1) Физические основы ЭВМ.

a) Если с точки зрения компьютера у нас два сигнала: 0 и 1, то в реальной жизни всё не так радужно. Для каждого логического элемента/схемы есть диапазон значений, который этот элемент трактует как 0, и соответствующий диапазон для 1. 0 - это от Vgnd (напряжение земли) до Vil (input low - нижнее входное значение). 1 - от Vih (input high - верхнее входное) до Vdd (напряжение источника). На выходе мы тоже получаем не абсолютные значения, а Vol(output low - нижнее выходное) в качестве 0 и Voh(output high - верхнее выходное) в качестве 1.

b) Внутри компьютера всё завязано на полупроводниках. Они бывают двух типов: p и n.

c) <неинтересная физика, могу обмануть>В n-типе у нас есть 4-валентный кремний, в который добавили 5-валентный мышьяк. “Лишний” электрон мышьяка не образует связей с кремнием и остаётся свободным. При комнатной температуре часть атомов мышьяка лишается электрона и становится положительными ионами, который, однако, не может захватить электрон у соседнего кремния, так как последний тупо сильнее держится за своё богатство. Поэтому дырки и течения электронов не возникает, в то время как оторвавшиеся электроны являются электронами проводимости.

d) P-тип, наоборот, образуется, когда мы к 4-валентному кремнию добавляем что-то 3-валентное (у Ромы на слайде алюминий). Тогда у нас образуются “дырки”, в которые природа хочет запихнуть электрончик.

e) Когда мы соединяем полупроводники n и p типов, на стыке образуется запирающий слой. Если мы теперь к этом подключим источник тока так, чтобы плюс источника подсоединился туда, где у нас много электронов, то электроны проводимости и “дырки” начнут удаляться друг от друга, а значит и от слоя, тем самым увеличивая толщину слоя и увеличивая сопротивление проводника. Если же теперь поменять полюсы источника местами, то “дырки” и электроны устремятся навстречу друг другу, уменьшая запирающий слой и сопротивление проводника.</неинтересная физика, могу обмануть>

f) Помимо транзисторов ещё есть конденсаторы, которые накапливают заряд тока. Выглядят, наверное, как две пластины, между которыми слой диэлектрика (обычно воздух). Но это неинтересно, вернёмся к нашим транзисторам.

g) Транзисторы раньше использовались биполярные, но они уже устарели. На смену пришли металл-оксид полупроводники (сокращённо на русском МОП, на английском - MOS). Транзисторы n-типа aka nMOS - являются подложкой p-типа, в которой расположены области n-типа. Транзисторы p-типа aka pMOS - наоборот. Между двумя областями: истоком (source) и стоком (drain) находится затвор (gate). Прикладывание напряжения к

затвору создаёт или убирает электрическое поле, которое обеспечивает переход электронов от истока к стоку через канал (channel).

h) Отличие nMOS от pMOS. nMOS не пропускает ток при отсутствии напряжения на затворе и начинает пропускать, если это напряжение появится. pMOS, наоборот, пропускает ток в обычном состоянии и говорит “ТЫ НЕ ПРОЙДЁШЬ”, если подвести ток.

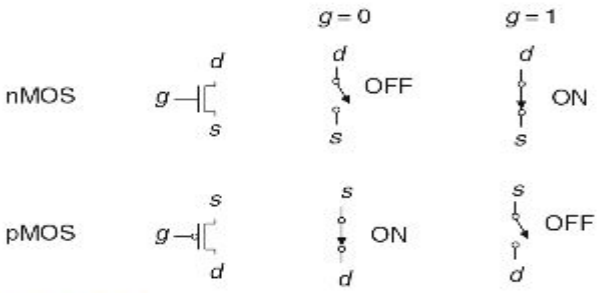


Рис. 1.31 Модели переключения полевых МОП-транзисторов

i) Из транзисторов собирают основные логические вентили: НЕ (NOT); И-НЕ, которое вообще-то НЕ-И (NAND); ИЛИ-НЕ, которое НЕ-ИЛИ (NOR). Схема НЕ: Схема И-НЕ: Схема ИЛИ-НЕ:

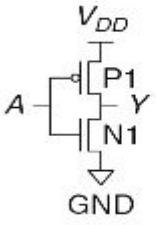


Рис. 1.32 Схема вентиля НЕ

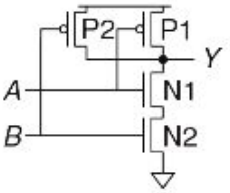


Рис. 1.33 Схема вентиля И-НЕ с двумя входами

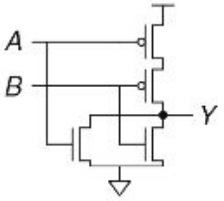


Рис. 1.36 Схема вентиля ИЛИ-НЕ с двумя входами

j)

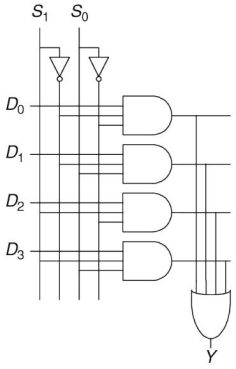
2) Функциональные схемы

a) Функциональных схем два вида: комбинационные и последовательные.

b) **Комбинационные (комбинаторные)** - схемы, которые выдают значение в зависимости только от входных значений. Вот они слева направо: мультиплексор, дешифратор, демультиплексор, сумматор.

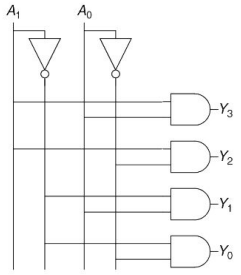
   i) **Мультиплексор** - n управляющих входов, 2^n входов данных, 1 выход. Значения управляющих входов являются двоичным

представлением входа данных, значение с которого передаётся

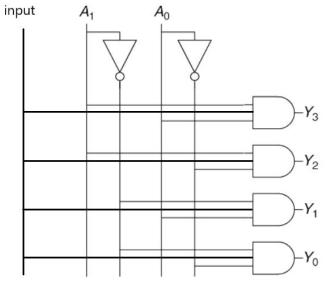


на выход.

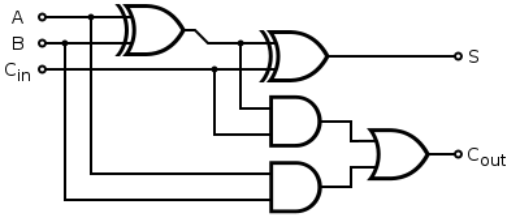
ii) **<u>Дешифратор</u>** - n входов, 2^n выходов. На входы подаётся в двоичном виде номер выхода, на который подастся 1.



iii) **<u>Демультиплексор</u>** - 1 вход данных, n управляющих входов, 2^n выходов. Управляющими входами определяется выход, на который передаётся значение входа данных.

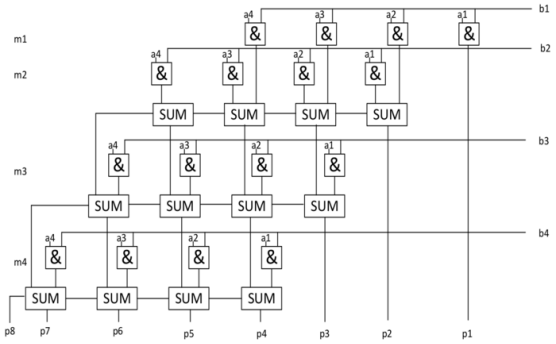


iv) **<u>Полный сумматор</u>** - говорит сам за себя. Принцип: два XOR'а, два AND'а и OR. Младший бит у нас 1 в случаях, если входные биты у нас одинаковы и бит переноса 1 или если бит переноса 0 и входные биты различаются. Старший бит у нас 1, если входные биты различаются И бит переноса 1 ИЛИ если оба входных бита 1.



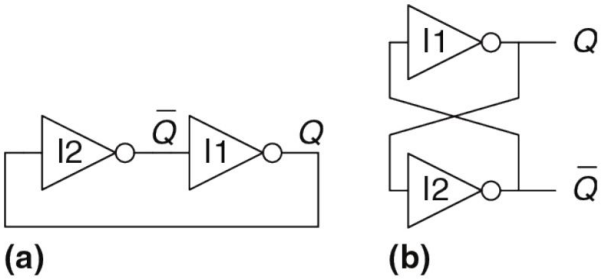
v) Так же ещё существует **<u>каскадный сумматор</u>** - это просто много полных сумматоров подряд. N сумматоров - можем складывать n-битные числа.

vi) **<u>Матричный умножитель</u>** - умножает столбиком, кек. Вот вам схемка, на всякий.

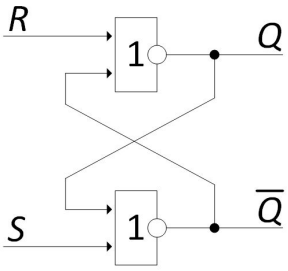


c) Последовательные схемы - зависят не только от входных данных, но и от предыдущего состояния схемы.

i) Самая простая штука - **<u>два инвертора</u>**. Хранят один бит информации без возможности изменения, но обычно можно ручками туда что-то ткнуть.

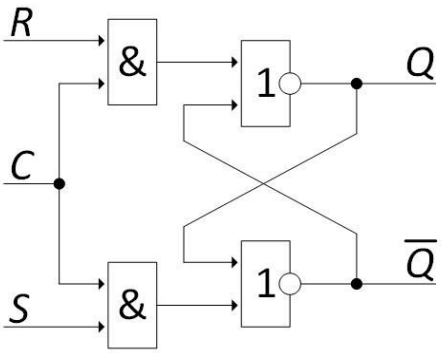


ii) **<u>RS-триггер</u>** aka защёлка(?). Имеет два входа: R(eset) и S(et) - и два выхода: Q и инвертированный Q. При подаче обоих нулей триггер выдаёт то значение, что было до этого. При подаче 1 на S триггер выдаёт 1 на Q, при подаче 1 на R триггер выдаёт 0 на Q. При подаче двух 1 у нас шляпа.



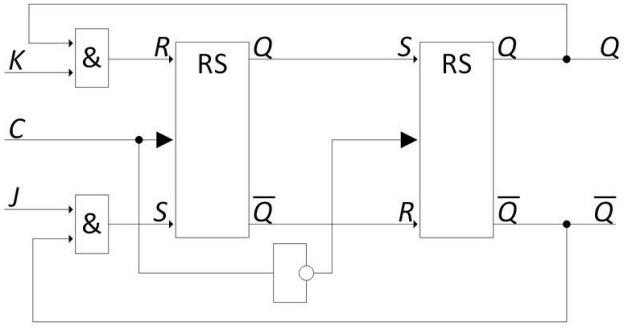
iii) Есть проблема с тем, что мы живём не в идеальном мире (плак-плак) и сигнал по проводам приходит не одновременно. Из-за этого мы можем получить не то, что нам хочется. Решение: добавим провод синхронизации. Если на синхронизации 0, то мы выдаём предыдущее значение, как только синхронизация 1,

начинаем работать по прежней схеме. Частоту смены синхронизации стоит делать такой, чтобы оба сигнала однозначно успевали.При двух единицах у нас всё ещё гроб-гроб-кладбище-кхм.

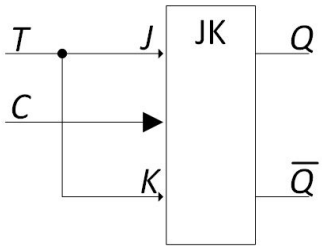


R	S	Q
0	0	Q
0	1	1
1	0	0
1	1	Forbidden

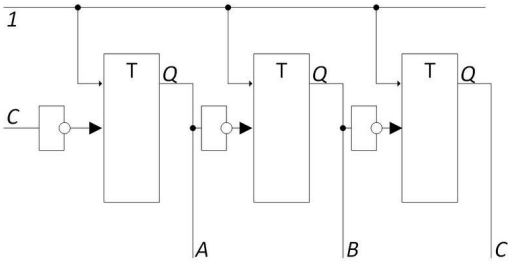
iv) **J(ump)K(ill)-триггер** aka прыгай-убивай - улучшенная версия RS-триггера. Jump = Set, Kill = Reset, а при подаче двух единиц значение Q (и соответственно не-Q) инвертируется. Если убрать из него синхронизацию, то подача двух единиц запускает кошмар эпилептика в виде постоянного переключения. Если синхронизацию (это тот инвертор посередине) перенести в начало, то существует секретная комбинация, которая ломает бедный триггер: 01 (вывели Qпред - поставили внутри 0), 11 (поставили на выход 0, на спаде поставится 1 внутри), 00(мы просили его вывести 0, а он нам выведет 1).



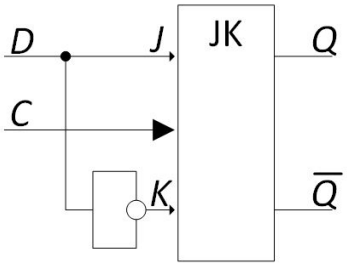
v) **T-триггер**: при подаче 0 выводит предыдущее значение, при подаче 1 инвертируется. Если поддерживать 1 на входе, то меняет значение каждый такт синхронизации.



Можно из таких собрать счётчик.



vi) **<u>D-триггер</u>** - хранит бит, который мы дали ему на вход (хороший мальчик). Если взять много таких, то можно сделать простую ячейку памяти, но об этом чуть позже.



3) Представление чисел в компьютере.

a) В компьютерах используется двоичная система счисления. Это удобно: 0 - нет тока/его мало, 1 - ток есть/гораздо выше, чем при 0. Оптимально хранить числа в системе счисления e (экспонента), но хер ты это реализуешь. В то же время, троичная система счисления обладает большей плотностью записи, но к тому времени как появились нормальные образцы на троичной системе счисления, уже всё устоялось на двоичной и менять на троичную было слишком дорогим и неоправданным риском (Эх, Сетунь, моя Сетунь).

b) Хранить положительные числа легко и приятно, но что делать с отрицательными?

   i) **<u>Прямой код</u>** aka первый бит под знак. Легко получить, имеем одинаковое кол-во положительных и отрицательных чисел. Минусы: у нас появилось два нуля (000 и 100), а также как теперь складывать?

ii) **Код со сдвигом**: ко всем числам прибавляем сдвиг. Т.е. если у нас 2^n чисел всего, то будем сдвигать на 2^{n-1}. Т.е. -5 -> -5 + 128 -> 123, 10 -> 10 + 128 -> 138. Код легко получить, нет двух нулей. Минусы: у нас разное кол-во положительных и отрицательных чисел, а также при реализации арифметических операций нам нужно учитывать сдвиг, что сложна.

iii) **Дополнение до 1** (это точно так называется?): Отрицательное число - инвертированное положительное. Легко преобразовывать, представление натуральных не поменялось, а ещё равенство положительных и отрицательных. Минусы: два нуля, а ещё сложно производить арифметические операции.

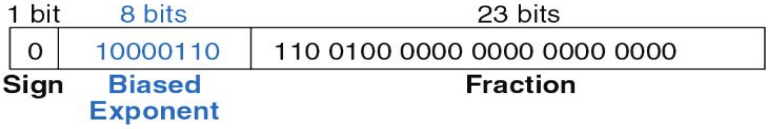
iv) Всякая херня, типа основание -2, основание 3 и прыжки вокруг нуля.

v) **Дополнения до 2**: вес старшего разряда -2^{n-1} (перед двойкой минус, если что). Отрицательное число выглядит как инвертированное положительное плюс один. Со сложением и вычитанием всё ок, нет двух нулей. Минусы: неудобно сравнивать, разное количество положительных и отрицательных. Самый часто используемый вариант.

c) Хранение нецелых - отдельная головная боль. Постоянно помним про неточность вычислений и хранения, сравнивать дробные числа на равенство - всё равно что подвесить над собой гильотину и пилить верёвку, и прочие прелести жизни.
В дробных числах используется бит под знак.

i) **Числа с фиксированной точкой**. Под целую и дробную часть у нас выделено константное кол-во бит. Просто реализовать и понять, удобные арифметические операции. Минусы: небольшой диапазон, также говорят, что непонятно, где заканчивается целая и начинается дробная часть.

ii) **Числа с плавающей точкой**. Один бит под знак, экспонента - целое со сдвигом (обычно на 128), остальное - мантисса.



Число получаем, как $(-1)^{sign} * x, fraction * 2^{exp}$, где x - 0 или 1 (см. дальше). x обычно в записи самого числа не появляется, а держится “в уме”.
Есть два нуля и специальные числа: +inf, -inf (все биты экспоненты 1, все биты мантиссы 0); NaN (все биты экспоненты 1, мантисса ненулевая).
Если нам нужно округлять числа, то округляем в сторону ближайшего чётного.
Обычно, у нас есть возможность одно число представить

бесконечностью способов ($0,5 * 10^1; 0,05 * 10^2; ...$), а теперь про x:

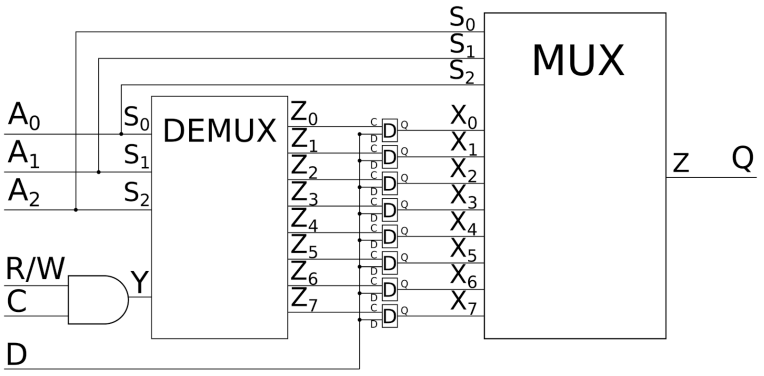
(1) **<u>Нормализованные числа</u>**: x = 1, мантисса принимает значение в диапазоне [1, 2). Не возникает проблемы с тем, что одно число можно представить разными способами. Разичают: half precision (1 + 5 + 10), single precision aka float (1 + 8 + 23), double precision aka double (1 + 11 + 52) и quad precision (1 + 15 + 112).

(2) **<u>Денормализованные числа</u>**: x = 0. Используют, чтобы повысить точность вычислений вблизи нуля. Если все биты экспоненты 0, то мы имеем дело с такими числами. Они работают медленно, но выигрываем в точности.

d) Умножение (деление): складываем (вычитаем) экспоненты, умножаем (делим) мантиссы. Сложение и вычитание: приводим к меньшему из порядков, складываем/вычитаем мантиссы, нормализуем, при необходимости округляем. Кек в том, что если складывать очень большое число и очень маленькое, то из-за ограничений в точности мы получим неизменённое большое число.

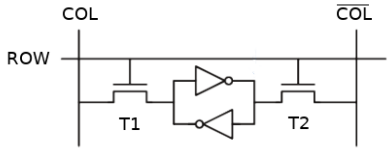
4) Устройство оперативной памяти

a) *Если взять много таких, то можно сделать простую ячейку памяти*

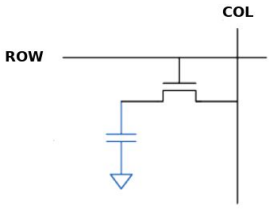


На восьми битах у нас всё хорошо, но в промышленных масштабах это занимает over дофига места, а ещё куча проводов, а ещё D-триггеры-Хрен-Синхронизируешь (а почему они Хрен-Синхронизируешь? Да потому что их хрен, мать его, синхронизируешь!), а ещё это энергии жрёт, словно корейская майнинговая ферма.

b) Немного отвлечёмся от игрушек и посмотрим на ячейки памяти. У нас есть статическая память



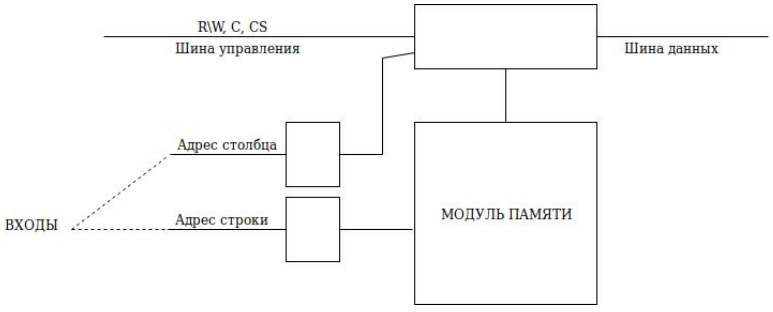
И динамическая



<u>Статическая память</u> как BMW: дорогая (целых 6 транзисторов) и быстрая, а ещё громоздкая. Ей необходимо постоянное энергообеспечение, но взамен она даёт нам стабильное сохранение полученного значения. Для записи подаём то, что хотим записать, на COLumn.

<u>Динамическая память</u> дешёвая (транзистор + конденсатор) и маленькая. За это мы платим тем, что заряд с конденсатора имеет свойство утекать, как следствие ячейку нужно регулярно перезаряжать (примерно раз в 64 мс). Чтобы почитать, подаём на COL 0,5 и если напряжение увеличивается, то в ячейке 1, иначе 0. После чтения нужно перезарядить. Чтобы записать, подаём на COL 0/1 -> заряд из ячейки утекает/втекает.

c) Как в общем выглядит оперативная память:



Писать и читать мы можем через один провод, потому что мы никогда не пишем и не читаем одновременно -> экономия.

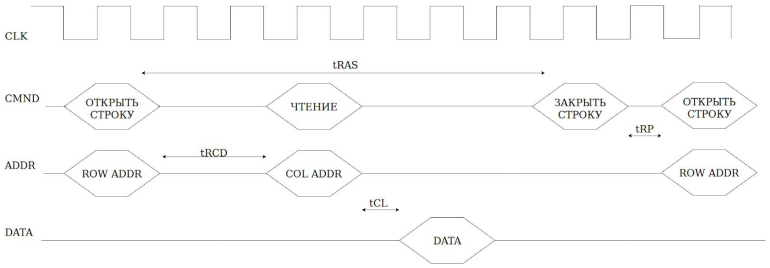
d) Как можно организовать модуль памяти?

Сделать по отдельной линии на каждую ячейку - жесть полная, мы в этих линиях утонем.

Расположить все ячейки в один ряд - нам понадобится ооогромный демультиплексор, а ещё мы хотим поэкономить место, а ещё возникают проблемы с синхронизацией и потерей сигнала.

Поэтому оптимально запихать всё в матричку.

e) Чтение и (наверное)всё, что с этим связано.



tRAS - время между открытием строки и её закрытием, то есть время, требуемое, чтобы полностью пройти цикл чтения из ячейки.
tRCD - задержка между открытием строки и открытием столбца.
tCL - задержка между открытием ячейки и началом получения из неё данных.
tRP - время на перезарядку ряда, т.е. то время, которое пройдёт между тем, как мы закончим читать, и тем, когда сможем снова открывать ячейку.
Здесь имеют значение два параметра: **latency** aka **скорость доступа** (время между началом подачи адреса и началом получения данных) и **throughput** aka **пропускная способность** aka **скорость передачи данных** (время между началом получения одной порции данных и началом получения следующей порции).
Все эти числа обычно измеряются в тактах.

f) Следует различать многопортовые ячейки и многобанковые ячейки памяти. Многопортовые - когда у нас в одну матрицу ячеек идут несколько проводов COL и ROW. Многобанковая - когда у нас есть несколько отдельных матриц с ячейками памяти, в каждую матрицу идут свои проводки.
g) Существует два типа ОЗУ: асинхронная и синхронная. Асинхронная - контроллер памяти и сам модуль не синхронизированы или даже работают на разных частотах, что влияет на скорость работы не в лучшую сторону. Синхронная - контроллер и модуль синхронизированы и работают на одной частоте.
h) Теперь про поколения:
    i) **FastPageMode DRAM** - начало 90-х, принцип - не закрывать строку, если следующий запрос идёт к ней же.
    ii) **EDO DRAM** - первая половина 90-х, принцип - сделаем буфер для адреса столбца и будем подавать адрес столбца, пока ещё идут данные из предыдущего запроса.
    iii) **BurstEDO DRAM** - заметим, что обычно нам, как программисту, не нужен отдельный бит, а нас интересует их последовательность. BurstEDO - модификация EDO, которая отдаёт нам данные сразу из четырёх столбцов подряд. Соответственно адрес столбца должен быть кратен четырём.
    iv) Все поколения до этого были асинхронные. **SDRAM** - вторая половина 90-х, наконец-то синхронизированы контроллер и модуль -> получилось увеличить частоту. А ещё расширили шину

данных (до 64 бит). Также начали появляться первые многобанковые структуры (обычно 2-4 банка).

v) **DDR** - 1998, расширили внутреннюю шину в два раза, а также начали передавать данные не только на фронте (возрастании) сигнала (наверное, синхронизации), но и на его спаде. Как следствие, “увеличили” частоту в 2 раза.

vi) **DDR2** - 2003, расширили внутреннюю шину (в два раза), увеличили частоту внешней шины (в два раза), увеличили число банков (до 8).

vii) **DDR3** - 2007, расширили внутреннюю шину (в два раза), увеличили частоту внешней шины (в два раза), снизили напряжение, а значит и энергопотребление.

viii) **DDR4** - 2014, уменьшили напряжение питания, увеличили число банков (до 16). Шину не трогали.

i) Для видеокарт не так важна скорость доступа (latency), как скорость передачи данных (throughput). Поэтому в видеокарточках стоят поколения GDDR.

i) **GDDR2** - производная от DDR2, не взлетела.

ii) **GDDR3** - производная от DDR2, это поколение взлетело, оптимизировали скорость передачи данных.

iii) **GDDR4** - производная от DDR3, не взлетела.

iv) **GDDR5** - производная от DDR3, взлетело, передаём 4 порции данных за такт, имеем два канала данных наружу.

v) **GDDR5x** - чуть улучшили скорость передачи по сравнению с пятым поколением.

vi) **GDDR6** - производная от DDR3 или DDR4, повысили скорость передачи, снизили напряжение.

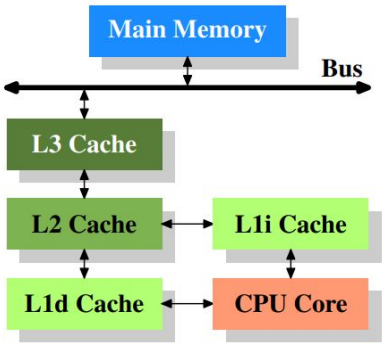
5)Кэши.

a) В самом начале частота ядра процессора была примерно равна частоте шины памяти, что делало доступ к памяти лишь немногим медленнее доступа к регистрам. С развитием ЦП частота ядер возрастает очень быстро, а шину мы такой же не сделаем (вернее, можем, но мы будем потреблять и греться, как корейская майнинговая ферма). Поэтому возникло решение поставить немножко быстрых статических ячеек прямо рядом с процессором специально для его нужд.

b) В кэшах обычно хранится копия данных из ОЗУ. Например, код команд, которые сейчас выполняются и будут выполняться (если у нас много циклов) или участок памяти (если мы идём по массиву, расположенному последовательно). Так как кэш близко к процессору - мы тратим мало времени на получение этих данных.

c) Примерные размеры кэшей и время на доступ к ним: L1 - (10 - 32) Кб/5 тактов, L2 - (100 - 256) Кб/20 тактов, L3 - 10 Мб/50 тактов. Для сравнения: для доступа к ОЗУ требуется 200-250 тактов. Обычно кэш первого уровня разделяют на кэш команд и кэш данных. Также кэши 1 и 2 уровней отдельные для каждого ядра/пары ядер, в то время как кэш

третьего уровня общий на все ядра.



d) Читаем из кэша. Для этого есть два подхода: look-aside и look-through.
Look-aside - смотрим необходимые данные как в кэше, так и сразу в оперативке. Плюсы: если данных в кэше нет, мы их быстрее найдём. Минусы: мы на любой запрос забиваем шину ОЗУ, а также кэш и оперативка висят на одной шине.
Look-through - смотрим сначала в кэше, а если не нашли - идём в память. Мы не забиваем шину оперативки, можем сделать широкую шину между ЦП и кэшом. Минус: при промахе (т.е. данных нет в кэше) мы дольше отвечаем на запрос.

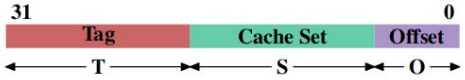
e) Пишем в кэш. Опять же два подхода: write-through и write-back.
Write-through - записываем сразу и в кэш, и в соответствующий участок оперативы. Плюсы: легко реализуется. Минусы: забиваем шину, часто переписываем один и тот же кусок.
Write-back - запишем в кэш, а там когда-нибудь скинем изменения в ОЗУ. Плюсы: мы не трогаем оперативку без лишней надобности. Минусы: это сложно реализовать на аппаратном уровне.
Если мы промахнулись и данной линии нет в кэше, то есть следующие выходы:
Write allocate/fetch on write - загрузить строчку в кэш, а потом её изменить.
Write around - писать сразу в память, не трогая кэш.

f) Если мы не обнаружили необходимую нам строчку в кэше, значит её туда нужно запихнуть. А для того, чтобы запихнуть что-то нужное, надо выкинуть что-нибудь ненужное.
Less Recently Used - выкинуть строчку, которую дольше всех не трогали.
Most Recently Used - выкинуть строчку, к которой в последний раз обращались.
Less Frequently Used - выкинуть строчку, к которой реже всего обращались.
First In First Out - выкинуть строчку, которая дольше всех сидит в кэше.
При выкидывании мы не просто её затираем, а сбрасываем в ОЗУ. Т.е. именно в этот момент все изменения окажутся в оперативке.

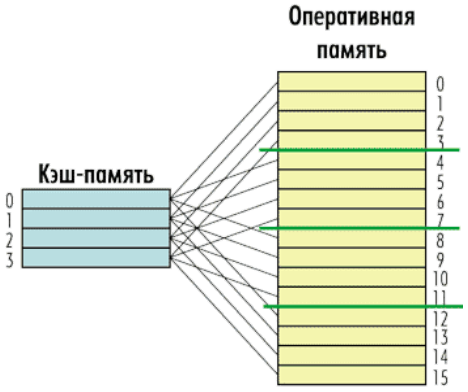
g) Есть два способа хранить данные в кэше.
Exclusive cache - способ от AMD. Освобождая место, мы строчку спихиваем на более нижний уровень (а там, как доминошки, тоже что-то

выкинется). Данные в кэшах не дублируются.
Inclusive cache - способ от Intel. Строки, хранящиеся в L1, также присутствуют в L2. L2, в свою очередь, присутствует в L3.
В inclusive быстро происходит удаление, в exclusive - загрузка в кэш.

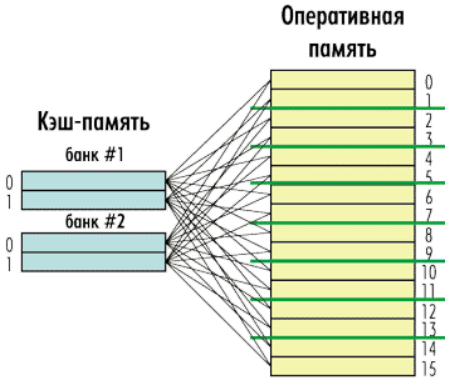
h) Ещё про хранение.
Если хранить в кэше побайтово, то нас всё плохо: вместе с 8 битами информации мы держим 32 бита адреса. Вместо этого хранится кусок данных, идущий подряд (64 или 32 байта). Адрес кратен 64 -> экономим 6 бит. Tag = адрес, offset = сдвиг внутри одного куска.



i) Политики кэширования.
Как нам отобразить оперативную память в кэш?
Вариант 1: **полная ассоциативность** - в любой строке кэша может находиться любая строка оперативки. Минусы: нам сложно искать информацию по тегу, нужно обойти все строки кэша.
Вариант 2: **прямое отображение** - строка “блока” оперативки проассоциирована ровно с одной строкой кэша. В ней легко искать, но мы очень часто будем замещать строки в кэше.



Вариант 3: **наборно-ассоциативный кэш** - делим кэш на банки. Внутри банка - прямое отображение, между банками - полная ассоциативность.



j) Когда у нас одно ядро, то всё хорошо: пишем в кэш, читаем из него, сами себе короли. Однако при появлении нескольких ядер, всё резко ухудшается. Грустный пример: ядро_1 прочитало себе в кэш строку, ядро_2 прочитало себе в кэш строку. Ядро_1 поменяло свою строку и сбросила в память, но ядро_2 всё ещё думает, что у него актуальная

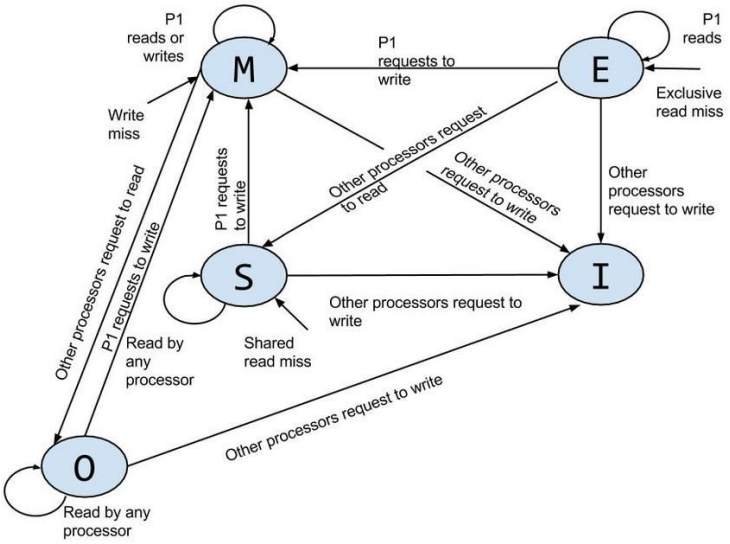
строка и вообще не узнает, что там что-то не так.
Чтобы такого не происходило, изобрели **протоколы когерентности кэшей**.

i) **MSI**: у кэш-линии может быть три состояния:
M(modified) - мы меняли кэш-линию, т.е. данные в кэше не совпадают с данными в памяти. Мы можем спокойно читать эту линию и менять её дальше. Однако на кэше лежит великая ответственность записать потом эту строчку в память.
S(shared) - кэш-линия не изменена и присутствует хотя бы в одном кэше. Можно эту строчку выкидывать без записи в память. Можно спокойно читать.
I(invalid) - кэш-линия не присутствует в данном кэше или неактуальна (т.е. в каком-то другом кэше эта линия была изменена). Естественно, мы с такой строчкой ничего делать не можем.
Когда мы загружаем строчку из памяти, она становится Shared. Как только мы её меняем, у нас она становится Modified, а у других Invalid. Если кто-то из тех, кто имеет её Invalid захочет поработать с этой строкой, к нам придёт уведомление, о том, что нам нужно скинуть эту строку в память. Мы сбрасываем, переводим строку в положение Shared. Те, кто хотел, забирают эту строку из памяти и тоже ставят Shared.
Проблемы: часто у процессора есть данные, которые имеются только у него, но при этом он всё равно должен каждый раз уведомлять остальных при изменении Shared -> шина забивается мусором.

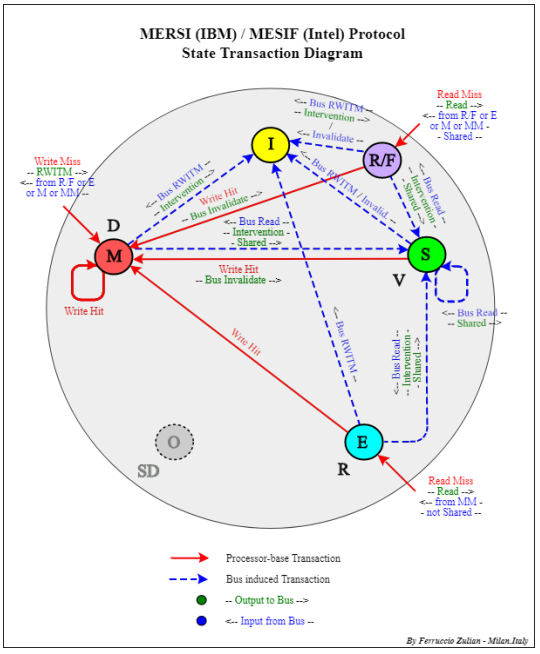
ii) **MESI**: добавили ещё одно состояние:
E(exclusive) - данная кэш-линия присутствует у нас и *только* у нас. Мы можем менять эту строку, никому об этом не говоря.
Проблемы: если несколько кэшей работают с одной линией, её постоянно приходится сбрасывать в память. Когда ядро запрашивает состояние линии, ему отвечают все, имеющие Shared, забивая тем самым шину.

iii) **MOESI** - модификация от AMD:
O(owned) - мы можем транслировать изменения линии всем Shared, не используя память. Мы приходим в это состояние, если мы были в состоянии M, и кто-то попросил почитать. Уходим в M, если нам понадобилось изменить её. Уходим в I, если кто-то

другой захотел поменять её.



iv) **MESIF** - модификация от Intel:
F(forward) - следующая ветка развития Shared. Forward должен отвечать всем другим кэшам на вопросы об этой кэш-линии, чтобы у нас не было потока мусорных ответов. Если вдруг никого с Forward не оказалось, то отвечает память. Чтобы такое реже происходило, состояние F переходит к получателю информации. Отличие F от O в том, что Owned -может быть изменённая линия, Forward - обязательно “чистая”, при изменении превращается в Modified.

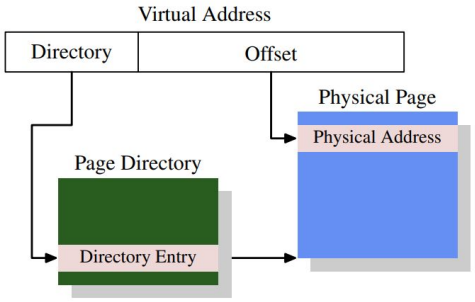


6)Виртуальная память.

a) Что это и зачем это? Мы изолируем процессы в памяти (в общем случае, процесс не имеет доступ к чужой памяти), каждый процесс считает, что он один и что вся память принадлежит ему -> упрощаем жизнь программисту. Если мы не можем выделить один цельный большой

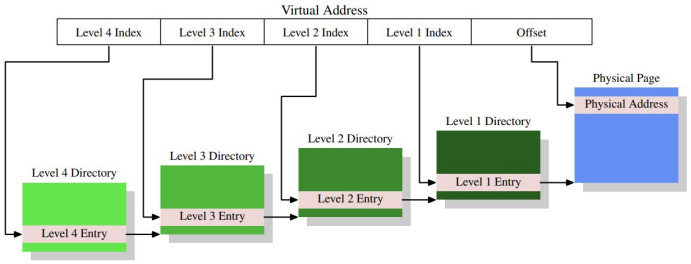
кусок, можно взять несколько разрозненных, “склеить”, а процессу будет ок.

b) Как реализуется? Виртуальный адрес состоит из адреса директории (т.е. страницы) и адреса внутри директории. По первому адресу мы в Page Table ищем ссылку на нужную нам страницу, переходим по ней и используем второй адрес.



c) Проблемы: стандартный размер страницы - 4 Кб. Обычно offset - 12 бит, directory - 20 бит. Если каждая запись весить 4 байта, то вся табличка весит 4 Мб. Каждый процесс имеет свою табличку, а процессов много -> у нас вся память ушла на таблички.

d) Чтобы такого не было, создаются многоуровневые таблицы страниц (обычно 4, редко 5 уровней). Адрес самой верхней Level 4 директории хранится в специальном регистре CR3. Если какая-либо запись пустая, то нам нет хранить директории более низкого уровня. Как следствие, получаем разреженное дерево.

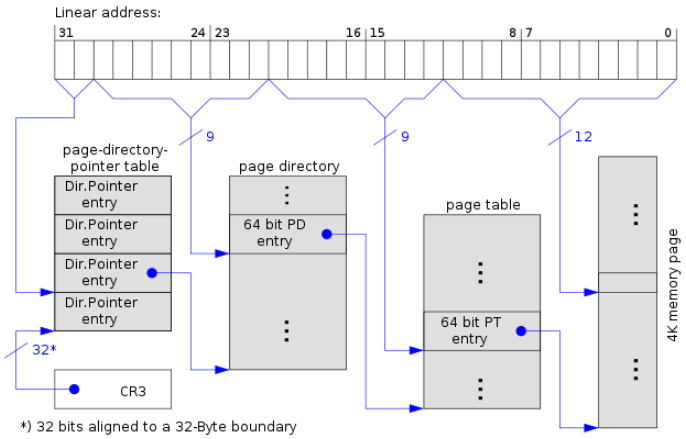


e) Однако каждый раз идти искать долго и неприятно. Давайте заведём специальный маленький кэш под это, так называемый **TLB (Translation Lookaside Buffer)**. Будем там хранить наиболее часто используемые записи page table. Если не попали в буфер, то с лицом лягушки спускаемся по дереву.
Проблемы: TLB один на процессор (общий для всех ядер), а у каждого процесса свой page table.
Возможное решение: сбрасывать TLB каждый раз, как меняется процесс. Исключение - само ядро (или ОС), его мы будем сохранять - дорого и неприятно, но реализуемо.
Другое возможное решение: расширим тег адреса, чтобы однозначно определять page table.

f) Page entry могут ссылаться как на ОЗУ, так и на диск, т.е. сами физические страницы могут храниться как в ОЗУ, так и на жёстком диске. Если мы хотим почитать страницу, а её нет в оперативе, то страница

подгружается с жёсткого диска. Здесь оперативная память фактически является кэшом для виртуальной.

g) **PAE (Page Address Extension)** - расширяем размер адресуемой памяти до 64 Гб на 32-битной машине: каждую запись (page entry) расширим до 64 бит, уменьшив их количество в два раза, а также добавим ещё один уровень.



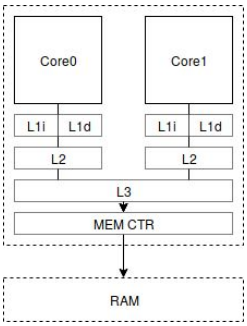
7) ISA (Instruction Set Architecture) и микроархитектуры.

a) Прежде всего, поговорим про организацию чипсетов и доступ к памяти.

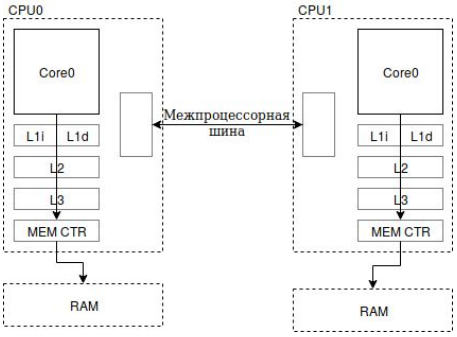
i) Существует так называемая **“Двухмостовая архитектура”**, её прикол в том, что у нас есть северный мост, к которому подключены ЦП и контроллеры памяти, и южный мост, на котором висит медленная периферия. Иначе это можно сформулировать как “Всё быстрое - на северном, всё медленное - на южном”.

ii) Развитием двухмостовой архитектуры стала система **“Система на кристалле”** (**System on Crystal** aka **SoC**). Она заключается в том, что мы подвешиваем почти всё на процессор: контроллеры памяти, видеокарты, ввода-вывода, иногда даже южный мост.

iii) С точки зрения доступа процессора к памяти различают **Uniform Memory Access** и **Not-Uniform Memory Access**.
UMA - однородный доступ к памяти, т.е. к любому блоку обращаемся за одинаковое время. Плюсы: мы не можем проиграть с тем, что какая-то информация лежит далеко. Минусы: сложно масштабировать. Канал к памяти - бутылочное горлышко (ботлнек), т.е. самая медленная часть всей системы, от которой всё зависит.



NUMA - у каждого процессора есть “свой” блок оперативной памяти, наиболее близкий к нему. Если нам понадобилась информация из не своего блока, то мы стучимся к другому ядру -> время доступа к памяти неоднородное. Плюсы: лучше масштабируется, мы можем расположить больше блоков памяти, а также возрастает параллельность запросов к памяти. Минусы: очевидное замедление, если мы не попали в свою память.



b) Сначала философия.
   i) **Архитектура фон Неймана**:
   Используем двоичную систему счисления. Альтернатива - троичная система, десятичная;
   Память состоит из ячеек, каждая из которых имеет свой адрес, к которому можно обратиться. Альтернатива - стековая архитектура, машина Тьюринга;
   Программное управление - ЭВМ выполняет программы, записанные в памяти. Альтернатива - аппаратное управление, когда код зашит в железе (например, микроконтроллеры и всеми любимая Arduino);
   Последовательное выполнение команд - ¯_(ツ)_/¯. Альтернатива - многопроцессорные (и многоядерные) архитектуры.
   Однородность памяти - код программ и данные хранятся в одной и той же памяти. Альтернатива - Гарвардская архитектура, в которой код и данные хранятся физически в разной памяти;
      (1) *Плюсы Гарвардской*: можно одновременно читать код и данные, память кода можно защитить от записи -> супер безопасно
      Минусы Гарвардской: усложняется контроллер памяти, нужно в два раза больше шин, память может простаивать

"без дела".
Особенность: можно использовать плашки памяти с разными характеристиками (например, побыстрее для данных, помедленнее для кода).

(2) *Плюсы фон Неймана*: эффективно используем всю память, проще контроллер, одна шина.
Минусы фон Неймана: память с кодом не защищена, можно перезаписать или прочитать не тем людям.
Особенность: однородный доступ к коду и данным.

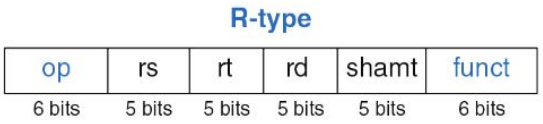
c) **Архитектура набора команд** aka **ISA** описывает следующее:
   i) Архитектура памяти, разрядность адресов, режимы адресации, кол-во регистров
   ii) Набор команд машинного языка, как они кодируются
   iii) Типы данных
   iv) Обработка прерываний и исключений
   v) Взаимодействие с внешним миром
d) Примеры архитектур: x86, ARM, MIPS
e) **Архитектуры памяти**:
   i) Стековая - операнды лежат на стеке, операции берут значения с верхушки стека и кладут результат обратно.
   ii) Аккумуляторная - стек размера 1; операции берут один операнд из памяти, второй из аккумулятора и кладут результат в аккумулятор.
   iii) Reg-Reg - операнды лежат в регистрах, результат кладём в регистры.
   iv) Reg-Mem - один из операндов может быть из памяти.
   v) Mem-Mem - оба операнда могут быть из памяти.
f) **Кодирование команд**:
   i) Команды переменной длины - команды имеют разную длину, часто используемые команды можно закодировать меньшим числом байт, но сложно декодировать. Пример - x86 (от 1 до 15 байт)
   ii) Команды постоянной длины - все команды имеют константную длину. Иногда приходится разбивать одну команду на две, но зато просто декодировать. Пример - MIPS(все по 4 байта).
g) **Наборы команд**:
   i) CISC - для каждой команды есть реализация в железе. Плюс: энергоэффективно за счёт аппаратной реализации всего и вся. Минус: редко используемые команды занимают место на кристалле.
   ii) RISC - на кристалле только простые вычислительные блоки. Сложные, но редкие команды реализуем программно. Это менее энергоэффективно, но взамен освобождается место, например, под кэш.
h) **Режимы адресации**:

i) Register/Регистровая адресация: *add r1 r2 r3 <=> r1 = r2 + r3* - всё берём из регистров
ii) Immediate/Непосредственная адресация: *add r1 r2 5 <=> r1 = r2 + 5* - что-то может быть вшито в код константой
iii) Register indirect/Косвенная регистровая адресация: *add r1 r2 (r3) <=> r1 = r2 + mem[r3]* - “сходи по адресу, который лежит в регистре”
iv) Displacement/Индексная адресация: *add r1 r2 100(r3) <=> r1 = r2 + mem[100 + r3]* - “сходи по адресу, который отличается от адреса в регистре на константу”
v) Absolute/Абсолютная адресация: *add r1 r2 0xabcd <=> r1 = r2 + mem[0xabcd]* - “сходи по вот этому адресу”, адрес вшит константой.

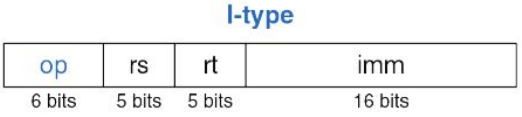
i) Теперь поговорим немного про ассемблер (на примере MIPS’a).
У процессора есть 32 32-битных регистра. Соответственно, каждый регистр можно однозначно задать пятью битами.

Название	Номер	Назначение
$0	0	Константный нуль
$at	1	Временный регистр для нужд ассемблера
$v0–$v1	2–3	Возвращаемые функциями значения
$a0–$a3	4–7	Аргументы функций
$t0–$t7	8–15	Временные переменные
$s0–$s7	16–23	Сохраняемые переменные
$t8–$t9	24–25	Временные переменные
$k0–$k1	26–27	Временные переменные операционной системы (ОС)
$gp	28	Глобальный указатель (англ.: global pointer)
$sp	29	Указатель стека (англ.: stack pointer)
$fp	30	Указатель кадра стека (англ.: frame pointer)
$ra	31	Регистр адреса возврата из функции

j) Есть три типа команд:
i) **R(register)-type** - все операнды регистры, op и funct определяют операцию. shamt - что-то про сдвиги.



ii) **I(immediate)-type** - использует константу вместо одного из регистров.



iii) **J(jump)-type** - операции для переходов.

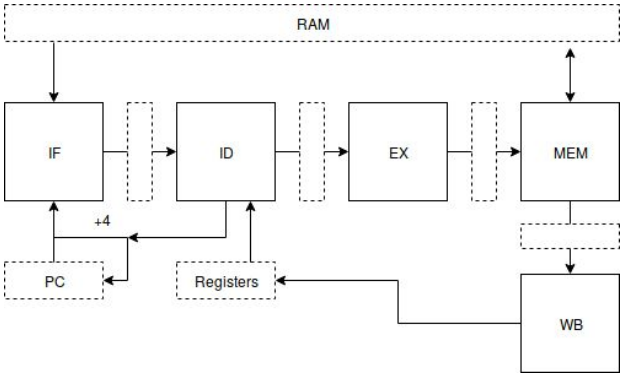
k) Что же такое микроархитектура? Ответ прост: **микроархитектура** - это конкретная реализация архитектуры в железе.

l) Виды микроархитектур:
Однотактная - все команды выполняются за один такт, такт - время выполнения самой долгой команды.
Многотактная - разбиваем команду на куски, переиспользуем вычислительные устройства.
Конвейерная.

8) Конвейерная микроархитектура.

a) Заметим, что если выполнять всё последовательно, то во время выполнения часть железа будет простаивать без дела. Давайте разделим процесс выполнения команды на несколько стадий, на каждую из которых будем тратить по такту. Соответственно, когда команда прошла первую стадию, можно подавать следующую команду.

b) Конвейер MIPS:



IF - Instruction Fetch - читаем команду из памяти. **PC (Program Counter)** хранит адрес выполняемой программы.
ID - Instruction Decode - декодируем команду, разбираем её на код регистров, констант и проч., вспоминаем R-, I-, J-type команды.
EX - Execute - выполняем команду, вычисляем на АЛУ, считаем для неё что-либо.
MEM - Memory - на этой стадии мы читаем из памяти или пишем в неё.
WB - Write-Back - посчитанные значения записываем в регистры (если необходимо).

c) У нас появилось некое подобие параллельности. В среднем, мы выполняем больше команд за единицу времени (одна команда за 5 тактов, но 100 команд за 104 такта). Ещё стадии у нас простые, поэтому тактовую частоту можно приподнять.

d) В каждой ложке дёгтя есть бочка мёда. Некоторые программы у нас не могут корректно выполняться на конвейере. Возникают так называемые **конфликты** aka **hazards**.

   i) **Data hazards (конфликты по данным)** - возникают, когда идущие подряд инструкции не независимы (a = b + c; x = a + y).
   Варианты решения:
      (1) Приостановить конвейер (то есть накидать пустых операций aka NOP).
      (2) Написать в документации (мол, всё хорошо после четырёх тактов, а до этого - вы сами себе злобный Буратино).

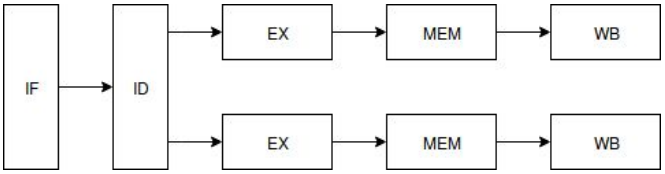
(3) Forwarding - сделать так, чтобы поздние стадии отдавали результаты, если ранние попросят.

ii) **Control hazards (конфликты управления)** - когда мы не знаем, какая команда должна выполняться следующей (условные ветвления, циклы). Варианты решения:
   (1) Подождать, пока условие вычислится.
   (2) Сказать в документации, мол, следующая после условия команда тоже выполнится (чисто по приколу).
   (3) Branch prediction - мы пытаемся угадать, куда мы пойдём и начинаем просчитывать одну из веток. Если нам повезло, радостно продолжаем работать. Иначе с лицом лягушки всё сбрасываем и рассчитываем другую ветку.

iii) **Structural hazards (конфликты по ресурсам)** - возникает, когда две разных стадии хотят использовать один и тот же ресурс (например, память). Варианты решения:
   (1) Подождать, пока поздняя стадия не освободит ресурс.
   (2) Добавить ресурсов, разделив память данных и команд (закос под Гарвардскую архитектуру).

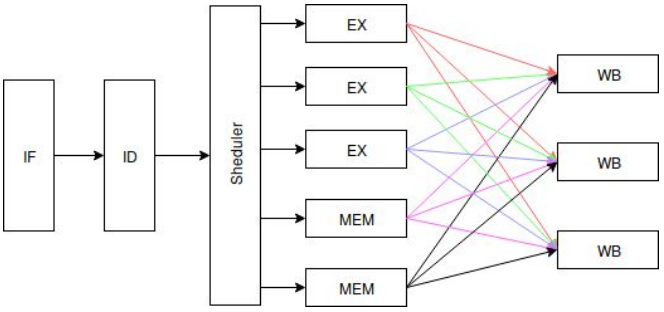
e) Пытаемся всё сделать ещё лучше, а именно апгрейдим конвейер, добавляя больше конвейеров:

i) **VLIW - Very Long Instruction Word** - сделаем “очень длинную операцию”, которая состоит из нескольких стандартных операций. Тогда на стадия IF и ID программа разобьётся на обычные и они пойдут на выполнение сразу нескольких конвейеров.



Плюсы: аппаратно всё легко и дёшево. Компиляторы более-менее умные и могут соптимизировать команды.
Минусы: ломаем жизнь разработчикам компиляторов. Меняем микроархитектуру - надо менять ISA или хотя бы компилятор (добавили конвейер - нужно добавлять ещё одну команду в длинное слово, убрали конвейер - команду из слова нужно убрать). Стадия ID усложняется. Также, если у нас код сильно зависимый, то VLIW’ы будут на приличную часть состоять из NOP’ов.

ii) **Superscalar** - добавим умную железку под названием “планировщик”, которая распределяет команды по конвейерам.



Плюсы: для программиста и разработчиков компилятора всё просто и прекрасно. Планировщик очень умный, поэтому в случае чего может соптимизировать. Изменение микроархитектуры не меняет ISA.

Минусы: наш планировщик должен быть достаточно умным, чтобы мы получили выигрыш в скорости -> повышение цены. У нас появляется больше возможностей застрелиться, ибо hazards. В конце нам нужно синхронизировать, если вдруг какой-либо конвейер закончил раньше.

iii) Виды Superscalar:

(1) In order / In order: команды выполняются в порядке поступления в очередь планировщика. Пока предыдущие не выполнились, новые не загружаются.
Плюсы: просто и дёшево. Только обычные конфликты.
Минусы: недостаточно быстро.

(2) In order / Out of order: команды поступают на выполнение в порядке поступления очереди планировщика, но какие-то закончить могут быстрее, т.е. порядок может не сохраниться.
Здесь появляется конфликт “Write after Write” - когда команда, которая начала выполняться раньше, но выполнялась дольше, затирает значение, вычисленное командой, которая поступила позже.

(3) Out of Order / Out of Order: планировщик говорит “Я здесь царь” и начинает менять порядок, как ему кажется нужным. Вы увидите все вышеназванные конфликты + “Write after Read” (когда команда не успела прочитать значение, которое ей полагалось, а его уже изменили).

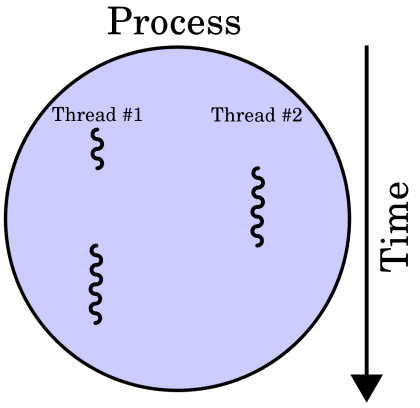
iv) Проблемами WaW и WaR занимается планировщик и стадия WB.

9) Процессоры.

a) Необходимая терминология:

Процесс - кусок программы, запущенный на выполнение. Процессы достаточно независимы друг от друга и обычно не имеют общей памяти/кода.

Тред/Поток - наименьшая последовательность инструкций внутри процесса, которой может независимо управлять планировщик.



Разбивать процесс на потоки - забота программиста.

b) Параллельность команд мы обсудили в предыдущем пункте. Теперь нам хочется параллельности программ.
Если у нас одно ядро, то труъ параллельности нам не достичь. Зато мы можем давать каждой программе поработать по чуть-чуть, а потом передавать управление другой программе, создавая иллюзию независимой работы. Пример: текстовый редактор взял несколько символов из буфера -> музыкальный проигрыватель обработал следующий кусок песни -> мессенджер отправил пакет по сети.

c) Со временем у нас возникли проблемы с задиранием тактовой частоты процессора, поэтому начали идти от улучшения одного ядра к добавлению нескольких ядер.

d) Здесь нужно различать:
Многоядерный: несколько вычислительный ядер на одном кристалле. Кэши первого и второго уровней у каждого свои, а кэш третьего уровня общий.
Многопроцессорный: физически несколько процессоров (они, кста, могут быть многоядерными).

e) Закон Амдалла, который говорит, что многопоточность не панацея и вообще зависит от кода:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}}$$

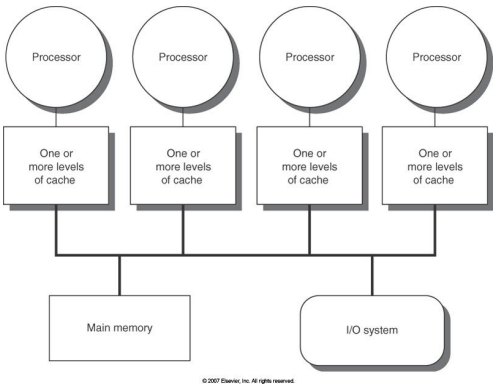
Здесь S - ускорение по сравнению с одноядерным ЦП, a - доля непараллельного кода, p - количество вычислительных ядер.

f) **SMT (Simultaneous MultiThreading)** - один процессор “параллельно” выполняет несколько потоков. Одно физическое ядро притворяется двумя.
HyperThreading - реализация SMT от Intel. *Что мы имеем*: процессор может хранить состояния двух потоков, имеет два набора регистров и два отдельных контроллера прерываний.
Что происходит: когда один поток простаивает по какой-либо причине, ядро занимается вторым потоком.

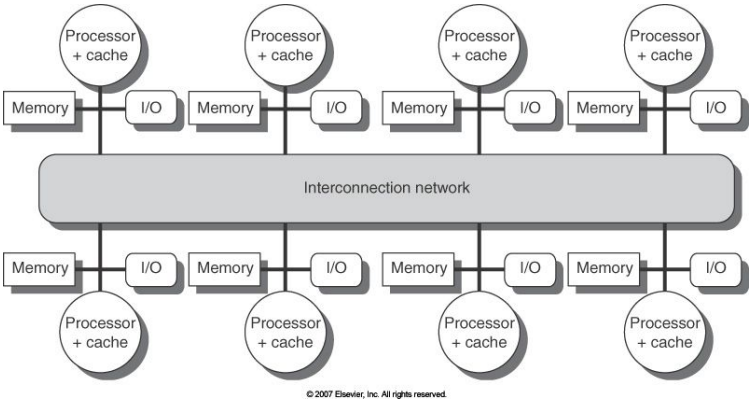
Почему работает: у нас происходят кэш-промахи, поток ожидает результатов выполнения функции или ждёт чего-то от ввода/вывода. В это время мы невозбранно можем заниматься вторым потоком.
Возможные проблемы: потоки могут не поделить ресурсы типа кэша или TLB; если мы действительно хорошо распараллеленный код запихнем на одно физическое ядро (у него биполярочка, но мы-то об этом не знаем), то вся наша параллельность накроется медным тазом.

g) Виды параллельных архитектур:
SISD (Single Instruction stream / Single Data stream) - простой одноядерный процессор.
SIMD (Single Instruction / Multiple Data) - характерно для видеокарт и векторных процессоров.
MISD (Multiple Instructions / Single Data) - в семье не без урода (бесполезен).
MIMD (Multiple Instructions / Multiple Data) - многоядерные процессоры, параллелизм на уровне потоков.

h) Организация памяти для MIMD:
   i) Centralized shared-memory - пока у нас немного ядер, можно поддерживать общую для всех память, т.е. UMA архитектура.



   ii) Distributed-memory - когда у нас много процессоров, нам неудобно поддерживать общую память, поэтому прибегаем к NUMA.

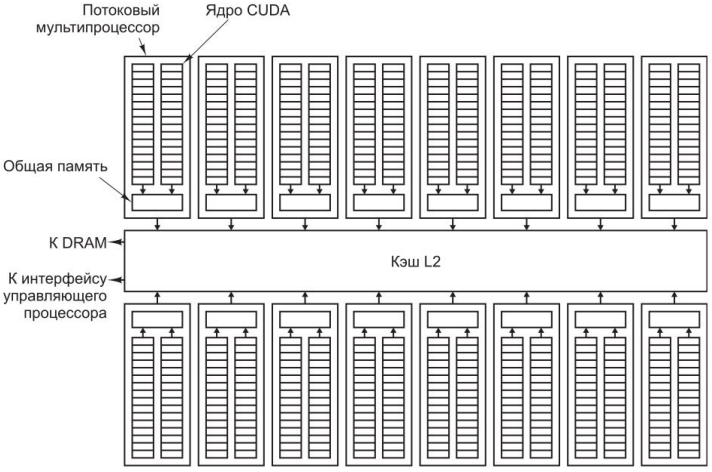


i) Теперь про SIMD:
Это используется, когда нам нужно провернуть одну и ту же операцию над массивом данных (перемножить матричку, посчитать скалярное произведение, всякая такая линально-геометрическая шняга).

Изначально использовалось в векторных процессорах, но они были вытеснены обычными, ибо мощнее, да и многие современные ЦП умеют в векторные вычисления.

j) При чём тут видеокарточки и GPU?
Дело в том, что видеокарты прошли длинный путь от посредника, который преобразует цифровой сигнал в аналоговый для экранов того времени, до непосредственно серьёзной вычислительной машины. Сначала они забрали у процессоров обязанности по обработке графических интерфейсов ОС (ибо последние очень сильно нагружали ЦП), затем занялись проецированием 3D изображений на 2D экраны. Прикол в том, что все вышеперечисленные действия требуют одного и того же:произвести одинаковые действия для огромного количества элементов.

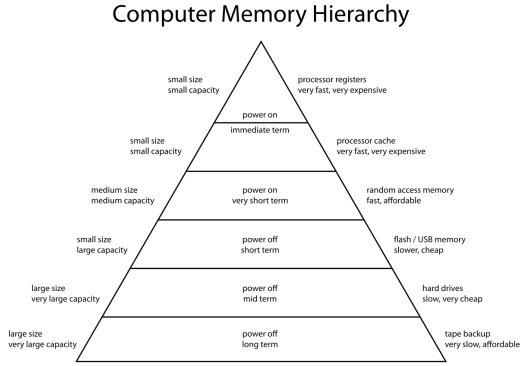
k) На текущий момент видеокарты приблизились к процессорам, что родило идею **GPGPU - General Purpose GPU** (как-то так расшифровывается): производить на видеокарточках не только вычисления графики, но и вообще вычисления.
Отдельного упоминания стоит **CUDA - Compute Unified Device Architecture.** У нас есть много (очень) не шибко сильных ядер, с помощью которых мы хорошо параллелим вычисления.



10) Внешние устройства памяти.

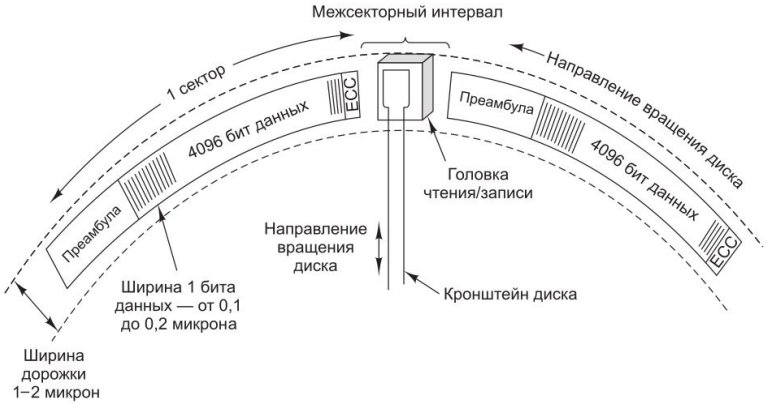
a) Иерархия устройств. Слева - размер самого устройства и вмещаемый объём. Посередине - зависимость от питания, время “жизни”. Справа -

примеры устройств, скорость, цена.

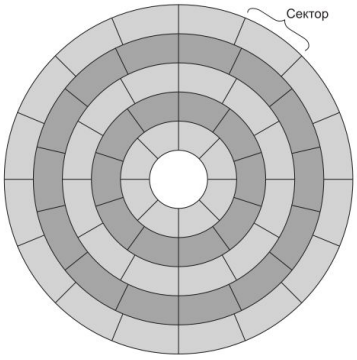


b) Всё началось с **перфокарт**. Перфокарты - картонные или металлические пластины, представляют информацию наличием или отсутствием отверстий в определённых позициях.
Двоичное кодирование - перфокарта транслировалось напрямую в 0 и 1.
Текстовое - каждая колонка/строка отвечала за какой-то символ.
На перфокартах хранились компиляторы ЯП и библиотеки для расчётов.
Считывание - с помощью маленьких металлических щёток или с помощью фотоэлектрических элементов.

c) Следующий этап - **магнитные накопители**, в виде магнитных лент и магнитных дисков.
Представляет собой одну/несколько ленточных/алюминиевых/стеклянных поверхностей с перемещающейся головкой диска.
У нас на устройстве чтения/записи есть магнитные головки, которые прижимаются к поверхности диска и намагничивают или считывают кусок диска. Диск вращается специальным моторчиком.
Дорожка - намагниченный участок, круговая последовательность бит.
Дорожка делится на секторы, где сектор - минимальная адресуемая единица памяти на диске. Суммарно дорожки выглядят как ряд концентрических кругов.
Перед данными есть преамбула, синхронизирующая головку, после данных - код восстановления (Хемминга/Рида-Соломона). Между секторами есть интервал.



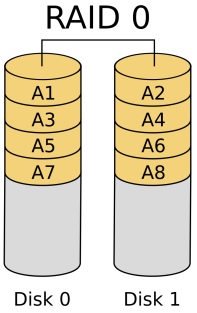
d) **Жёсткие диски**. У нас есть набор пластин, одна или обе стороны пластины используются в качестве рабочей поверхности (обычно 1-12 пластин, 1-24 поверхностей).
Производительность ЖД зависит от: времени на сдвиг головки, время на позиционирование, на нахождение правильного сектора; плотности записи данных, скорости вращения. (Тут всё плохо со случайным доступом)
У нас возникает проблема с тем, что длина внешних дорожек больше, чем длина внутренних. Диск вращается с постоянной скоростью -> мы либо не успеваем, либо слишком долго ждём. Вариант решения 1: сделать на самой маленькой центральной дорожке максимальную плотность записи, а потом уменьшать её, сохраняя постоянным кол-во секторов (неоптимально). Вариант решения 2: поделить диск на зоны. Дорожки внутри одной зоны имеют одинаковую плотность записи.
(Зоны - это по кругу идут)



e) Контроллер жёсткого диска - микросхемка, которая обеспечивает связь диска с внешним миром. Именно контроллер получает все команды на чтение/запись, он управляет перемещением головки, исправлением ошибок в коде, а также следит за “здоровьем” ЖД.
f) Раньше были системы с IDE (Integrated Drive Electronics) - подключались напрямую к материнской плате. Общение с диском происходило через BIOS.
Поддерживали следующую нумерацию: головки и цилиндры шли от 0, секторы от 1. Максимальный размер - 504Мбайта.
Проблема - схема вызовов была зашита в BIOS и не поддерживала диски больше 504Мб (вернее, поддерживала, но с лютыми костылями).
g) Логическим продолжением стали Extended IDE и LBA (Logical Block Addressing). Теперь секторы у нас нумеруются линейно от 0 до $2^{28} - 1$.
Контроллер занимался преобразованием в понятный себе адрес. Теперь ограничение стало 128 Гб.
h) ATA (Advanced Technology Attachment) - 3, ATAPI-X (ATA Packet Interface), SATA (Serial ATA). Теперь линейный адрес 48 бит, ограничение 48 Пбайт, живём.
Сначала это всё шумело и требовало много энергии, затем на SATA смогли разогнать до 1,5 Гбит/с.

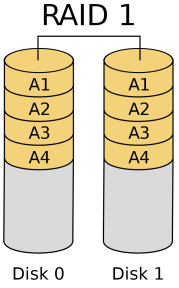
i) **RAID**.Что это? **Redundant Array of Inexpensive/Independent Disks (Избыточный массив недорогих/независимых дисков)**. Альтернатива - **SLED - Single Large Expensive Disk (Один большой дорогой диск)**.
Мы объединяем несколько дисков в один с помощью RAID-контроллера и с точки зрения внешнего мира получаем один цельный диск.
Зачем? Дешевле, быстрее, отказоустойчивее (последние два зависят от конкретной конфигурации).
(Все примеры расписаны на минимальном количестве дисков, но масштабируются на большее число)

i) **RAID 0**. Будем одну половину записывать/читать на один диск, вторую - на другой.



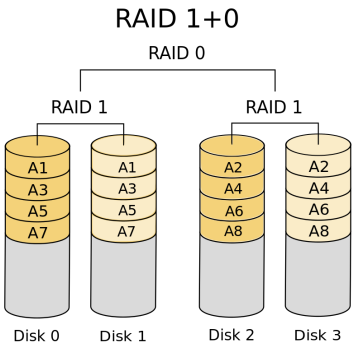
Если читаем/пишем большой кусок, идущий подряд, выигрываем.
Проигрываем в надёжности, ибо потеряем один диск - грохнется всё.

ii) **RAID 1**. Продублируем данные с одного диска на другой.

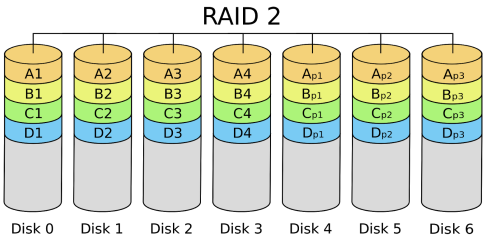


Мы теряем в объёме в два раза.
Выигрываем в надёжности, потому что потерять два диска с одинаковыми данными - очень редкий проигрыш.

iii) **RAID 1+0** aka **RAID 10**. Комбинация предыдущих. Возьмём каждый диск в RAID 0 и продублируем, как в RAID 1.

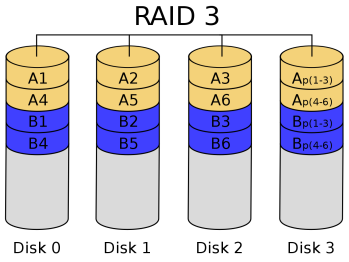


iv) **RAID 2**. Оперируем блоками или байтами, которые пишем на разные диски. Храним всё в коде Хемминга. Т.е. 1-й диск - под первый блок, 2-й - под второй и т.д.



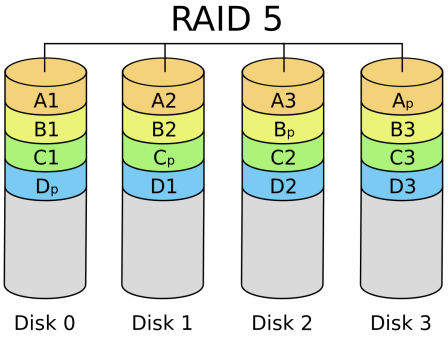
Исправляем ошибки с лёгкостью, но здесь нам уже нужно минимум 7 дисков.

v) **RAID 3/4**. Упрощённая версия RAID 2. Вместо кода Хемминга храним на последнем диске байт/блок чётности.



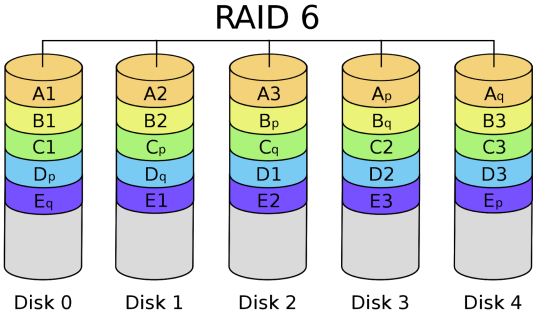
Хорошо и параллельно читаем. При записи постоянно теребонькаем последний диск. Требуется чуть больше времени на исправление ошибки.

vi) **RAID 5**. Распределим блоки чётности равномерно по всем дискам.



Те же плюшки, что и RAID 3/4, но теперь при записи распределяем теребоньканье по всем дискам -> диски в среднем нагружаются одинаково.

vii) **RAID 6**. RAID 5 с апгрейдом в виде кода Рида-Соломона.



j) **Оптические накопители**. Почти то же самое, что магнитные.
У нас есть спиралька на весь диск, она разбита на секторы. На спиральке есть дырки и площадки (“низины” и “плато”). Переход дырка-площадка трактуется как 1, площадка-дырка - 0 (там что-то с физикой, оптикой и лазером. Обратитесь к другому факультету).
Устройство секторов такое же, как у магнитных: преамбулы, коды коррекции, все дела.
Здесь мы скорость вращения меняем в зависимости от дорожки (чтобы ваше аниме/любимый альбомчик КиШа проигрывались с одинаковой скоростью).
Чтобы иметь возможность сделать диск перезаписываемым, добавим на него ещё несколько слоёв для выжигания. Тогда при перезаписи верхний слой стирается/сгорает, и мы работаем с чистым диском.
CD - обычный compact disk.
DVD - Digital Video Disk и Bluray - увеличили плотность записи, пошаманили с длиной волны лазера.

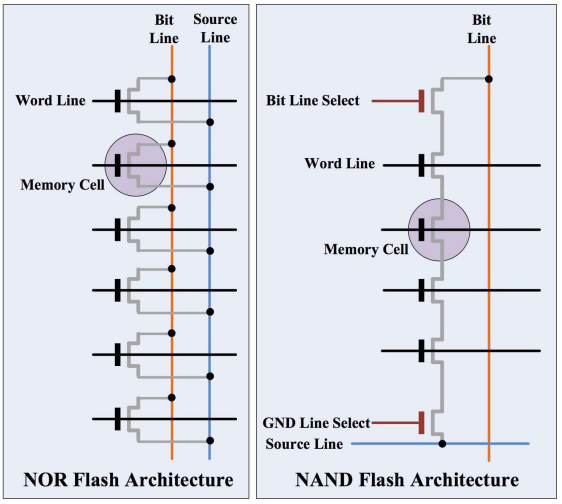
k) **Флеш-накопители** aka **SSD** и **флешки**.
<опять физика> Во флеш-накопителях используются транзисторы с плавающим затвором. Их отличие в том, что заряд на затворе сохраняется при отключении питания. Расплата за это - деградация транзистора при изменении заряда.
Заряд на затворе определяет напряжение, которое необходимо для включения транзистора. </опять физика>

l) Типы транзисторов:
   i) SLC (Single-level cell) - транзистор хранит два состояния (0 и 1), до 10^6 операций перезаписи. Почти нигде не используются (ходят слухи, что это проделки маркетологов и производителей).
   ii) MLC (Multi-level cell) - на самом деле Double-level. 4 состояния (храним два бита), до 10^5 операций. Работает медленнее.
   iii) TLC (Triple-level cell) - 8 состояний, до 3000 операций, ещё медленнее. Места занимает поменьше.
   iv) QLC (Quad-level cell) - 16 состояний, почти не используется.

m) В жизни чаще используют MLC и TLC.
Флеши имеют нулевое время поиска в сравнении с HDD и не погибают от перемещений и резких движений. В то же время заряд на транзисторе пропадает со временем. Поэтому флешки, если лежат без дела

несколько месяцев, бьются и могут потерять данные.
SSD - та же флешка, только больше и с контроллером, который распределяет нагрузку и в целом заботится о диске.
Чтобы перезаписать, нам нужно сначала везде поставить 0, а потом уже наши данные. Это действие занимает время, поэтому контроллер сначала пишет всё на свободное место в один блок, при удалении помечает невалидные секторы, потом, когда диск неактивен, переносит нужные файлы в другой блок и обнуляет предыдущий.

n) Существуют две архитектуры флеш-накопителей: **NOR** и **NAND**.



NOR: обычная матрица ячеек, у каждой ячейки есть своя пара проводов. Маленькая плотность ибо провода и всё такое, но выше скорость при случайном доступе. Используется в памяти микроконтроллеров.
NAND: <вот здесь я не понел> Здесь провода Bit и Source общие на какое-то количество ячеек (обычно на 8). То есть имеем конструкцию в виде трёхмерного массива.</вот здесь я не понел> Это всё компактно, но медленнее. Используется во флешках и SSD.

o) В завершение про SSD. Хоть там и мелькают весьма маленькие числа про ресурс флеш-накопителей, не стоит бояться. Даже если вы поставите SSD собирать логи Гугла, он продержится 2 месяца (в нормальной жизни вы даже далеко от такой нагрузки не будете -> на ваш век хватит).

11) Общие вопросы.

a) Препода зовут Мельников Роман Вячеславович.
b) Предмет - Архитектура ЭВМ.
c) На всякий случай, уровни абстракции ЭВМ:
   i) Physics - физический: электрончики, квантовая физика и боль.
   ii) Devices - устройства: полупроводники, транзисторы и конденсаторы, диоды.
   iii) Analog Circuits - аналоговые схемы: соединяем полупроводники вместе получаем (Могучего Рейнджера) схемы типа усилителей, которые на вход и выход получают непрерывный диапазон значений.

iv) Digital Circuits - цифровые схемы: воспринимают 2 дискретных уровня напряжения, на которых строятся логические вентили.
v) Logic - логический уровень: сложные схемы из вентилей, например, сумматоры, мультиплексоры, триггеры.
vi) Micro-architecture - микроархитектура: связывает логический и архитектурный уровни. Железки, которые уже достаточно сложные и реализуют команды из архитектуры.
vii) Architecture - архитектура: компьютер с точки зрения низкоуровневого программиста (не в смысле новичок, а в смысле компиляторами и драйверами занимается).
viii) Operating systems - ОСи: управляет низкоуровневыми операциями с железом, посредник между человеком и аппаратурой.
ix) Application software - приложения: высокоуровневый софт.

d) “Так исторически сложилось” - тоже ответ, но не всегда.
e) /*Философия на первом курсе маги - больно*/