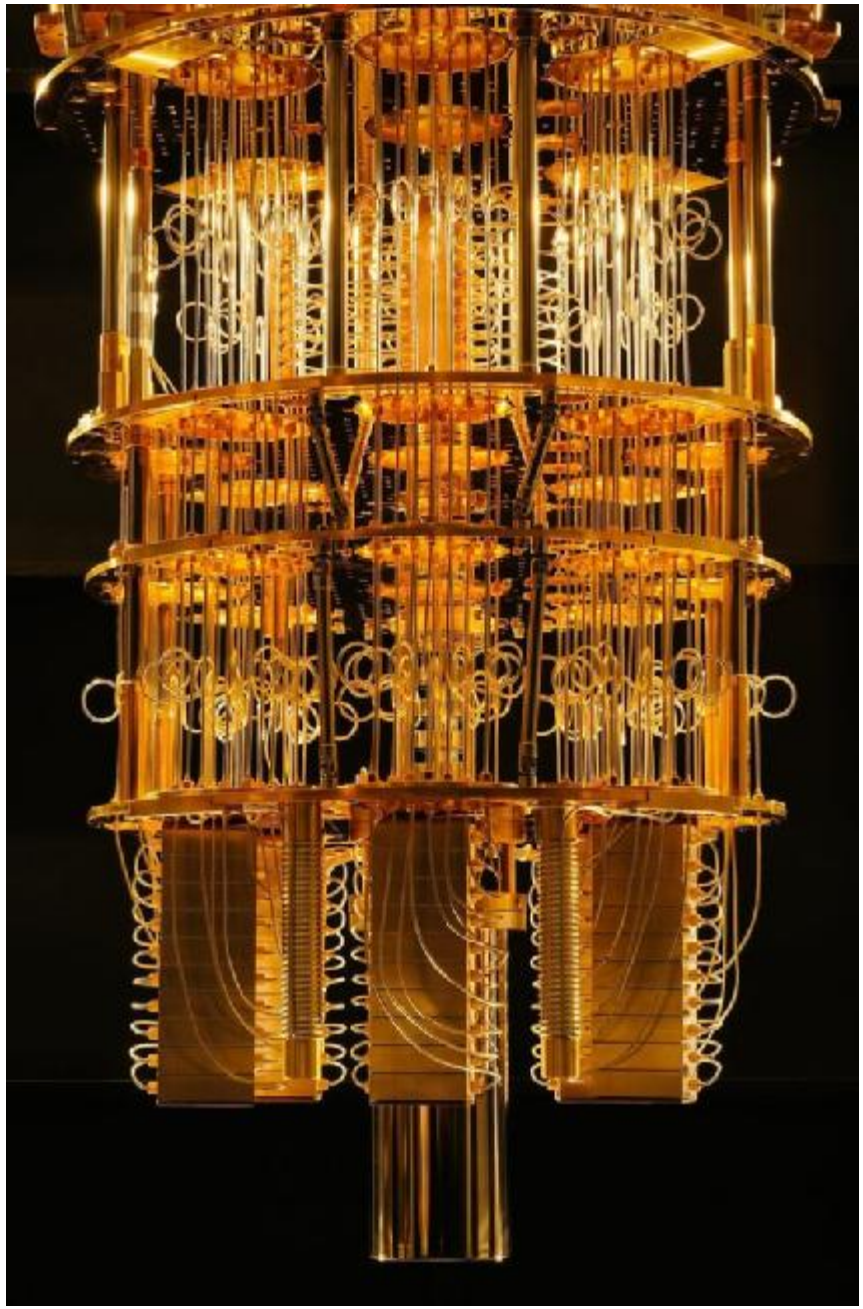
  Это четвёртая и последняя часть серии статей о разработке ЦП, рассказывающей о проектировании и изготовлении процессоров. Начав с высокого уровня, мы узнали о том, как компьютерный код компилируется в язык ассемблера, а затем в двоичные инструкции, которые интерпретирует ЦП. Мы обсудили то, как проектируется архитектура процессоров и они обрабатывают инструкции. Затем мы рассмотрели различные структуры, из которых составлен процессор.

- Компании-разработчики не делятся с общественностью своими исследованиями или подробностями современных технологий, поэтому нам сложно чётко представить, что именно находится внутри ЦП компьютера. Однако мы можем взглянуть на современные исследования и узнать, в каком направлении движется отрасль.
  Одним из знаменитых образов индустрии процессоров является закон Мура. Он гласит, что количество транзисторов в чипе удваивается каждые 18 месяцев. Долгое время это эмпирическое правило было справедливым, но рост начинает замедляться. Транзисторы становятся такими крошечными, что мы начинаем приближаться к пределу физически достижимых размеров. Без революционной новой технологии нам придётся в будущем исследовать другие возможности



- Из этого разбора следует один вывод: для повышения производительности компании начали увеличивать вместо частоты количество ядер. По этой причине мы наблюдаем, как широкое распространение получают восьмиядерные процессоры, а не двухядерные процессоры с частотой 10 ГГц. У нас просто осталось не так много пространства для роста, кроме как добавление новых ядер.

  С другой стороны, огромное пространство для будущего роста обещает область *квантовых вычислений*. Я не специалист, и поскольку её технологии по-прежнему разрабатываются, в этой области в любом случае пока мало реальных «специалистов». Чтобы развеять мифы, скажу, что квантовые вычисления не смогут обеспечить вам 1000 кадров в секунду в реалистичном рендере, или нечто подобное. Пока основное преимущество квантовых компьютеров заключается в том, что они позволяют использовать более сложные алгоритмы, ранее бывшие недостижимыми.
- *Один из прототипов квантовых компьютеров IBM*



- В традиционных компьютерах транзистор находится или во включенном, или в отключенном состоянии, что соответствует 0 или 1. В квантовом компьютере возможна *суперпозиция*, то есть бит одновременно может находиться в состоянии 0 и 1. Благодаря этой новой возможности учёные могут разрабатывать новые методы вычислений и у них появится возможность решать задачи, на которые у нас пока не хватает вычислительной мощности. Дело не столько в том, что квантовые компьютеры быстрее, а в том, что они являются новой моделью вычислений, которая позволит нам решать другие виды задач.

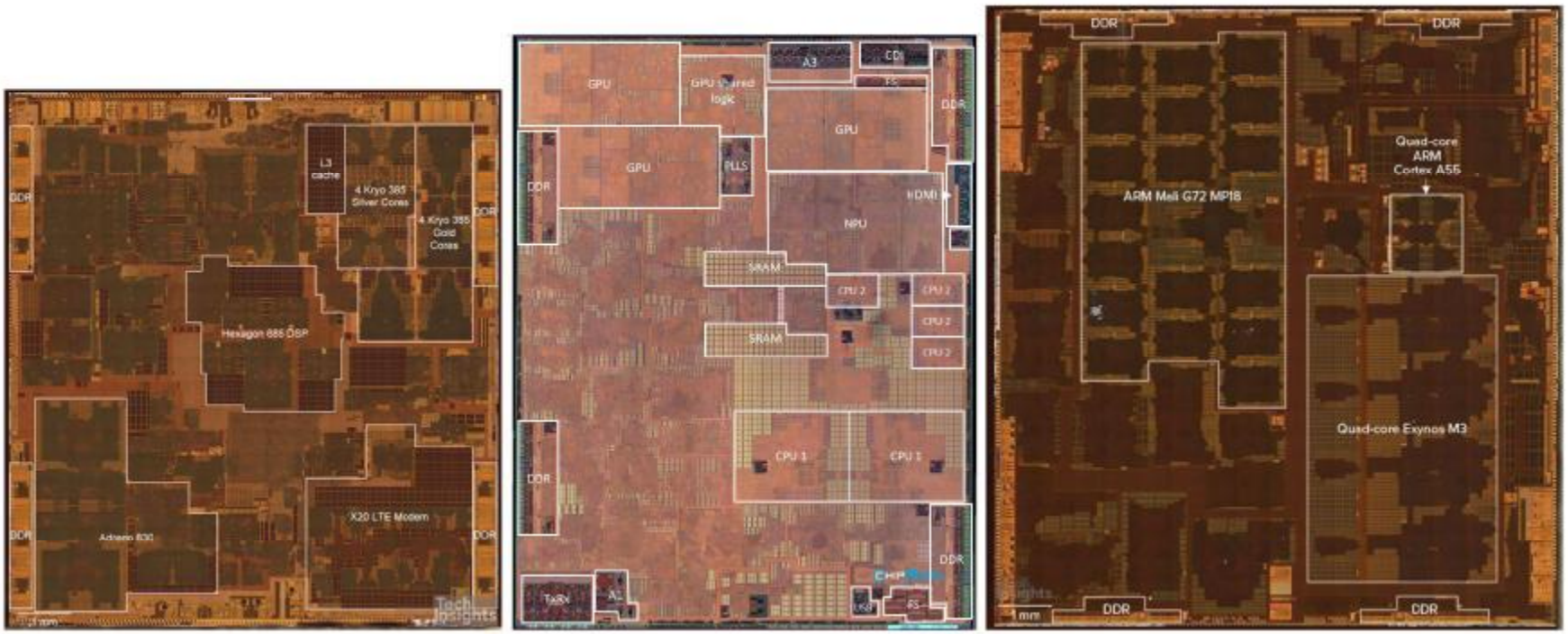
  До массового внедрения этой технологии осталось ещё одно-два десятилетия, поэтому какие же тенденции мы начинаем видеть в реальных процессора сегодня? Ведутся десятки активных исследований, но я коснусь только некоторых областей, которые, по моему мнению, окажут наибольшее влияние.

  Нарастает тенденция влияния *гетерогенных вычислений*. Эта методика заключается во включении в одну систему множества различных вычислительных элементов. Большинство из нас пользуется преимуществами такого подхода в виде отдельных GPU в компьютерах. Центральный процессор очень гибок и может с приличной скоростью выполнять широкий диапазон вычислительных задач. С другой стороны, GPU спроектированы специально для выполнения графических вычислений, например, перемножения матриц. Они очень хорошо с этим справляются и на порядки величин быстрее ЦП в подобных видах инструкций. Перенеся часть графических вычислений с ЦП на GPU, мы можем ускорить расчёты. Любой программист может оптимизировать ПО, изменив алгоритм, но оптимизировать оборудование гораздо сложнее.

- Но GPU — не единственная область, в которой акселераторы становятся всё популярнее. В большинстве смартфонов есть десятки аппаратных акселераторов, предназначенных для ускорения очень специфических задач. Такой стиль вычислений называется *морем акселераторов (Sea of Accelerators)*, его примерами могут быть криптографические процессоры, процессоры изображений, ускорители машинного обучения, кодеры/декодеры видео, биометрические процессоры и многое другое.

  Нагрузки становятся всё более и более специализированными, поэтому проектировщики включают в свои чипы всё больше акселераторов. Поставщики облачных услуг, например AWS, начали предоставлять разработчикам FPGA-карты для ускорения их вычислений в облаках. В отличие от традиционных вычислительных элементов наподобие ЦП и GPU, имеющих фиксированную внутреннюю архитектуру, FPGA гибки. Это почти программируемое оборудование, которое можно настраивать в соответствии с нуждами компании.

  Если кому-то нужно распознавание изображений, то он реализует эти алгоритмы в «железе». Если кто-то хочет симулировать работу новой аппаратной архитектуры, то перед изготовлением её можно протестировать на FPGA. FPGA обеспечивает бОльшую производительность и энергоэффективность, чем GPU, но всё равно меньше, чем у ASIC (application specific integrated circuit — интегральная схема специального назначения). Другие компании, например, Google и Nvidia, разрабатывают отдельные ASIC машинного обучения для ускорения распознавания и анализа изображений.



Snapdragon 845
(~90 mm^2)

A11 Bionic
(87.66 mm^2)

Exynos 9810
(118.94 mm^2)

- *Снимки кристаллов популярных мобильных процессоров, демонстрирующие их структуру.*

  Взглянув на снимки кристаллов относительно современных процессоров, можно увидеть, что бОльшую часть площади ЦП на самом деле занимает не само ядро. Всё бОльшую долю занимают различные акселераторы. Это позволило ускорить очень специализированные вычисления, а также значительно снизить энергопотребление.

- К упомянутым отдельным FPGA-акселераторам добавлю такую тему как встраиваемые FPGA (embedded FPGA, eFPGA). В этом случае на кристалл к процессору или SoC добавляется логика FPGA, причем заказчик может заранее указать требуемые соотношения и количество LUT/DSP/BRAM блоков в зависимости от своей специфики, а также добавить свои блоки. Такими вещами занимаются например Menta, Flex-Logic, Achronix.

- Раньше при необходимости добавления в систему обработки видео разработчикам приходилось устанавливать в неё новый чип. Однако это очень неэффективно с точки зрения энергопотребления. Каждый раз, когда сигналу нужно выходить из чипа по физическому проводнику к другому чипу, на бит требуется огромное количество энергии. Сама по себе крошечная доля джоуля не кажется особо большими тратами, но передача данных внутри, а не снаружи чипа может быть на 3-4 порядка величин эффективнее. Благодаря интеграции таких акселераторов с ЦП мы наблюдали в последнее время рост количества чипов с сверхнизким энергопотреблением.

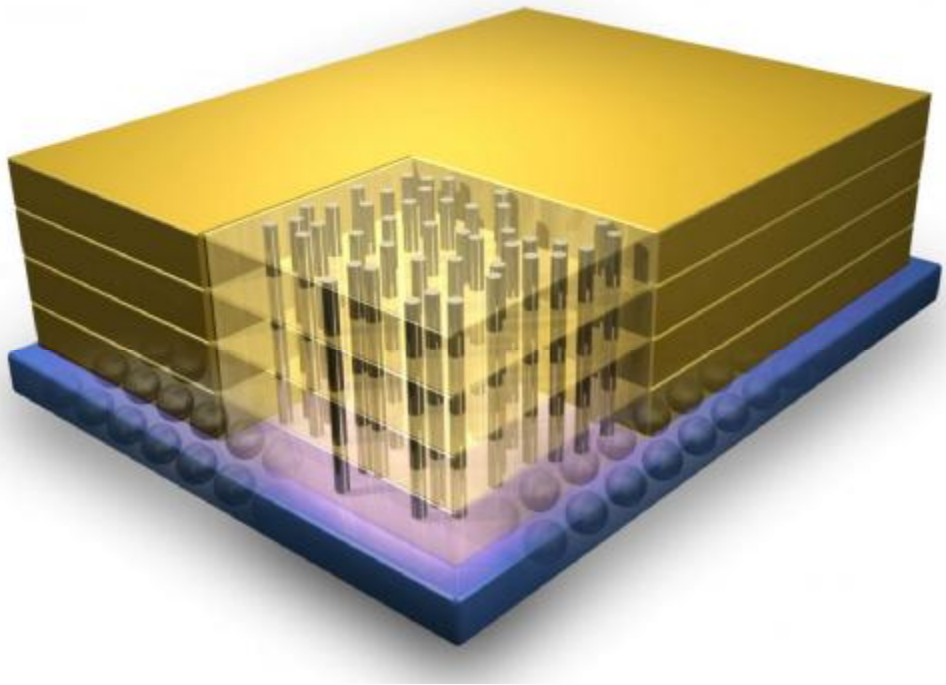
  Однако акселераторы не идеальны. Чем больше мы добавляем их в схемы, тем менее гибким становится чип и мы начинаем жертвовать общей производительностью в пользу пиковой производительности специализированных видов вычислений. На каком-то этапе весь чип просто превращается в набор акселераторов и перестаёт быть полезным ЦП. Баланс между производительностью специализированных вычислений и общей производительностью всегда очень тщательно настраивается. Это разногласие между оборудованием общего назначения и специализированными нагрузками называется *разрывом специализации (specialization gap)*.

- Хотя кое-кто считает. что мы находимся на пике пузыря GPU/Machine Learning, скорее всего стоит ожидать, что всё больший объём вычислений будет передаваться специализированным ускорителям. Облачные вычисления и ИИ продолжают развиваться, поэтому GPU выглядят лучшим решением для достижения уровня требуемых объёмных вычислений.

- Ещё одна область, в которой проектировщики ищут способы повышения производительности — это память. Традиционно считывание и запись значений всегда были одним из самых серьёзных «узких мест» процессоров. Нам могут помочь быстрые и большие кэши, но считывание из ОЗУ или с SSD может занимать десятки тысяч тактовых циклов. Поэтому инженеры часто рассматривают доступ к памяти как более затратный, чем сами вычисления. Если процессор хочет сложить два числа, то ему сначала нужно вычислить адреса памяти, по которым хранятся числа, выяснить, на каком уровне иерархии памяти есть эти данные, считать данные в регистры, выполнить вычисления, вычислить адрес приёмника и записать значение в нужное место. Для простых инструкций, выполнение которых может занимать один-два цикла, это чрезвычайно неэффективно.

  Новая идея, которую сейчас активно исследуют — это техника под названием *Near Memory Computing*. Вместо того, чтобы извлекать небольшие фрагменты данных из памяти и вычислять их быстрым процессором, исследователи переворачивают работу вниз головой. Они экспериментируют с созданием небольших процессоров непосредственно в контроллерах памяти ОЗУ или SSD. Благодаря тому, что вычисления становятся ближе к памяти, существует потенциал огромной экономии энергии и времени, ведь данные теперь не надо передавать так часто. Вычислительные модули имеют прямой доступ к нужным им данным, потому что находятся непосредственно в памяти. Эта идея всё ещё находится в зачаточном состоянии, но результаты выглядят многообещающе.

- Одно из препятствий, которое нужно преодолеть для near memory computing — это ограничения процесса изготовления. Как говорилось, процесс производства кремния очень сложен и в нём задействованы десятки этапов. Эти процессы обычно специализированы для изготовления или быстрых логических элементов, или плотно расположенных накопительных элементов. Если попытаться создать чип памяти с помощью оптимизированного для вычислений процесса изготовления, то получится чип с чрезвычайно низкой плотностью элементов. Если попробовать создать процессор с помощью процесса изготовления накопителей, то получим очень низкую производительность и большие тайминги.



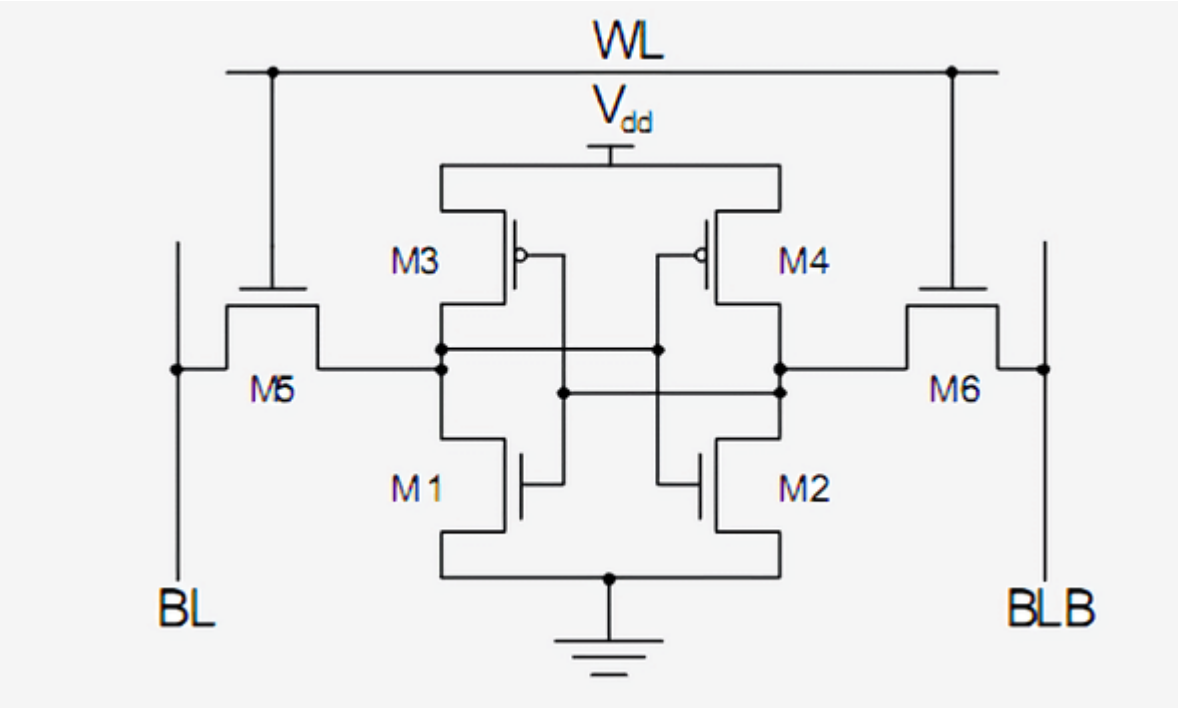
Пример 3D-интеграции, демонстрирующий вертикальные соединения между слоями транзисторов.

- Одно из потенциальных решений этой проблемы — *3D-интеграция*. Традиционные процессоры обладают одним очень широким слоем транзисторов, но это имеет свои ограничения. Как понятно из названия, трёхмерная интеграция — это процесс расположения нескольких слоёв транзисторов друг над другом для повышения плотности и снижения задержек. Вертикальные столбцы, производимые на разных процессах изготовления, могут быть затем использованы для соединений между слоями. Эта идея была предложена уже давно, но индустрия потеряла к ней интерес из-за серьёзных сложностей в её реализации. В последнее время мы наблюдаем возникновение технологии накопителей 3D NAND и возрождение этой области исследований.

  Кроме физических и архитектурных изменений, на всю полупроводниковую отрасль сильно повлияет ещё одна тенденция — больший упор на безопасность. До недавнего времени о безопасности процессоров думали чуть ли не в последний момент. Это похоже на то, как Интернет, электронная почта и многие другие системы, которые мы сегодня активно используем, разрабатывались почти без оглядки на безопасность. Все существующие меры защиты «прикручивались» по мере случавшихся инцидентов, чтобы мы чувствовали себя в безопасности. В области процессоров такая тактика больно ударила по компаниям, и особенно по Intel.

  Баги Spectre и Meltdown — это, вероятно, самые известные примеры того, как проектировщики добавляют функции, значительно ускоряющие процессор, не в полной мере осознавая связанные с этим угрозы безопасности. При разработке современных процессоров гораздо большее внимание уделяется безопасности как ключевой части архитектуры. При повышении безопасности часто страдает производительность, но учитывая ущерб, который могут нанести серьёзные баги безопасности, можно с уверенностью сказать, что лучше фокусироваться на безопасности в той же степени, что и на производительности.

- Секвенциальная логика строится аккуратным соединением инверторов и других логических элементов так, чтобы их выходы передавали сигналы обратной связи на вход элементов. Такие контуры обратной связи используются для хранения одного бита данных и называются *статическим ОЗУ (Static RAM)*, или SRAM. Эта память называется статическим ОЗУ в противовес динамической (DRAM), потому что сохраняемые данные всегда напрямую соединены с положительным напряжением или заземлением.
Стандартный способ реализации одного бита SRAM — это показанная ниже схема из 6 транзисторов. Самый верхний сигнал, помеченный как WL (*Word Line*) — это адрес, и когда он включен, то данные, хранящиеся в этой 1-битной ячейке передаются в *Bit Line*, помеченную как BL. Выход BLB называется *Bit Line Bar*; это просто инвертированное значение Bit Line. Вы должны узнать два типа транзисторов и понять, что M3 с M1, как и M4 с M2, образуют инвертор.



- SRAM используется для построения сверхбыстрых кэшей и регистров внутри процессоров. Эта память очень стабильна, но для хранения каждого бита данных требует от шести до восьми транзисторов. Поэтому по сравнению с DRAM она чрезвычайно затратна с точки зрения стоимости, сложности и площади на чипе. С другой стороны, Dynamic RAM хранит данные в крошечном конденсаторе, а не использует логические элементы. Она называется динамической, потому что напряжение на конденсаторе может значительно изменяться, так как он не подключён к питанию или заземлению. Есть только один транзистор, используемый для доступа к хранящимся в конденсаторе данным.

  Поскольку DRAM требует всего по одному транзистору на бит и очень масштабируема, её можно плотно и дёшево упаковывать. Недостаток DRAM заключается в том, что заряд конденсатора так мал, что его необходимо постоянно обновлять. Именно поэтому после отключения питания компьютера все конденсаторы разряжаются и данные в ОЗУ теряются.