

# Билеты к экзамену по Арх. ЭВМ (Первый семестр)

Created by Sabitov Kirill, Panasyuk Igor, Berezhnoy Akim

January 2023

## Содержание

Билеты к экзамену по Арх. ЭВМ (Первый семестр) .....	1
1. Физические основы. ....	3
1.1. Электрические сигналы .....	3
1.2. Полупроводники .....	3
1.3. Диоды .....	4
1.4. Конденсаторы .....	4
1.5. Транзисторы .....	4
1.6. Логические схемы на транзисторах .....	5
2. Представление чисел. ....	6
2.1. Представление чисел .....	6
2.2. Способы хранения целых чисел .....	6
2.3. Способы хранения нецелых чисел .....	9
3. Функциональные схемы. ....	11
3.1. Комбинационные схемы .....	11
3.2. Сумматоры .....	12
3.3. Последовательные схемы .....	13
3.4. Схемы для выполнения арифметических операций .....	15
4. Память. ....	16
4.1. Типы ячеек памяти .....	16
4.2. Организация ячеек памяти .....	17
5. Организация чипсетов. ....	18
5.1. Развитие двухмостовой архитектуры .....	18
5.2. Модели разделения памяти в мультипроцессорах (UMA/NUMA) .....	19
6. Оперативная память. ....	21
6.1. Чтение и запись в оперативную память .....	21
6.2. Тайминги для чтения .....	22
6.3. Историческая справка .....	22
6.4. Про модификации модулей памяти .....	22
6.5. Исторически сложившиеся стандарты оперативной памяти .....	23
7. Кэш-память. ....	24
7.1. Чтение из кэш памяти .....	24
7.2. Запись в кэш память .....	25
7.3. Способы размещения данных на уровнях кэша .....	25
7.4. Политики замещения данных .....	25
7.5. Хранение данных кэшей .....	25
7.6. Политики кэширования .....	25
7.7. Когерентность кэша .....	26
8. Виртуальная память. ....	27
9. ISA. ....	30
9.1. Архитектура фон Неймана .....	30
9.2. ISA (Instruction Set Architecture) .....	31
9.3. Assembler MIPS .....	33
10. Микроархитектуры. ....	34
10.1. Пример составных частей микроархитектуры .....	34

10.2. Однотактовый и многотактовый процессоры .....	34
10.3. Конвеер MIPS .....	35
10.4. Развитие конвеерной архитектуры .....	36
11. Внешние запоминающие устройства. ....	38
11.1. Внешние запоминающие устройства .....	38
11.2. Магнитные накопители .....	38
12. Параллелизм. ....	42
12.1. Вводная терминология .....	42
12.2. Виды параллельных архитектур (параллелизм по инструкциям и по данным) .....	42

# 1. Физические основы.

*Физические основы. Реализация логических вентилей с помощью транзисторов.*

## 1.1. Электрические сигналы

### Аналоговые сигналы

Аналоговый сигнал – непрерывный сигнал, изменяющийся во времени. Принимающий всевозможные значения из заданного промежутка. Все реальные сигналы являются аналоговыми, то есть непрерывны и постоянно изменяются.

### Дискретные (цифровые) сигналы

Значений аналоговых сигналов бесконечно много, в следствии чего их тяжело интерпретировать. Решение проблемы – введение абстракции в виде дискретных сигналов. Дискретный сигнал – прерывистый сигнал, который изменяясь по времени принимает значения только из списка возможных.

В реальности удобно вводить ряд констант для дискретизации аналогового сигнала:

1. Напряжение земли -  $V_{\text{gnd}}$  (обычно 0V)
2. Напряжение источника питания, то есть максимально возможное значение аналогового сигнала -  $V_{\text{dd}}$
3. Верхняя граница входного аналогового напряжения для получения 0 на выходе; то есть 0 на выходе, если  $V \in [0; V_l] - V_l$
4. Нижняя граница напряжения на входе для получения 1 на выходе, то есть 1 на выходе, если  $V \in [V_h; V_{\text{dd}}] - V_h$



Аналоговый сигнал



Цифровой сигнал

## 1.2. Полупроводники

Проводники – категории материалов обладающие рядом особых свойств, делающих их строительными блоками для почти любой цифровой системы. В нашем случае будем считать, что полупроводники состоят из кремния (Si) валентности 4, т.е. имеющего 4 электрона в оболочке, и примеси (атомов других химических элементов с отличной валентностью).

В зависимости от валентности примеси выделяют два типа полупроводников:

### Полупроводники *n*-типа

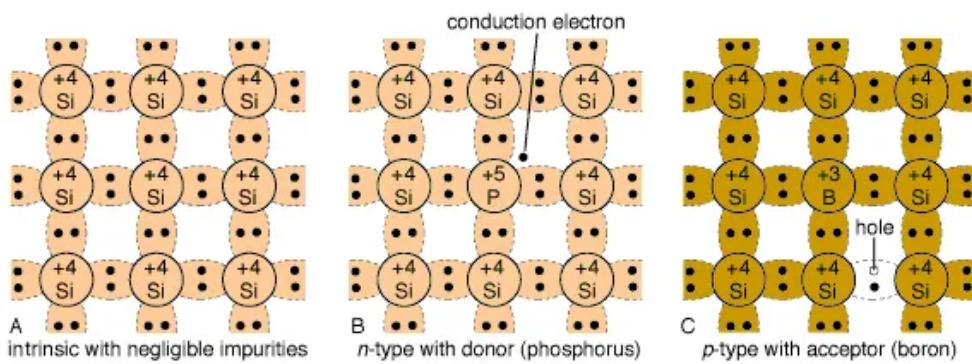
При добавлении примеси с валентностью больше, чем у кремния, образуется полупроводник *n*-типа (negative). Носителями заряда являются свободные электроны от атомов примеси, свободно летающие по кристаллической решетке. Тип связи – донорная.

Пример примеси с валентностью больше, чем у кремния - Фосфор с валентностью 5.

### Полупроводники *p*-типа

При добавлении примеси с валентностью меньше, чем у кремния, образуется полупроводник *p*-типа (positive). В месте связи атомов примеси с атомами кремния останется блюжающая по кристаллической решетке “дырка” (т.к. не хватает электрона для валентной связи). Эта “дырка” и является носителем заряда. Тип связи – акцепторная.

Пример примеси с валентностью меньше чем у кремния - Бор с валентностью 3.

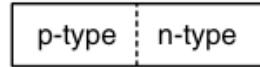


© 2004 Encyclopædia Britannica, Inc.

### 1.3. Диоды

Если соединить полупроводники *n* и *p* типов, то в месте их соприкосновения образуется *p-n* переход, при этом часть с полупроводником *n*-типа называют *катод*, с *p*-типа – *анод*. Такое явление лежит в работе прибора *диод*, имеющего нелинейную ВАХ (вольт-амперную характеристику).

Суть явления заключается в том, что на аноде образуется недостача электронов, он заряжен положительно; на катоде много свободных электронов, он заряжен отрицательно. В следствии этого образуется достаточно большая разность потенциалов. Тогда, если добавить внешнюю разность потенциалов, сонаправленно с внутренней (которая идет от анода к катоду), то ток пойдет, если противоправленно и внешняя разность потенциалов будет компенсирована внутренней. В случае противоправленной внешней связи, если она будет достаточно большой, то произойдет пробой.



anode      cathode



*p-n* переход и обозначение диода.

### 1.4. Конденсаторы

Иногда полезно умение контролировать время за которое проходит заряд и накапливать его. Для этого существуют *конденсаторы*. Конденсатор состоит из двух проводников и диэлектрика между ними. Если подать на один из проводников заряд  $V$ , то между проводниками образуется магнитное поле, через какое-то время на первом проводнике будет заряд  $Q$ , на другом  $-Q$ .

Вводят понятие емкости конденсатора  $C = \frac{Q}{V}$ , она прямопропорциональна площади проводников и обратно пропорциональна расстоянию между ними. Более высокая емкость означает, что электрическая схема будет работать медленнее и потребует для своего функционирования больше энергии.

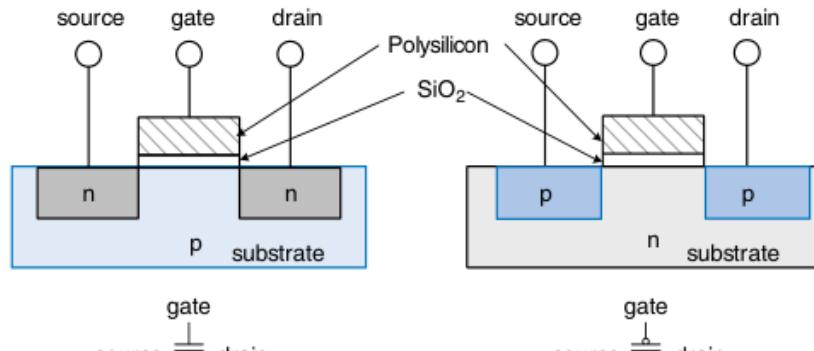


Обозначение конденсатора с емкостью  $C$ .

### 1.5. Транзисторы

*Транзистор* (в нашем случае именно *полевой*, *МОП-транзистор*, *MOSFET*, *metal-oxide-semiconductor field-effect transistor*) – несколько слоев проводников, полупроводников и диэлектриков. Принцип работы – пока не повзаимодействовать с *gate* (затвор), ток от *source* (исток) не дойдет до *drain* (сток). Все части транзистора закреплены на *substrate* (база). Примерный размер современных МОП-транзисторов –  $1\mu\text{m} = 10^{-6}\text{ m}$ .

Выделяют два типа МОП-транзисторов:

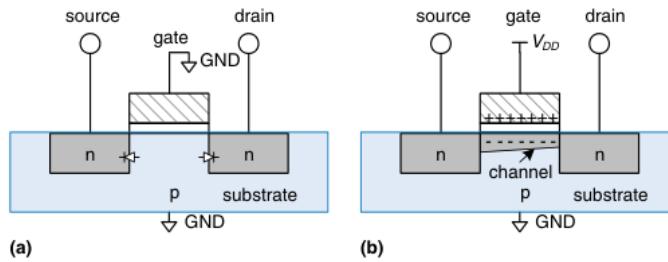


*nMOS* (слева) и *pMOS* (справа) транзисторы.

## n-p-n (nMOS)

Substrate соединен с землей, source с источником питания.

Если не подавать ток на gate, то диоды source to substrate и substrate to drain разнонаправлены, в следствии чего ток не идет. Если подать ток на gate, после зарядки конденсатора (который образован gate и substrate) образуется канал из отрицательных электронов, по которому заряд может идти от source на drain.



Разные состояния nMOS проводника (напряжения на gate) – 0V на gate слева,  $V_{dd}$  на gate справа.

Так же из-за ряда физических явлений nMOS транзисторы плохо передают 1, а pMOS плохо передают 0, что стоит учитывать при создании электросхем.

**Производство кристаллов для процессора.** Размещается substrate (база) – круглая подложка из кремния, полученная нарезанием кремниевого цилиндра, в нее имплантируются примеси соответствующего типа (валентность которых или ниже или выше валентности кремния). Таким образом на подложке образуется множество мельчайших транзисторов, затем подложка нарезается на готовые чипы.

## 1.6. Логические схемы на транзисторах

Тривиальные схемы для базовых логических (бинарных) операций:

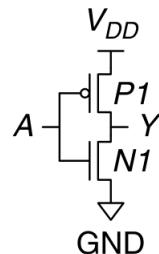


Схема для логического HE.

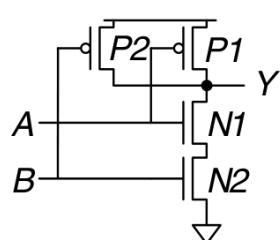


Схема для штриха Шеффера ↑, NAND.

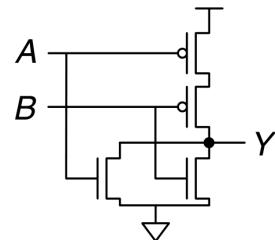


Схема для стрелки Пирса ↓, NOR.

Схемы для AND и OR можно было бы получить, поменяв тип транзисторов в схемах для NAND и NOR, но из-за эффекта плохой проводимости напряжений 0 и 1 соответствующими транзисторами на практике так делать нельзя. Решение – просто добавить NOT к NAND, NOR.

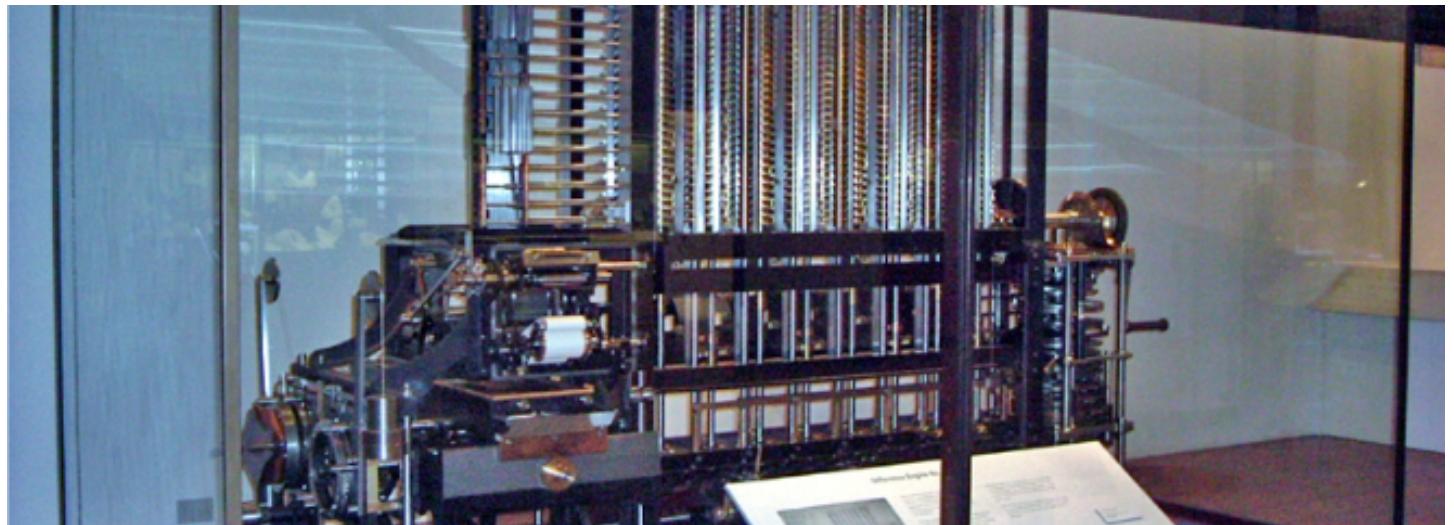
## 2. Представление чисел.

*Представление чисел. Способы хранения целых чисел: бит под знак, код со сдвигом, дополнение до 1, дополнение до 2). Способы хранения нецелых чисел: числа с фиксированной точкой, числа с плавающей точкой и стандарт IEEE 754 (как производить арифметические операции, специальные числа).*

### 2.1. Представление чисел

#### Историческая справка

Аналитическая машина Бэббиджа - первая в мире программируемая вычислительная машина. Конструкция разностной машины основывалась на использовании десятичной системы счисления.



Аналитическая машина Бэббиджа

#### Разновидности систем счисления

Числа представлены в двоичной системе счисления. Почему?

1. *Система счисления с основанием e.* Имеет максимальную плотность записи информации. Однако по историческим причинам она не получила сильного применения. Переход на другую систему счисления обошёлся бы очень дорого.
2. *Троичная система счисления.* Имеет плотность записи информации выше, чем у двоичной системы счисления. Однако по историческим причинам она не получила сильного применения. Переход на другую систему счисления обошёлся бы очень дорого.
3. *Десятичная система счисления.* Крайне неудобна из-за необходимости в огромном физическом сопровождении.

### 2.2. Способы хранения целых чисел

#### Прямой код (бит под знак)

При записи числа в прямом коде старший разряд является знаковым разрядом. Если его значение равно нулю, то представлено положительное число или положительный ноль, если единице, то представлено отрицательное число или отрицательный ноль. В остальных разрядах, которые называются цифровыми, записывается двоичное представление модуля числа. Таким способом в n-битовом типе данных можно представить диапазон чисел  $[-2^{n-1} + 1; 2^{n-1} - 1]$ .

#### Плюсы

1. В таком представлении числа легко конвертировать в десятичную запись и обратно.
2. Из-за того, что 0 обозначает +, коды положительных чисел относительно беззнакового кодирования остаются неизменными.
3. Однаковое количество положительных и отрицательных чисел.

#### Минусы

1. Выполнение арифметических операций с отрицательными числами требует усложнения архитектуры центрального процессора (например, для вычитания невозможно использовать сумматор, необходима отдельная схема для этого).
2. Существуют два нуля: -0 (100...000) и +0 (000...000), из-за чего усложняется арифметическое сравнение.

Из-за весьма существенных недостатков прямой код используется очень редко.

## Код со сдвигом

При использовании кода со сдвигом целочисленный отрезок от нуля до  $2^n$ , где  $n$  – количество бит, сдвигается влево на  $2^{n-1}$ , а затем получившиеся на этом отрезке числа последовательно кодируются в порядке возрастания кодами от 000...0 до 111...1. По сути, при таком кодировании к кодируемому числу прибавляют  $2^{n-1}$ , после чего переводят получившееся число в двоичную систему исчисления. Можно получить диапазон значений  $[-2^{n-1}; 2^{n-1} - 1]$ .

### Плюсы

1. Отсутствует проблема с двумя нулями.

### Минусы

1. При арифметических операциях нужно учитывать смещение, то есть проделывать на одно действие больше (например, после «обычного» сложения двух чисел у результата будет двойное смещение, одно из которых необходимо вычесть).
2. Ряд положительных и отрицательных чисел несимметричен.

*Из-за необходимости усложнять арифметические операции код со сдвигом для представления целых чисел используется не часто, но зато применяется для хранения порядка вещественного числа.*

$2^n - 1$	<b>011...111</b>
.....	.....
2	<b>000...010</b>
1	<b>000...001</b>
0	<b>000...000</b>
-0	<b>100...000</b>
-1	<b>100...001</b>
-2	<b>100...010</b>
.....	.....
$-(2^n - 1)$	<b>111...111</b>

Прямой код.

$2^n - 1$	<b>111...111</b>
.....	.....
2	<b>100...010</b>
1	<b>100...001</b>
0	<b>100...000</b>
-1	<b>011...111</b>
-2	<b>011...110</b>
.....	.....
$-(2^n - 1)$	<b>000...001</b>
$-2^n$	<b>000...000</b>

Код со сдвигом.

## Дополнение до единицы

В качестве альтернативы представления целых чисел может использоваться код с дополнением до единицы.

Алгоритм получения кода числа:

- Если число положительное, то в старший разряд, который является знаковым, записывается ноль, а далее записывается само число.
- Если число отрицательное, то код получается инвертированием представления модуля числа, таким образом, получается обратный код.
- Если число является нулем, то его можно представить двумя способами: +0 (000...000) или -0 (111...111).

Таким способом можно получить диапазон значений  $[-2^{n-1} + 1; 2^{n-1} - 1]$ .

### Плюсы

- Простое получение кода отрицательных чисел.
- Из-за того, что 0 обозначает +, коды положительных чисел относительно беззнакового кодирования остаются неизменными.
- Количество положительных чисел равно количеству отрицательных.

## Дополнение до двух

Чаще всего для представления отрицательных чисел используется код с дополнением до двух. Алгоритм получения кода числа:

- Если число неотрицательное, то в старший разряд записывается ноль, далее записывается само число.
- Если число отрицательное, то все биты модуля числа инвертируются, то есть все единицы меняются на нули, а нули — на единицы, к инвертированному числу прибавляется единица, далее к результату дописывается знаковый разряд, равный единице.

Для получения из дополнительного кода самого числа нужно инвертировать все разряды кода и прибавить к нему единицу. Таким образом, можно получить диапазон значений  $[-2^{n-1}; 2^{n-1} - 1]$ .

### Плюсы

- Возможность заменить арифметическую операцию вычитания операцией сложения и сделать операции сложения одинаковыми для знаковых и беззнаковых типов данных, что существенно упрощает архитектуру процессора и увеличивает его быстродействие.
- Нет проблемы двух нулей.

### Минусы

- Выполнение арифметических операций с отрицательными числами требует усложнения архитектуры центрального процессора.
- Существуют два нуля: +0 и -0.

### Минусы

- Ряд положительных и отрицательных чисел несимметричен, но это не так важно: с помощью дополнительного кода выполнены гораздо более важные вещи, желаемые от способа представления целых чисел.
- В отличие от сложения, числа в дополнительном коде нельзя сравнивать как беззнаковые, или вычтать без расширения разрядности.

Несмотря на недостатки, дополнение до двух в современных вычислительных системах используется чаще всего.

- Дополнительный код также удобно использовать для вычислений в длинной арифметике, особенно для операций сложения и вычитания.
- Для умножения и деления лучше всего использовать прямой код (бит под знак). Ибо выполнять умножения с числами в дополнительном коде не всегда оптимально.

$2^n - 1$	011...111	↑
...	.....	.....
2	000...010	↑
1	000...001	↑
0	000...000	↑
-0	111...111	↑
-1	111...110	↑
...	.....	.....
$-(2^n - 2)$	100...001	↑
$-(2^n - 1)$	100...000	↑

Дополнение до единицы

$2^n - 1$	011...111	↑
...	.....	.....
2	000...010	↑
1	000...001	↑
0	000...000	↑
-1	111...111	↑
-2	111...110	↑
...	.....	.....
$-(2^n - 1)$	100...001	↑
$-2^n$	100...000	↑

Дополнение до двух

## 2.3. Способы хранения нецелых чисел

### Числа с фиксированной точкой

В представлении нецелых чисел с фиксированной точкой число разбивается на три части:

1. Бит под знак.
2. Представление целой части ( $N$  бит).
3. Представление дробной части ( $M$  бит).

Числа  $M, N$  являются константами. Дапозон целой части чисел –  $[-2^N + 1, 2^N - 1]$ .

#### Плюсы

1. Имеем удобные арифметические операции.

#### Минусы

1. Фиксированное количество бит под целую и дробную части.
2. Непонятно, где заканчивается целая и начинается дробная части.
3. Чтобы получить отрицательные числа, необходимо использовать дополнение до двух или прямой код (бит под знак).

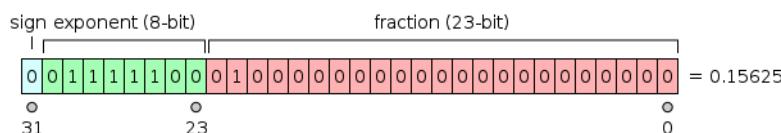
### Числа с плавающей точкой по стандарту IEEE754

Числа с плавающей точкой – один из возможных способов представления действительных чисел, который является компромиссом между точностью и диапазоном принимаемых значений, его можно считать аналогом экспоненциальной записи чисел, но только в памяти компьютера.

*Нормальной* называется форма представления числа, при которой абсолютное значение мантиссы десятичного числа находится на полуинтервале  $[0, 1)$

*Нормализованной* называется форма представления числа, при которой абсолютное значение мантиссы десятичного числа лежит на полуинтервале  $[1, 10)$ , а двоичного на полуинтервале  $[1, 2)$ .

В формате IEEE754 число с плавающей запятой представляется в виде набора битов, часть из которых кодирует собой мантиссу числа, другая часть – показатель степени, и ещё один бит используется для указания знака числа (0 – если число положительное, 1 – если число отрицательное). При этом экспонента записывается как целое число в коде со сдвигом, а мантисса – в нормализованном виде, своей дробной частью в двоичной системе счисления. *Например*, число  $-2435e9$ . 1 - знак, 10 - основание, 2345 - мантисса, 9 - экспонента.



При этом лишь некоторые из вещественных чисел могут быть представлены в памяти компьютера точным значением, в то время как остальные числа представляются приближёнными значениями.

Существуют различные форматы точности:

Название в IEEE 754	Название типа переменной в Си	Диапазон значений	Бит в мантиссе	Бит на переменную
Half precision	-	$6,10 \times 10^{-5} .. 65504$	11	16
Single precision	float	$-3,4 \times 10^{38} .. 3,4 \times 10^{38}$	23	32
Double precision	double	$-1,7 \times 10^{308} .. 1,7 \times 10^{308}$	53	64
Extended precision	На некоторых архитектурах (например в сопроцессоре Intel) long double	$-3,4 \times 10^{4932} .. 3,4 \times 10^{4932}$	65	80

## Особые значения чисел с плавающей точкой

**Ноль (со знаком).** В нормализованной форме невозможно представить ноль. Для его представления в стандарте зарезервированы специальные значения мантиссы и экспоненты.

Знак	Мантисса									
	Экспонента				Мантисса					
0/1	0	0	0	0	0	1,	0	0	0	0

 $= \pm 0$ 

**Бесконечность (со знаком).** Для приближения ответа к правильному при переполнении, в стандарте можно записать бесконечное значение. Так же, как и в случае с нолем, для этого используются специальные значение мантиссы и экспоненты.

Знак	Мантисса									
	Экспонента				Мантисса					
0/1	1	1	1	1	1	1,	0	0	0	0

 $= \pm \infty$ 

Бесконечное значение можно получить при переполнении или при делении ненулевого числа на ноль.

**Неопределенность.** В математике встречается понятие неопределенности, для этого предусмотрено псевдо-число, которое арифметическая операция может вернуть даже в случае ошибки.

Знак	Мантисса									
	Экспонента				Мантисса					
0/1	1	1	1	1	1	1,	0/1	0/1	0/1	0/1

 $= NaN$ 

## Арифметические операции с числами с плавающей точкой

**Сложение и вычитание.** Идея метода сложения и вычитания чисел с плавающей точкой заключается в приведении их к одному порядку. Сначала выбирается оптимальный порядок, затем мантиссы обоих чисел представляются в соответствии с новым порядком, затем над ними производится сложение/вычитание, мантисса результата округляется и, если нужно, результат приводится к нормализованной форме

**Умножение и деление.** Самыми простыми для восприятия арифметическими операциями над числами с плавающей запятой являются умножение и деление. Для того, чтобы умножить два вещественных числа в нормализованной форме необходимо перемножить их мантиссы, сложить порядки, округлить и нормализовать полученное число. Соответственно, чтобы произвести деление нужно разделить мантиссу делимого на мантиссу делителя и вычесть из порядка делимого порядок делителя. Затем точно так же округлить мантиссу результата и привести его к нормализованной форме.

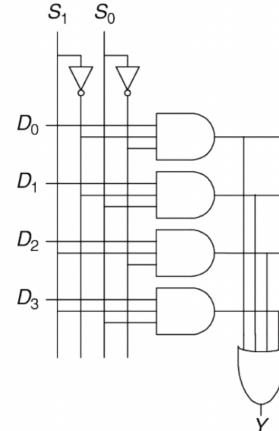
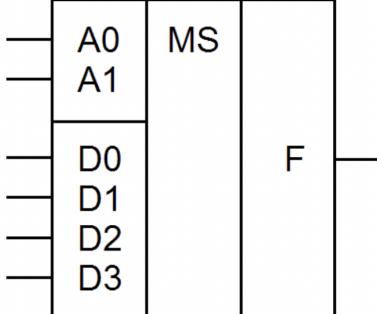
### 3. Функциональные схемы.

**Функциональные схемы.** Комбинационные схемы (мультиплексор, демультиплексор, дешифратор, сумматоры). Последовательные схемы (*RS-триггер*, *JK-триггер*, *T-триггер*, *D-триггер*). Схемы для выполнения арифметических операций (каскадный сумматор, АЛУ).

#### 3.1. Комбинационные схемы

##### Мультиплексор

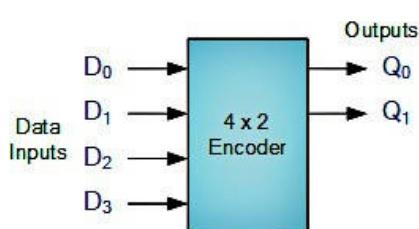
*Мультиплексор* — схема, позволяющая передавать сигнал с одного из  $2^n$  входов на единственных выходе. При этом выбор желаемого входа осуществляется подачей соответствующей комбинации управляющих сигналов. Для этого на вход также подаются  $n$  бит.



Внутренняя реализация для  $2^n$  значений использует  $n$  инверторов,  $2^n(n+1)$ -битных and'ов и единственный  $2^n$ -битный or.

##### Шифратор

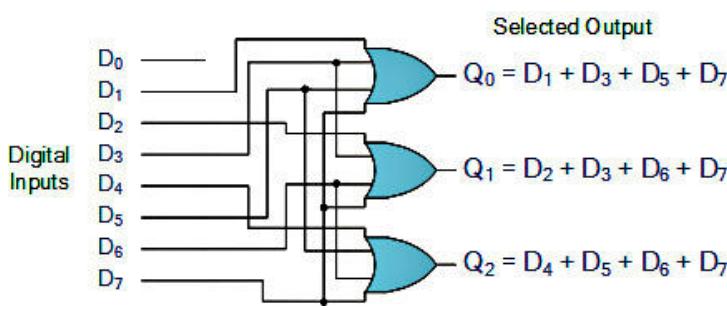
*Шифратор* — схема из функциональных элементов, которая позволяет по  $n$ -битному двоичному числу, в котором установлен ровно 1 бит, получить номер этого бита. Таким образом, при подаче сигнала на один из  $n$  входов (обязательно на один, не более) на выходе появляется двоичный код номера активного входа.



Обозначение шифратора на схемах.

D3	D2	D1	D0	Q1	Q0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
0	0	0	0	x	x

Таблица истинности шифратора.



Внутреннее устройство схемы может показаться запутанным. На самом деле, чтобы её реализовать, достаточно посмотреть на двоичную запись чисел от 0 до  $n - 1$ :

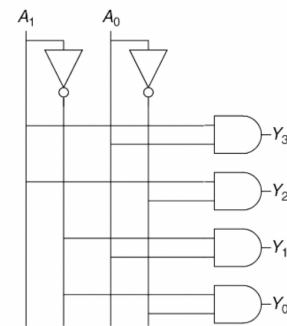
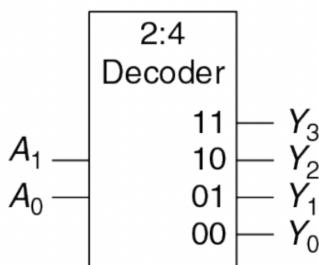
$$\begin{array}{ll} 0_{10} = 000_2 & 4_{10} = 100_2 \\ 1_{10} = 001_2 & 5_{10} = 101_2 \\ 2_{10} = 010_2 & 6_{10} = 110_2 \\ 3_{10} = 011_2 & 7_{10} = 111_2 \end{array}$$

- При подаче бита 0 не требуется устанавливать никакой бит, этот провод не участвует.
- При подаче бита 1 требуется установить только нулевой бит, поэтому провод идёт в  $or_0$ .
- При подаче бита 2 требуется установить только первый бит, поэтому провод идёт в  $or_1$ .
- При подаче бита 3 требуется установить биты 0 и 1, поэтому провод идёт в  $or_0$  и  $or_1$ .

Реализация более сложных шифраторов  $\forall n \in \mathbb{N}$  проводится по такому же принципу.

## Дешифратор

*Дешифратор* — схема из функциональных элементов, имеющая  $n$  входов и  $2^n$  выходов. Позволяет по  $n$ -битному двоичному числу установить единицу именно в тот выход, номеру которого соответствует двоичное число на входе.

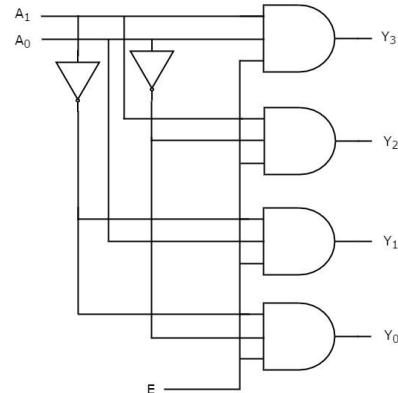
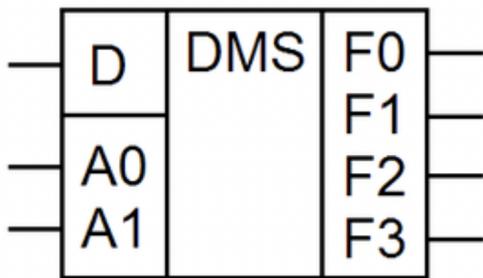


Внутренняя реализация для  $n$  входов использует  $n$  инверторов и  $2^n$   $n$ -битных and'ов. Легко заметить, что никакие 2 входа одновременно не могут быть выставлены как 1.

## Демультиплексор

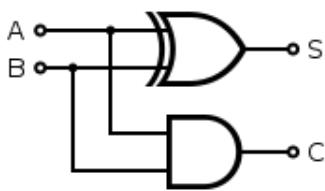
*Демультиплексор* — это логическое устройство, предназначенное для переключения сигнала с ровно одного информационного входа на один из информационных выходов. Таким образом, демультиплексор в функциональном отношении противоположен мультиплексору.

Схема демультиплексора легко строится, если вы ознакомились со схемой дешифратора. Достаточно в каждый из and'ов подключить бит, который мы передаём. Таким образом схема принимает вид, подозрительно напоминающий дешифратор:

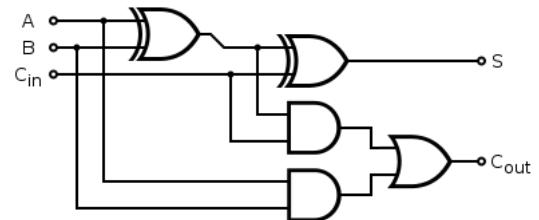


## 3.2. Сумматоры

### Частичный сумматор



### Полный сумматор



*Частичный сумматор* реализован просто: бит результата получается, как xor входных битов. Бит переноса же устанавливается только тогда, когда оба входных бита установлены.

*Полный сумматор* предназначен для суммирования длинных чисел (длина  $> 1$ ), так как в нём бит переноса может приходить из предыдущего разряда в следующий. Заметим в схеме аспекты:

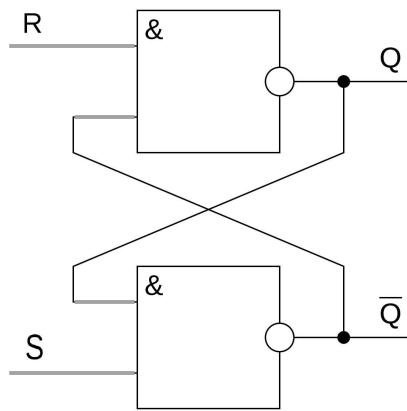
- Результатирующий бит числа получается, как  $xor$   $i$ -х битов с  $(i - 1)$ -м битом переполнения (тем, который пришел с предыдущего разряда).
- Бит переполнения устанавливается в тех случаях, если:
  - входные биты установлены ( $A = 1 \wedge B = 1$ );
  - входные биты различны ( $A = 1 \wedge B = 0$ )  $\vee$  ( $A = 0 \wedge B = 1$ ) и бит переполнения установлен ( $C = 1$ ).

Таким образом, полный сумматор корректен. Подключив несколько таких последовательно, мы можем складывать  $n$ -битный числа  $\forall n \in \mathbb{N}$ .

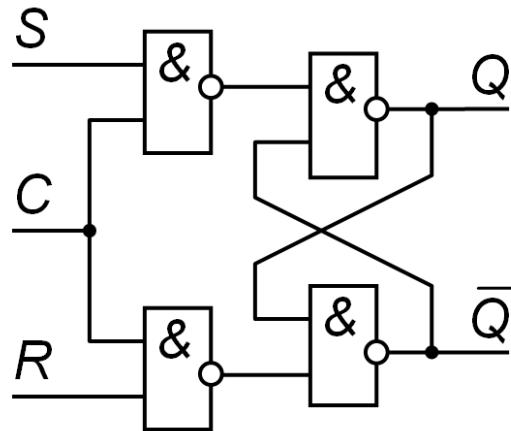
### 3.3. Последовательные схемы

Последовательной логической схемой называется схема с памятью.

#### RS-триггер



ассинхронный RS-триггер



синхронный RS-триггер

Принцип работы *RS-триггера* заключается в том, что он может сохранять своё предыдущее состояние пока оба входа неактивны и изменять его при подаче на один из входов единицы. При подаче единицы на оба входа состояние триггера вообще говоря неопределено.

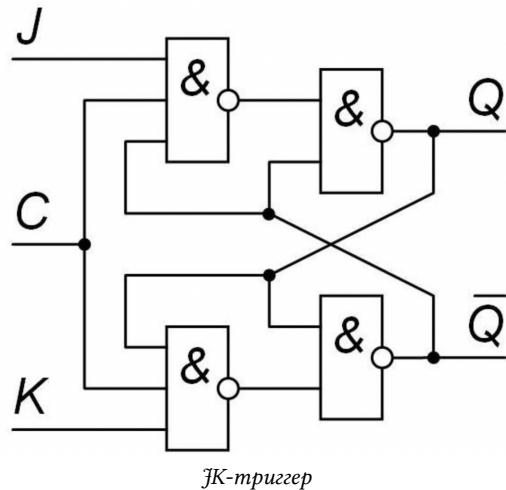
У *RS-триггера* есть 2 возможных состояния:

- Если подать на  $S$  (set) 1, а на  $R$  (reset) 0, то  $Q$  станет 1, а  $\neg Q$  станет 0.
- Если подать на  $S$  (set) 0, а на  $R$  (reset) 1, то  $Q$  станет 0, а  $\neg Q$  станет 1.

Такие значения сохранятся даже, когда  $R$  и  $S$  переключатся на 0.

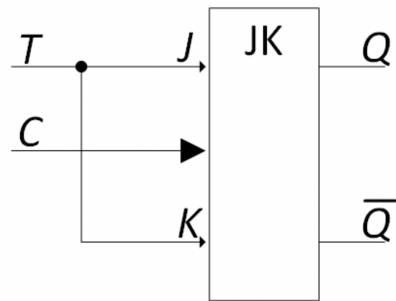
Существуют два способа реализации *RS-триггера*: *синхронный* и *ассинхронный*. Отличие у них лишь в том, что в случае *синхронного RS-триггера* значения на  $Q$  и  $\neg Q$  не изменяются, пока не будет подан сигнал синхронизации  $C$ , в то время как у *ассинхронного RS-триггера* значения меняются сразу при изменении  $R$  или  $S$ .

## JK-триггер



Работа *JK-триггера* практически совпадает с тем, как работает *синхронный RS-триггер*. Для того чтобы исключить запрещённое состояние(когда  $R = 1 \wedge S = 1$ ), его схема изменена таким образом, что при подаче двух единиц *JK-триггер* инвертирует хранимое значение:  $Q$  превращается в  $\neg Q$ , а  $\neg Q$  в  $Q$ .

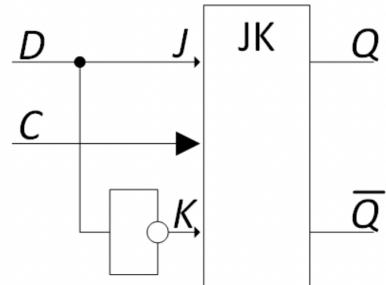
## T-триггер



*T-триггер*

*T-триггер* – это *JK-триггер*, в который на  $J$  и  $K$  подаются **только одинаковые** значения. Таким образом, мы можем только инвертировать хранимый бит.

## D-триггер



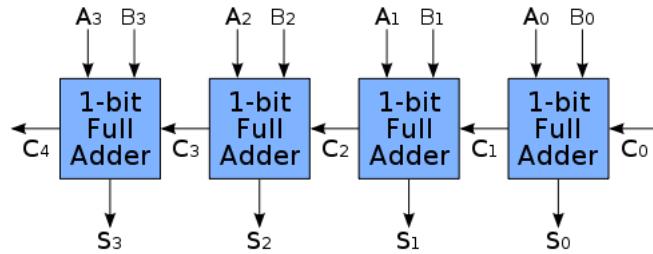
*D-триггер*

*D-триггер* – это *JK-триггер*, в который на  $J$  и  $K$  подаются **только различные** значения. Таким образом, мы можем только устанавливать хранимый бит нулём либо единицей.

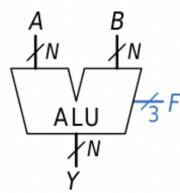
### 3.4. Схемы для выполнения арифметических операций

#### Каскадный сумматор

*Каскадный сумматор* – логическая схема, осуществляющая сложение многоразрядных двоичных чисел. Реализуется простой цепочкой полных однобитных сумматоров.

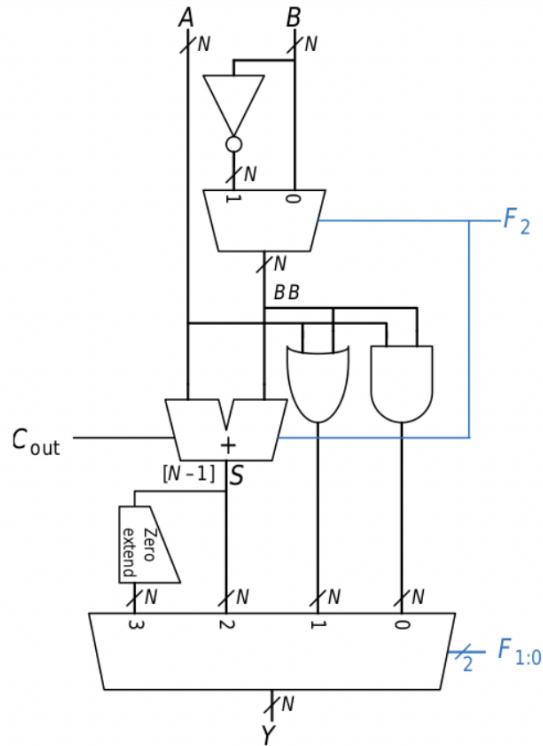


#### АЛУ (арифметико-логическое устройство)



$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\bar{B}$
101	A OR $\bar{B}$
110	A - B
111	SLT

*Арифметико-логическое устройство* – блок процессора, который служит для выполнения арифметических и логических преобразований (начиная от элементарных) над данными, называемыми в этом случае операндами.



Легко заметить, внутреннее устройство АЛУ элементарно и крайне понятно. Если вы по каким то причинам не понимаете, как работает данная логическая схема, авторы статьи коллективно рекомендуют вам пойти и написать свой 32-х разрядный многотактовый процессор на архитектуре MIPS, используя язык verilog .

## 4. Память.

Память (статическая и динамическая ячейки, их преимущества и недостатки).

Модули памяти отличаются типом ячеек и способом их организации.

### 4.1. Типы ячеек памяти

В современном мире существует два основных типа ячеек памяти – *статические* и *динамические*. Статические ячейки памяти имеют быстрый доступ к чтению и записи, обладают полным набором всех возможностей. Динамические ячейки памяти более медленные, но намного дешевле и имеют более компактную схему.

Статические ячейки памяти используются в кэш памяти и регистровых файлах, где не нужен большой объем, но важна скорость доступа. Динамические ячейки памяти из-за дешевизны и компактности используются в оперативной памяти (DDR\*).

#### Статический

Как уже было упомянуто, статические ячейки памяти достаточно громоздки и дороги в производстве. Ячейка состоит из 6 транзисторов. Транзисторы  $M_1, M_2$  и  $M_3, M_4$  – 2 разнонаправленных инвертора. Чтение состояния ячейки всегда доступно на  $BL, \overline{BL}$  (подав 1 на  $WL$ ). Для записи подаем необходимые значения на  $BL, \overline{BL}$ , и 1 на  $WL$ .

Состояние ячейки хранят два инвертора, оно стабильно пока подается напряжение на  $V_{dd}$  (ячейка энергозависима). Получение состояния возможно почти сразу после подачи сигнала на  $WL$ , что обеспечивает большую скорость доступа.

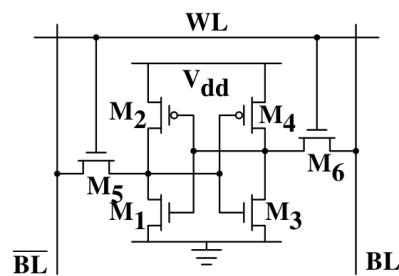


Схема статической ячейки памяти.

#### Динамический

Динамические ячейки памяти намного более простые и компактные. Состоят из транзистора и конденсатора.

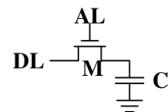


Схема динамической ячейки памяти.

Для чтения подается 1 на  $AL$ , что увеличит или уменьшит напряжение на  $DL$  (если увеличилось, то конденсатор был заряжен и состояние ячейки – 1, если уменьшилось, то разряжен, мы его начали заряжать, состояние – 0). Для записи значения ячейки необходимо подать соответствующий сигнал на  $DL$  и на  $AL$  на достаточное для зарядки/разрядки конденсатора время.

Состояние динамической ячейки памяти хранится в конденсаторе ( $C$ ), который немного разряжается после каждого чтения и с течением времени – ячейка памяти не стабильна. Решение – дозаряжать конденсатор после каждого чтения.

Время для чтения достаточно большое из-за необходимости периодически (примерно каждые 64мс) и после каждого чтения памяти заряжать конденсатор.

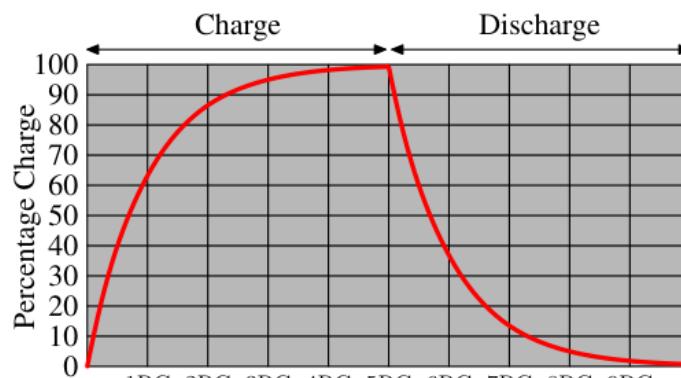
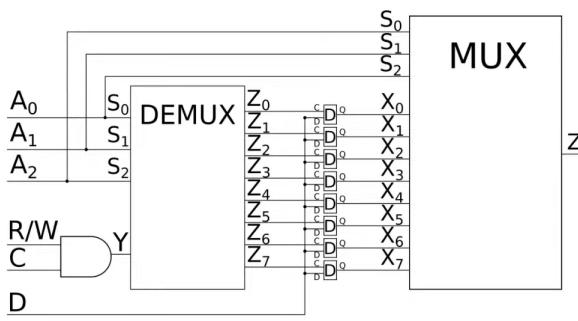


График  $Q(t)$  зарядки и разрядки конденсатора,  $Q_{\text{charge}} = Q_0 \left(1 - e^{-\frac{t}{RC}}\right)$ ,  $Q_{\text{discharge}} = Q_0 e^{-\frac{t}{RC}}$

## 4.2. Организация ячеек памяти

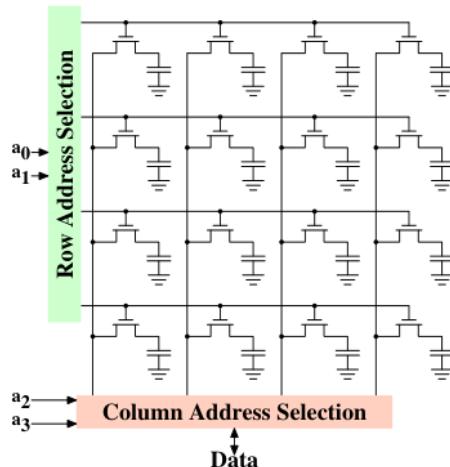
- Одиночная организация.** На каждую яческу памяти собственные контакты чтения/записи. Абсолютно не практична из-за количества проводов – на 4GiB ОЗУ их потребуется  $2^{32}$ .
- Линейная организация.** В линию располагаем ячейки, подсоединяя шину для записи к демультиплексору, для чтения к мультиплексору. Намного компактнее предыдущего, всего 23 контакта на 1GiB. Для доступа к  $2^N$  линиям нужен  $N$  to 1 DMUX/MUX, имеющий большие размеры (увеличивается экспоненциально). Также возникают проблемы с синхронизацией большого количества ячеек, расположенных линейно.



Пример. Схема игрушечного модуля памяти на D-триггерах. Непреемлем в жизни из-за того, что:

- сами ячейки памяти очень дороги в производстве (целый D-триггер – много транзисторов)
- доступ к ним происходит линейно через 1 мультиплексор и демультиплексор
- шина для передачи адреса чтения/записи очень длинная, отдельно соединена с мультиплексором.

- Матричная организация.** Располагаем ячейки в ряды и колонки, доступ происходит через 1 демультиплексор для кодирования ряда, 1 мультиплексор для кодирования колонки.



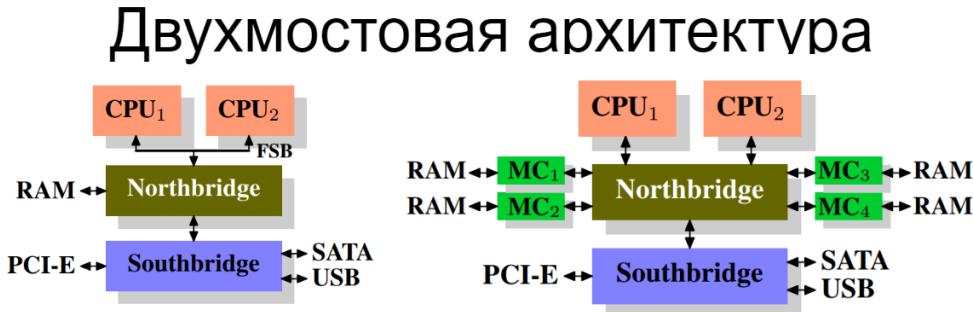
Пример организации 16 динамических ячеек памяти в матрицу.

На практике, безусловно, применяется матричный вариант из-за компактности.

## 5. Организация чипсетов.

*Организация чипсетов (двухмостовая архитектура и ее развитие, UMA/NUMA).*

### 5.1. Развитие двухмостовой архитектуры



*Пример двухмостовой архитектуры с различной реализацией шин CPU и контроллерами памяти*

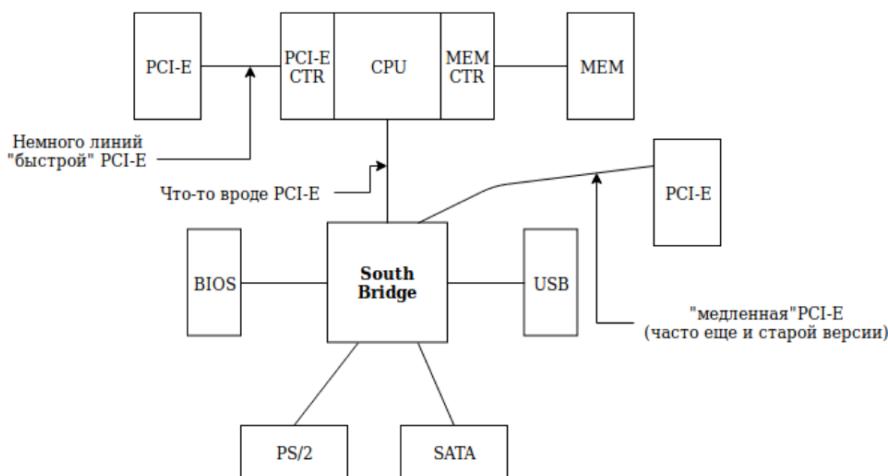
В двухмостовой архитектуре, в зависимости от конкретной реализации, процессоры либо каждый по отдельности подключены к северному мосту, либо подключены через общую шину (обычное подключение идет через шину FSB - front side bus). Северный мост в свою очередь соединен с южным мостом

#### Основные идеи двухмостовой архитектуры:

1. Вся быстрая периферия (все быстрые устройства, с которыми нужно работать процессору) подключается к северному мосту. Например, контроллеры RAM.
2. Все медленные устройства подключаются к южному мосту. Например, внешние жесткие диски, USB устройства, различные устройства, которые могут быть подключены с помощью шины PCI-Express.

#### Основные недостатки двухмостовой архитектуры:

1. В общем случае внешним устройствам для того, чтобы взаимодействовать с оперативной памятью, нужно реализовывать запросы через процессор. Шина между процессором и северным мостом становится так называемым узким местом (bottleneck). Однако, данный недостаток в некоторых реализациях был компенсирован с помощью DMA (Direct memory access). Данный подход несет в себе идею предоставления устройствам периферии прямой доступ к оперативной памяти без отправки запросов в процессор.
2. В общем случае двухмостовой архитектуры существует всего один канал доступа к оперативной памяти. Таким образом, например, нескольким ядрам процессора приходится поочередно отправлять запросы к оперативной памяти, что уменьшает производительность. Однако, данный недостаток в некоторых реализациях был компенсирован выносом контроллера оперативной памяти из северного моста наружу и добавлением нескольких контроллеров оперативной памяти. Таким образом, появилась возможность к различным модулям оперативной памяти обращаться параллельно, увеличив производительность.

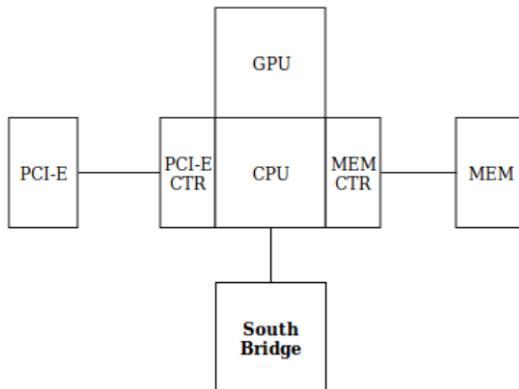


*Пример двухмостовой архитектуры с контроллером памяти и PCI-E на кристалле процессора.*

Впоследствии было замечено, что PCI-E устройства также достаточно быстрые и взаимодействие с ними через южный мост снижает производительность. Например, через PCI-E можно подключать дорогостоящие видеокарты или быстрые SSD-диски. Поэтому контроллер для взаимодействия с устройствами по шине PCI-E был перенесён на северный мост, а впоследствии на кристалл процессора.

После чего было замечено, что память становится все быстрее и работать с ней нужно все больше, поэтому контроллер памяти с северного моста был перенесён на кристалл процессора.

Таким образом, на кристалле процессора находятся контроллеры для взаимодействия с самой быстрой периферией (RAM и PCI-E), северный мост потерял надобность и был удален.



*Приблизительная архитектура современных портативных девайсов (SoC).*

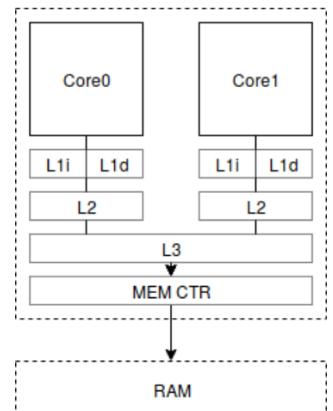
После чего пришла эра портативной электроники. Графические ускорители было решено расположить также на кристалле процессора. Таким образом, конечным этапом развития двухмостовой архитектуры является SoC (System on crystal). Эта архитектура подразумевает под собой расположение всех контроллеров периферии на кристалле процессора. Этот подход является типичным практический для всей портативной электроники.

## 5.2. Модели разделения памяти в мультипроцессорах (UMA/NUMA)

### UMA (Uniform Unix Access / Однородный доступ к памяти)

Система UMA (Uniform Memory Access) - это архитектура с общей памятью для многопроцессорных систем. В этой модели используется единственная память, к которой обращаются все процессоры представленной многопроцессорной системы с помощью межсоединительной сети. Каждый процессор имеет равное время доступа к памяти (задержка) и скорость доступа. Он может использовать либо одну шину, несколько шин или коммутатор.

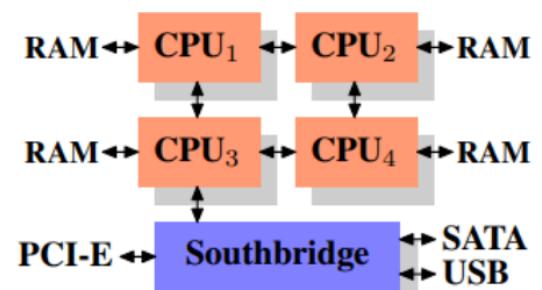
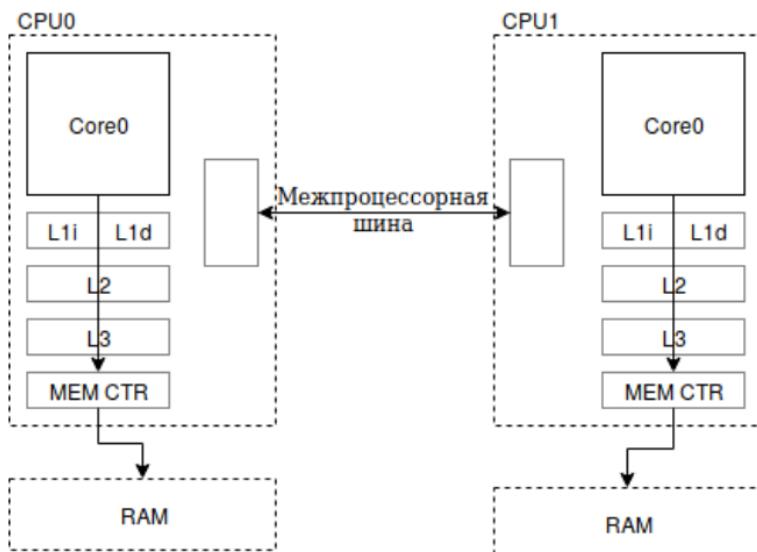
Данный подход хорошо показывает себя в работе, если количество вычислительных ядер находится в среднем диапазоне. Однако, при большом количестве вычислительных ядер данный подход показывает себя хуже, поскольку контролировать запросы к общей памяти в такой ситуации становится весьма затруднительно.



### NUMA (Non-Uniform Memory Access / Неоднородный доступ к памяти)

NUMA также является многопроцессорной моделью, в которой каждый процессор связан с выделенной памятью. Однако эти небольшие части памяти объединяются в единое адресное пространство. Время доступа к памяти зависит от расстояния, на котором расположен процессор, что означает изменение времени доступа к памяти. Это позволяет получить доступ к любой ячейке памяти, используя физический адрес, однако время доступа при этом получается неоднородным.

# NUMA



У каждого процессора есть «своя» оперативная память, наиболее близкая к нему. Чтобы обращаться с другой памятью, нужно идти к другим процессорам, таким образом время доступа получается неоднородное

Данный подход хорошо показывает себя при большом количестве вычислителей, каждый из которых реализует независимые вычисления. Хорошим примером являются удалённые сервера. Сервер разделяется на блоки, каждый из которых является практически независимым. Таким образом одновременно работает множество независимых процессов.

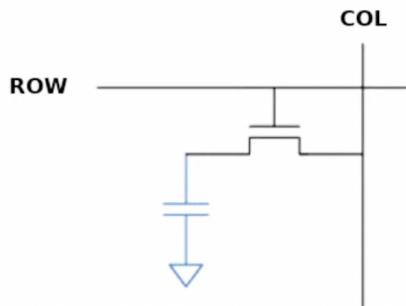
## 6. Оперативная память.

Оперативная память (процесс чтения и записи, тайминги для чтения, развитие оперативной памяти).

### 6.1. Чтение и запись в оперативную память

#### Внутреннее устройство

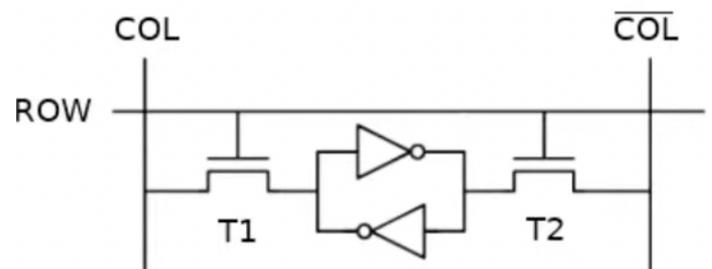
Для начала вспомним, как выглядит оперативная память.



DRAM (Dynamic Random Access Memory)

Динамическая память имеет:

- 1 n-mos транзистор
- 1 конденсатор



SRAM (Static Random Access Memory)

Статичнская память имеет:

- 2 n-mos транзистора
- 2 инвертора (ещё 2 n-mos и 2 p-mos)

#### Чтение

##### Процесс чтения из DRAM:

1. Подаём на COL 0.5 от нормального напряжения;
2. Подаём 1 на ROW (открываем n-mos транзистор);
3. Производим замер:
  - если напряжение стало  $0.5 - \varepsilon$ , значит хранился 0;
  - если напряжение стало  $0.5 + \varepsilon$ , значит хранился 1.

##### Процесс чтения из SRAM:

1. Подаём на COL и  $\overline{COL}$  нормальное напряжение;
2. Подаём 1 на ROW (открываем n-mos транзисторы);
3. Смотрим напряжение на COL и  $\overline{COL}$ :
  - если  $V_{COL} < V_{\overline{COL}}$ , значит в ячейке был записан 0;
  - если  $V_{COL} > V_{\overline{COL}}$ , значит в ячейке был записан 1.

#### Запись

Что если мы хотим записать значение  $x \in \{0, 1\}$  в ячейку?

##### Процесс записи в DRAM:

1. Подаём на COL x;
2. Подаём 1 на ROW (открываем n-mos транзистор);
3. Ждём тайминг зарядки конденсатора.

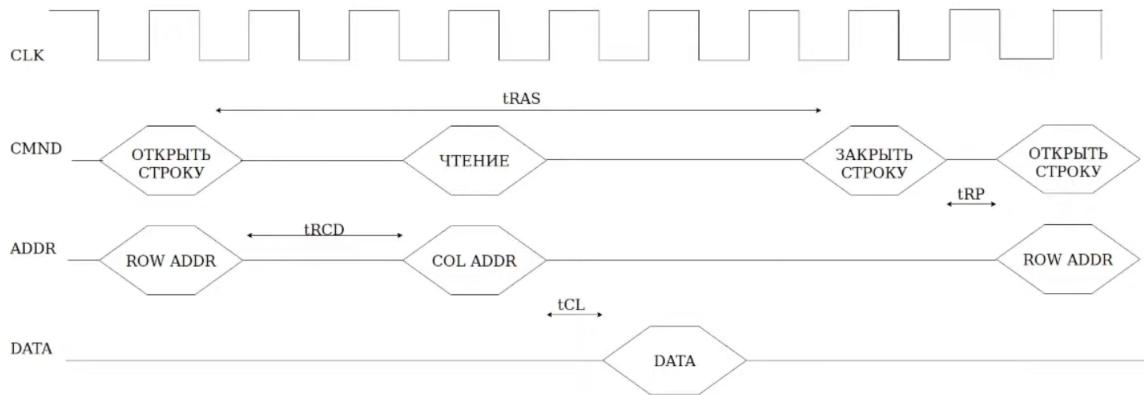
##### Процесс записи в SRAM:

1. Подаём на COL x, а на  $\overline{COL}$   $\neg x$ ;
2. Подаём 1 на ROW (открываем n-mos транзисторы);

Тем самым мы фактически либо заземляем значение на  $Q$  и подаём напряжение на  $\overline{Q}$ , если хотим записать 0, либо подаём напряжение на  $Q$  и заземляем на  $\overline{Q}$ , если хотим записать 1.

## 6.2. Тайминги для чтения

Тайминг оперативной памяти – временная задержка сигнала при работе динамической оперативной памяти. Измеряются в тактах тактового генератора. От них в значительной степени зависит пропускная способность участка «процессор-память» и задержки чтения данных из памяти и, как следствие, быстродействие системы.



- tRAS (Row Active Strobe) Минимальное время между открытием и закрытием строки.
- tRCD (RAS to CAS Delay) Число тактов между открытием строки и доступом к столбцам в ней.
- tCL (CAS Latency) Время, требуемое на чтение бита из памяти, когда нужная строка уже открыта.
- tRP (Row Precharge) Время перезарядки конденсаторов после того, как мы использовали строку.

### Latency и Throughput

Latency (скорость доступа) – время от подачи запроса до начала получения данных. (tRCD + tCL).

Throughput (пропускная способность) – время между получением двух порций данных. (tRAS + tRP).

## 6.3. Историческая справка

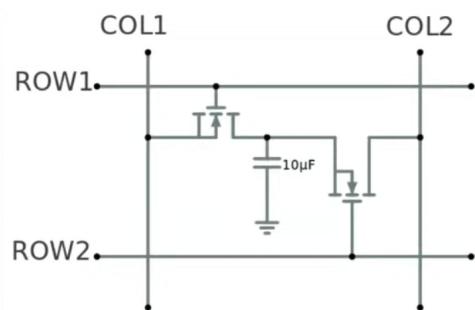
«Так исторически сложилось, что до какого-то момента скорость доступа получалась увеличивать довольно неплохо, но в какой-то момент улучшение скорости доступа почти полностью остановилось и после этого весь прогресс стремится к скорости передачи данных для какого-то конкретного устройства. Есть не очень много способов, чтобы увеличивать скорость доступа, но есть огромное количество ухищрений и техник, чтобы увеличивать скорость передачи данных.»

Роман Мельников ©

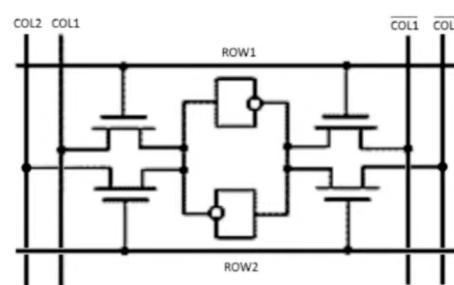
## 6.4. Про модификации модулей памяти

### Многопортовая ячейка памяти

Отличием многопортовой ячейки памяти от однопортовой является то, что у каждой ячейки есть 2 набора проводов. Это позволяет одновременно производить операции с двумя ячейками памяти (правда с оговоркой: они должны быть не из одной строки).



Двухпортовая DRAM



Двухпортовая SRAM

### Многобанковая ячейка памяти

В многобанковых ячейках памяти, RAM разделена на отдельные независимые банки. При обращении в разные банки, тк они являются независимыми и при считывании ячейки (открыть строчку, задать столбец, закрыть строку, перезарядить конденсатор) на другие банки не оказывается никакое влияние, мы можем производить чтение не дожидаясь перезарядки конденсаторов. Таким образом, такой подход выходит выгоднее, когда у нас последовательные запросы обращаются в разные участки памяти(банки).

## **Синхронная и асинхронная память**

**Ассинхронная память** — когда контроллер и модуль памяти взаимодействуют не согласованно(может быть, даже работают с разной частотой).

**Синхронная память** — контроллер и модуль памяти синхронизированы, частота одинаковая.

Раньше память была асинхронна, но сейчас (со второй половины 90-х) инженеры научились нормально собирать всё и согласовывать ячейки с контроллером.

## **6.5. Исторически сложившиеся стандарты оперативной памяти**

### **1. FPM DRAM (начало 90-х)**

- Не закрываем строку и не перезаряжаем конденсаторы, если запросы идут последовательно в одну и ту же строку, но в разные столбцы. Например, выполняется последовательный кусок кода, или происходит итерация по массиву;
- Ассинхронна.

### **2. EDO DRAM (первая половина 90-х)**

- Заведём дополнительный буфер, в котором мы храним адрес столбца, к которому мы сейчас обращаемся. Таким образом мы можем передавать адрес столбца для следующего запроса параллельно с тем, как мы получаем данные от предыдущего запроса.
- Асинхронна.

### **3. BEDO DRAM (всё ещё первая половина 90-х)**

- Будем вместе с запрошенной ячейкой отдавать данные сразу из 4-х следующих ячеек. Это опять же даёт нам ускорение, так как подавляющее большинство данных лежат в памяти последовательно и читаются последовательно.
- Адрес должен быть кратен 4.
- Всё ещё асинхронна.

### **4. SDRAM (вторая половина 90-х)**

- Наконец-то контроллер и модуль памяти стали синхронизированы. За счёт этого получилось значительно увеличить частоту.
- Шина взаимодействия была расширена до 64 бит.
- Начали появляться банки памяти(2-4 шт).

### **5. DDR (Double Data Rate — удвоенная скорость передачи данных, 1998 год)**

- Внутренняя шина была расширена в 2 раза.
- Передача данных происходит и на фронте, и на спаде сигнала синхронизации. Благодаря этому частота «выросла» в 2 раза.

### **6. DDR2 (2003 год)**

- Внутренняя шина расширена ещё в 2 раза.
- Число банков увеличено до 8.
- Частота внешней шины увеличена в 2 раза.

### **7. DDR3 (2007 год)**

- Расширили внутреннюю шину ещё в 2 раза.
- Увеличили частоту внешней шины ещё в 2 раза.
- Снизили напряжение питания до 1.5 V.

### **8. DDR4 (2014 год)**

- Раенирили внутреннюю шину еще в 2 раза.
- Уменьшили напряжение питания до 1.2 V.
- Появились группы банков.
- Увеличили число банков до 16.

### **9. DDR5 (2020 год)**

- Уменьшили напряжение питания до 1.1 V.
- Увеличили чисто банков до 32.
- Количество данных, отдаваемых подряд (burth length) увеличили вдвое.
- У DIMM два независимых канала по 32 бита.

## 7. Кэш-память.

**Кэш-память (Как устроена? Почему кэш-промахи не случаются часто?, инклюзивность/эксклюзивность, политики замещения, виды ассоциативности, протоколы когерентности).**

В современном мире скорость доступа к памяти значительно медленее скорости работы процессора. Решением проблемы стал концепт кэш-памяти – добавляем какое-то количество статической памяти рядом с процессором, в которой храниться копия каких-то данных из оперативной памяти. Менеджмент этой памяти ложиться на ОС, программист не имеет интерфейса для прямого взаимодействия с ней (проверить закэшированы какие-то данные или нет).

Доступ процессора к оперативной памяти происходит через кэш, а не напрямую через шину. В момент доступа происходит проверка на наличие необходимых данных в кэше, если данных нет в кэше то происходит *cache miss*.

кэш делится на несколько уровней:

1. Самый маленький, порядка 32KiB, время обращения ~4 такта. Чаще всего разделяется на 2 части – отдельно для данных и для инструкций.
2. Размер ~500KiB, время обращения ~20 тактов
3. Размер ~10MiB, время доступа ~40 тактов.

Как правило кэш 1 и 2 уровней собственные для каждого ядра процессора, кэш 3 уровня общий для всех ядер.

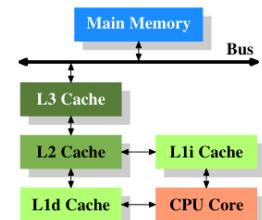
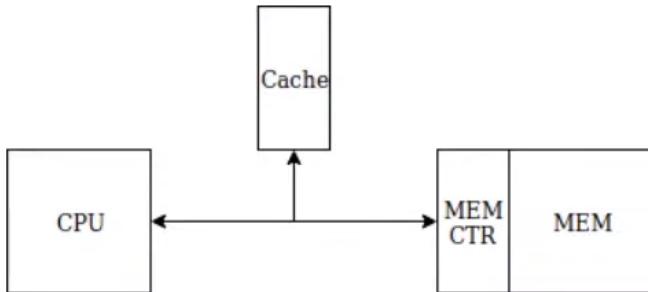


Иллюстрация организация доступа процессора к оперативной памяти.

### 7.1. Чтение из кэш памяти

#### Look-aside

При чтении оперативной памяти запрос одновременно отправляется так же в кэш память, если данные есть в кэше, то быстро отвечают на запрос, игнорируя данные из оперативной памяти.



#### Плюсы:

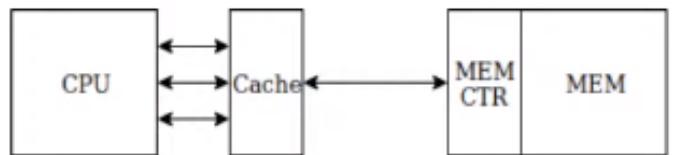
- Быстрый доступ к незакэшированным данным – не тратится время на проверку всего кэша (время сравнимо с временем доступа к оперативной памяти).

#### Минусы:

- Шина доступа к оперативной памяти забивается ненужными данными.
- кэш и оперативная память подключены через одну шину (достаточно длинную, из-за чего она не может работать на слишком большой частоте).

#### Look-through

Сначала ищем данные только в кэше, если не нашли их, то только тогда отправляем запрос к оперативной памяти.



#### Плюсы:

- Шина не забита ненужными данными.
- Есть возможность использования разных шин для обмена данными между процессором и кэшем, кэшем и оперативной памятью (первая короткая, может работать на большей частоте).

#### Минусы:

- Время ответа на запрос получения данных может быть до двух раз больше.

## 7.2. Запись в кэш память

### Write-through

Одновременно отправляем два паралельных запроса на запись и в кэш, и в оперативную память.

#### Плюсы:

- Легко реализовать в железе.

#### Минусы:

- Постоянное перезаписывание данных при кэшировании, что забивает шину.

### Write-back

Сначала данные записываются в кэш, потом при необходимости в оперативную. Производительность данного подхода зависит от реализации конкретного контроллера, реализованного хардварно.

#### Плюсы:

- Нет лишних запросов к оперативной памяти.

#### Минусы:

- Сложнее реализовать, контроллер занимает какое-то место.

## 7.3. Способы размещения данных на уровнях кэша

### Inclusive cache

В кэше более высокого уровня храниться копия данных из кэша более низкого уровня. При доступе данные ищутся в кэше от низкого до более высокого уровней, при нахождении на уровне  $n$ , данные записываются на уровня  $[1, n]$ .

При освобождении данные просто выкидываются из самого высокого уровня, в более низких остается копия этих данных.

### Exclusive cache

Данные хранятся эксклюзивно на каждом уровне. При получении данных из уровня  $n$ , они переносятся на первый, чтение происходит всегда из кэша первого уровня.

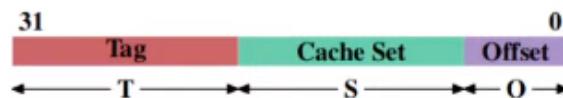
При необходимости освободить место на каком-то уровне, они переносятся на более низкий уровень.

## 7.4. Политики замещения данных

- **Random replacement** (ARM CPUs). При необходимости перезаписать данные, перезаписывается случайный фрагмент кэш-памяти. Очень просто в реализации, не требуется сложных контроллеров, работает не слишком плохо.
- **LRU, Least Recently Used**. Храним возраст для каждой строчки кэша (при взаимодействии со строкой ее возраст становится самым маленьким), выкидываются самые «старые».
- **TLRU, Time Aware LRU**. Помимо возраста храним время хранения, если оно больше каких-то констант, то данные выкидываются.
- **LFU, Least Frequently Used**. Данные выкидываются на основе частоты использования.
- **FIFO/LIFO**. Выкидываем на основе порядка добавления.
- Всевозможные комбинации вышеприведенных политик.

## 7.5. Хранение данных кэшей

Помимо данных в кэше необходимо хранить адрес их расположения в оперативной памяти. Адрессация по байтам невыгодна (влишком много места), адресуются блоки по 64 байта с дополнительным смещением внутри одного блока.



Хранение блока данных в кэше, tag - адрес блока в оперативной памяти.

## 7.6. Политики кэширования

Способы проверить закэшированы какие-то данные.

### Полная ассоциативность

В любой линии кэш памяти может храниться любая линия оперативной памяти. В таком случае если в кэш памяти  $n$  линий, то проверка наличия линии оперативной памяти с заданным тегом за  $O(n)$ .

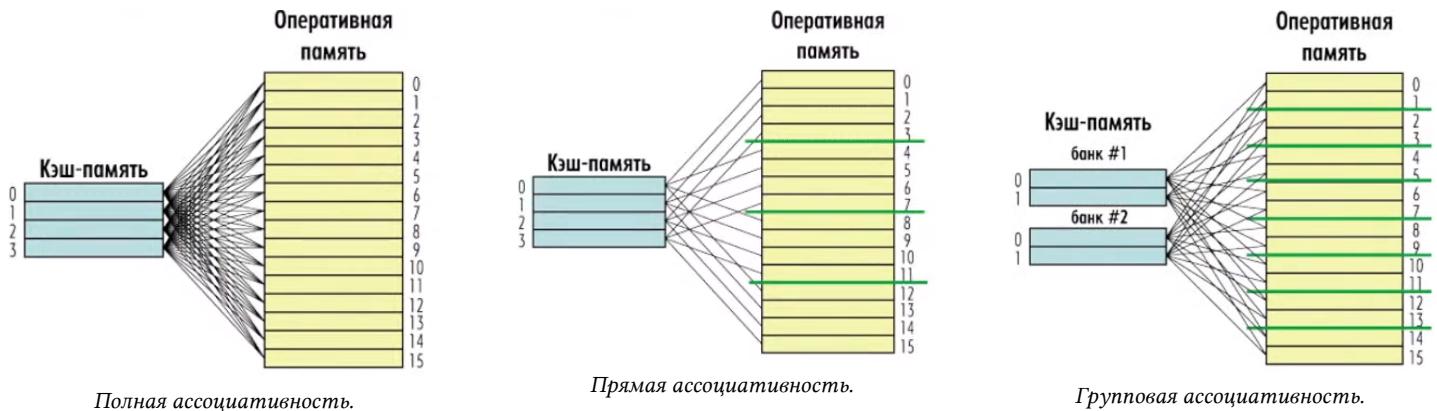
### Прямая ассоциативность (прямое отображение)

Предполагается, что 1 линия оперативной памяти может быть отображена только в одну линию кэш памяти. В таком случае оперативная память разбивается на блоки размера  $C$ , время проверки –  $O(1)$ .

Возникает проблема. Если активно используются какие-то конкретные линии из одного блока, то они будут часто перезаписывать друг-друга. В следствии этого второй способ может не дать существенного прироста, по сравнению с первым.

### Групповая ассоциативность

Синтез первых двух подходов. Вся память делиться на банки (группы), внутри которых реализуется полная ассоциативность.



### 7.7. Когерентность кэша

При использовании многовдерных/многопроцессорных систем может случиться несогласованность кэша, в момент когда кэш 1-2 уровней одного из ядер изменился, но изменения не были перенесены в оперативную.

### Протоколы когерентности MSI

Каждая кэш линия каждого кэша (т.е. свое состояние на каждое ядро) имеет одно из трех состояний:

- **M (Modified)** – линия была изменена, но изменение не было перенесено в оперативную память, во время этого состояния можно читать и писать данные. Гарантируется, что изменения будут перенесены в оперативную память, состояние из **M** измениться на **S**. Для конкретного тега оперативной памяти ровно в одном кэше может быть состояние **M**.
- **S (Shared)** – кэш линия не изменена и присутствует еще в хотя бы одном кэше, данные можно свободно читать, для изменения требуется смена состояния на **M**.
- **I (Invalid)** – кэш линия не присутствует в конкретном кэше или не валдина (данные не согласованы с другими кэшами). Для чтения необходимо получить данные из оперативной памяти.

### Протокол когерентности MESI

Новое состояние - **E (Exclusive)** – данная данные с таким тегом есть только в одном кэше, можно свободно менять. Состояние активируется, в случае когда данные подгружаются из оперативной памяти (переход из **I**) и они не представлены ни в каком другом кэше. Если при подгрузке данные есть в кэш линии с состоянием **E**, то линии в обоих кэшах становятся **S**.

### Протокол когерентности MOESI

Новое состояние - **O (Owned)** – при изменении данные броадкастятся в другие кэши напрямую (в обход оперативной памяти), что бы линии с таким же тегом в состоянии **S** поменяли данные.

### Протокол когерентности MESIF

Новое состояние - **F (Forward)** – линия с таким состоянием может отвечать на запросы о наличии данных с определенным тегом в кэше. Благодаря этому в других кэшах можно перейти из **I** в **S** без доступа к оперативной памяти. Состояние **F** переходит к последнему кэшу, который запрашивал данные с нужным тегом (предыдущий владелец меняет состояние на **S**).

## 8. Виртуальная память.

Виртуальная память (Зачем нужна? Что такое страница, Page Table, TLB? Зачем нужны многоуровневые таблицы страниц?).

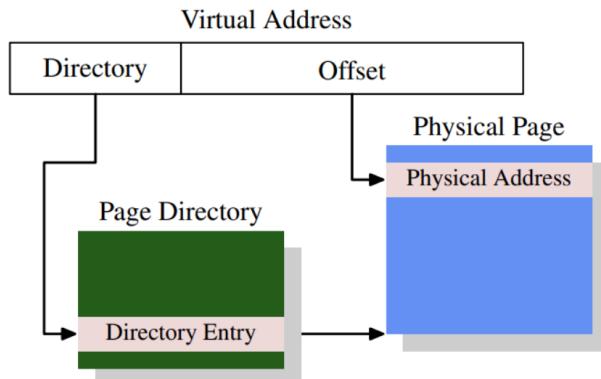
### Основной смысл использования виртуальной памяти

1. Виртуальная память разделяет адресное пространство процессов, которые работают в операционной системе
2. Помимо того, что адресные пространства процессов при работе виртуальной памяти разделены, эти же адресные пространства представляют собой один большой непрерывный блок, что в значительной степени упрощает жизнь.
3. Виртуальная память позволяет создать иллюзию того, что все процессы могут использовать больше оперативной памяти, чем ее есть на самом деле физически у текущего устройства.

### Изолирование процессов в памяти

1. У каждого процесса создается представление, что он один находится в памяти. Таким образом процессу не надо задумываться о том, что какой-либо участок памяти может быть использован кем-то другом.
2. Наблюдается хороший фактор безопасности. При наличии виртуальной памяти по умолчанию процесс не может обратиться к памяти другого процесса. Но существует возможность контролировать данное ограничение, разрешая доступ тем или иным процессам.

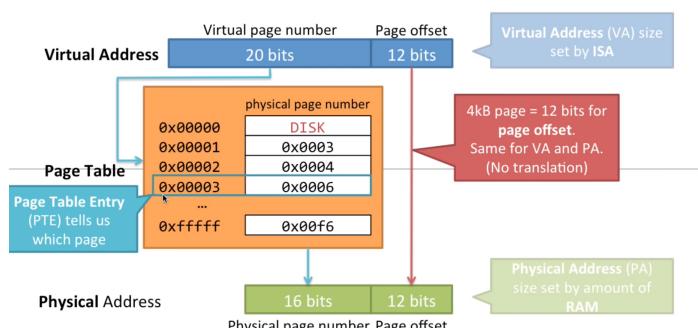
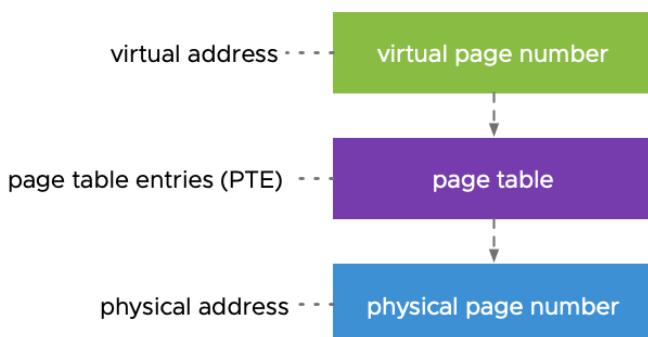
### Реализация механизма виртуальной памяти



Механизм виртуальной памяти реализуется в блоке управления памятью (MMU – memory management unit). Информация о том, как транслировать адресное пространство процессов в реальное адресное пространство физической памяти (отображение) хранится в специальных структурах данных, каждую из которых называют page table (page directory / таблица директорий).

#### Основная идея реализации:

1. Вся память разбивается на блоки фиксированного размера. Каждый такой кусочек будет иметь название *страница*. Характерный размер страниц в современных компьютерах - 4 Кб.
2. После чего создаётся специальная структура. Для каждого процесса создаётся так называемый page table.
3. Page table внутри себя хранит информацию об отображении между страницами виртуальной памяти и страницами физической памяти. У каждого процесса будет свой page table, соответственно отображение у каждого процесса будет свое.



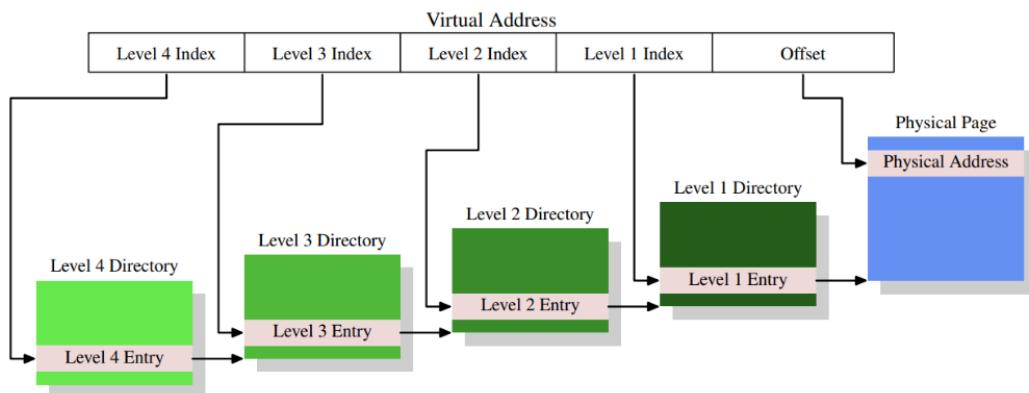
Важно понимать, что виртуальный адрес (на примере 32 бит) в первых 20 бит хранит в себе информацию о virtual page number. В остальных же 12 бит хранится смещение (offset). Таким образом, по первым 20 битам можно понять необходимый virtual address. После чего с помощью полученного виртуального адреса, используя page table, можно получить physical page number. Далее, используя offset можно дополнить полный физический адрес. Offset не меняется при трансляции виртуального адреса в физический.

#### *Основная проблема одноуровневой реализации:*

Как уже было сказано, типичный размер страницы - 4 Кб. Также offset - 12 бит и 20 бит на выбор директории. Если каждый directory entry по 4 байта, то вся таблица весит около 4 Мб. Поскольку у каждого процесса своя таблица, а процессов может быть сотни тысяч, таким образом, может получиться так, что место, которое нужно, чтобы хранить таблицы страниц для процессов, уже займёт всю оперативную память.

#### **Многоуровневые таблицы страниц**

Заметим, что процессы практически никогда не используют все свое адресное пространство. В реальности процессам нужно десятки, в худшем случае сотни мегабайт. Таким образом, в среднем процессам нужно не очень много памяти, но при этом одноуровневый подход явно пытается похоронить отображения для всего адресного пространства, не смотря на то, что большая часть этого маппинга никак не используется. Для того, чтобы хранить отображения только того, что действительно используется, используют многоуровневые таблицы страниц.



#### *Основная идея:*

Page Table сможет ссылаться не только на страницы физической памяти, но и на другие page table. Таким образом, адрес будет разбит не на две части, как было до этого, а на несколько частей (в зависимости от реализации, обычно используется разбиение 4 уровня, то есть на 5 частей). Внутри этого адреса будут храниться смещения (offset) внутри других page table.

Основной плюс такого подхода заключается в том, что в случае, если процесс использует не очень много памяти, то для такого процесса можно обойтись сравнительно малым количеством page table с второго по четвёртый уровень (в показанной выше реализации). Соответственно, при неиспользовании какого-либо участка адресного пространства, то соответствующие ему page table на некоторых уровнях можно не хранить.

1. Тривиально использование 4 уровней.
2. Адрес Level 4 Directory хранится в специальном регистре (CR3)
3. Если какой-либо directory entry пустой, то можно не хранить директории более низкого уровня.

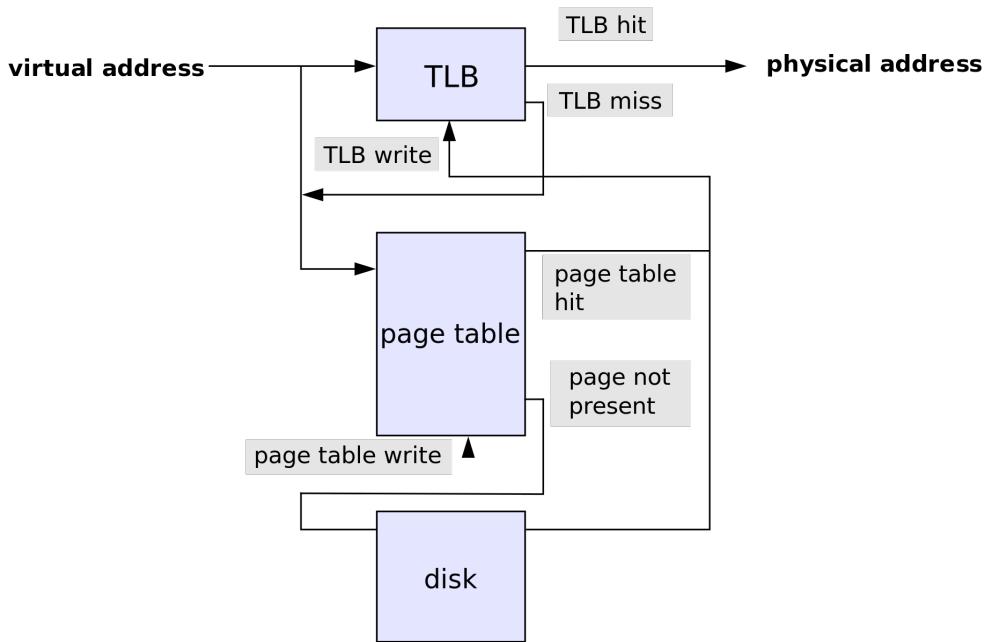
Таким образом, в общем случае многоуровневая структура представляет собой разреженное дерево, где корнем является page table 4 уровня. Для хранения такой структуры требуется значительно меньше памяти.

#### **Transaction Look-aside Buffer (TLB)**

Не трудно заметить, что в случае, если, например, в многоуровневой таблице используются 4 уровня, то на один запрос к памяти нужно сделать 5 запросов к физической памяти (получить page table 4,3,2,1 и потом по физическому адресу страницы получить адрес к которому идёт обращение).

TLB - специализированный кэш центрального процессора, используемый для ускорения трансляции адреса виртуальной памяти в адрес физической памяти.

TLB используется всеми современными процессорами с поддержкой страничной организации памяти. Каждая запись содержит соответствие адреса страницы виртуальной памяти адресу физической памяти. Если адрес отсутствует в TLB, процессор обходит таблицы страниц и сохраняет полученный адрес в TLB, что занимает в 10 – 60 раз больше времени, чем получение адреса из записи, уже закэшированной в TLB. Вероятность промаха TLB невысока и составляет в среднем от 0,01% до 1%.



В современных процессорах может быть реализовано несколько уровней TLB с разной скоростью работы и размером. Самый верхний уровень TLB будет содержать небольшое количество записей, но будет работать с очень высокой скоростью, вплоть до нескольких тактов. Последующие уровни становятся медленнее, но вместе с тем и больше.

Поскольку TLB существует в единственном экземпляре, то при переключение на другой процесс нужно очищать TLB. В зависимости от реализации существуют разные решения данной проблемы. Одно из них - хранить идентификатор данных ядра операционной системы, поскольку наиболее часто переключение происходит именно между ей и текущим процессом.

## 9. ISA.

ISA (Архитектура фон Неймана и ее альтернативы. Что описывает ISA? Assembler MIPS).

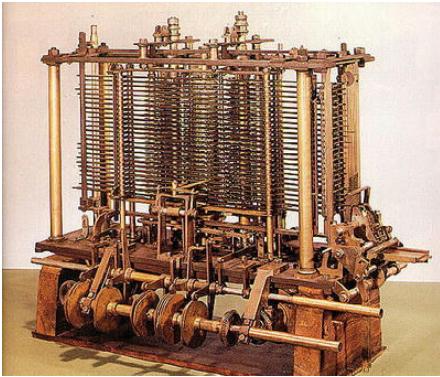
### 9.1. Архитектура фон Неймана

Архитектура фон Неймана (a.k.a. Принстонская архитектура) — широко известный принцип совместного хранения команд и данных в памяти компьютера. В общем случае, когда говорят об архитектуре фон Неймана, подразумевают принцип хранения данных и инструкций в одной памяти, но вообще она включает в себя пять базисных принципов устройства ЭВМ:

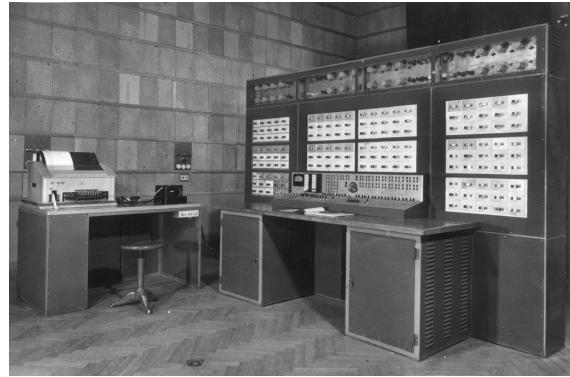
#### Использование двоичной системы счисления

ЭВМ, реализующая архитектуру фон Неймана должна использовать двоичную систему счисления.

Альтернативы:



Аналитическая машина Бэббиджа — первая в мире программируемая вычислительная машина, работает на основе десятичной системы счисления.

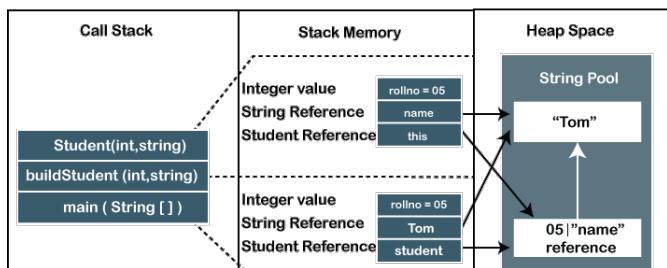


Советская «Сетунь» — единственная в мире ЭВМ на основе троичного кода.

#### Принцип адресности памяти

Структурно основная память состоит из пронумерованных ячеек, причём процессору в произвольный момент доступна любая ячейка. Двоичные коды команд и данных разделяются и хранятся в ячейках памяти, а для доступа к ним используются номера соответствующих ячеек — адреса.

Альтернативы:



Стековая архитектура  
(`Сетунь`, JVM)

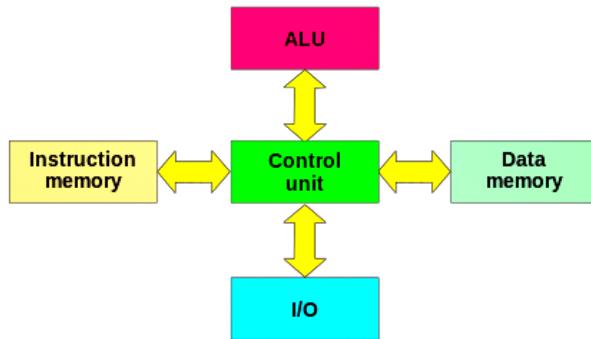


Ленточная архитектура  
(машина Алана Тьюринга)

## Однородность памяти

Код и данные хранятся в одной и той же памяти.

Альтернативы:



*Гарвардская архитектура  
(память данных и команд раздельна)*

### Архитектура фон Неймана

#### Плюсы

- Можно эффективно использовать всю имеющуюся память.
- Проще контроллер.
- Одна шина.

#### Минусы

- Так как инструкции хранятся в памяти, их можно случайно испортить во время выполнения программы.

#### Непонятно

- Однородный доступ к коду и данным.

### Гарвардская архитектура

#### Плюсы

- Можно одновременно читать и код, и данные.
- Если сделать память кода read-only, то вполне безопасно.

#### Минусы

- Усложняется контроллер памяти.
- В 2 раза больше шин.
- Память может «простаивать без дела». Например, если программа маленькая.

#### Непонятно

- Для данных и инструкций можно использовать память с разными характеристиками.

## Принцип программного управления

ЭВМ выполняет программы, записанные в памяти. Альтернативой были попытки «зашить» программу прямо в железо, но, очевидно, такой подход обречён, так как он предлагает гораздо более маленькую вариативность кода: под конкретную программу нужно собирать свою железку.

## Принцип последовательного выполнения команд

Казалось бы, понятный принцип, который соблюдается в современных компьютерах: команды выполняются последовательно, одна за другой. Но с появлением многоядерных процессоров возникло многопоточное программирование. Так что, фактически, каждый компьютер, в котором установлен многоядерный процессор, нарушает данный принцип.

## 9.2. ISA (INSTRUCTION SET ARCHITECTURE)

ISA – это абстрактная модель компьютера. Как правило, ISA определяет архитектуру памяти, разрядность адресов, режимы адресации, количество регистров, наборы команд машинного языка и их кодирование, типы данных, обработку исключений, etc.

Примеры ISA:

- x86
- ARM
- RISC-V
- MIPS

## Организация памяти и взаимодействие с памятью

### Стековая

- Операнды лежат на стеке
- Операция берёт два значения с верхушки стека и возвращает одно

### Reg-Reg

- Операнды в регистрах, результат помещается в регистр

### Reg-Mem

- Один из операндов может быть из памяти

### Аккумуляторная

- Стек размера 1
- Один операнд из аккумулятора, второй из памяти. Результат помещается в аккумулятор

### Mem-Mem

- Оба операнда могут быть из памяти

## Кодирование команд

### Команды переменной длины

- Команды имеют разную длину
- Частоиспользуемые команды могут кодироваться меньшим количеством байт, что хорошо
- Сложно декодировать

*ISA x86*: команды имеют длину 1-15 байт

### Команды постоянной длины

- Команды имеют постоянную длину
- Иногда приходится разбивать одну команду на две из-за нехватки бит для сложных команд
- Просто декодировать

В *MIPS* все команды по 4 байта

## Наборы команд

### CISC - complex instruction set computer

- Акцент за железо
- Минимизируется количество инструкций в программе
- Инструкции могут выполняться несколько тактов процессора
- Сложнее реализация железа

Архитектура x86 предоставляет программисту интерфейс CISC, но под капотом интерпритирует команды в простейшие RISC инструкции.

### RISC - reduced instruction set computer

- Акцент за программы
- Сложные команды реализуются программно через простые
- Все инструкции выполняются за один такт
- Проще реализация железа

## Режимы адресации

- Register: add r1 r2 r3, ( $r1 = r2 + r3$ )
- Immediate: add r1 r2 5, ( $r1 = r2 + 5$ )
- Register indirect: add r1 r2 (r3), ( $r1 = r2 + \text{mem}[r3]$ )
- Displacement: add r1 r2 100(r3), ( $r1 = r2 + \text{mem}[100+r3]$ )
- Absolute: add r1 r2 0xabcd, ( $r1 = r2 + \text{mem}[0xabcd]$ )

## 9.3. ASSEMBLER MIPS

### Регистры

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

### Типы команд

R-type					
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

*op* - код операции  
*rs, rt* - регистры операндов  
*rd* - регистр для результата  
*shamt* - для побитового сдвига  
*funct* - доп. код операции

I-type			
op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

*op* - код операции  
*rs* - регистр операнда  
*rt* - регистр для результата  
*imm* - 16-и битная константа

J-type	
op	addr
6 bits	26 bits

*op* - код операции  
*addr* - адрес, с которого произойдет дальнейшее выполнение инструкций

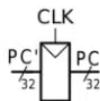
# 10. Микроархитектуры.

*Микроархитектуры (Составные части микроархитектуры. Недостатки однотактной архитектуры, ее отличия от многотактной. Конвейер MIPS: стадии, конфликты. VLIW, Superscalar).*

Микроархитектура — способ реализации ISA в конкретном процессоре. Микроархитектура постоянно изменяется для оптимизации выполнения различных процессов. Чаще всего микроархитектура процессоров внутри одной линейки почти не отличается.

## 10.1. Пример составных частей микроархитектуры

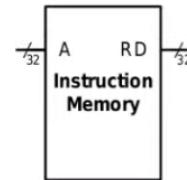
Обозначения портов на нижеприведенных схемах: **A** — адрес данных для чтения/записи, **RD** — прочитанные данные, **WD** — данные для записи, **WE** — активация записи.



Обозначение на схемах для Program Counter.

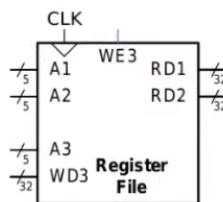
Счетчик инструкций (Register Program Counter) — 32-ух битный регистр, хранящий адрес следующей инструкции, к которому нет прямого доступа со стороны программиста.

PC' используется для записи адреса следующей исполняемой инструкции, PC излучает его на спаде сигнала.



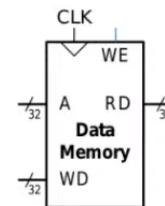
Обозначение на схемах для Instruction Memory.

Память для инструкций (Instruction memory) — память для выполняемых инструкций, к которой нет доступа со стороны программиста, данные не перезаписываются (нет интерфейса для этого). Инструкции адресуются по одной.



Обозначение на схемах для Register File.

Регистровый файл (Register File) — регистровая память с интерфейсом для чтения (сразу двух регистров) и записи данных. Предназначена для хранения операндов исполняемых команд.

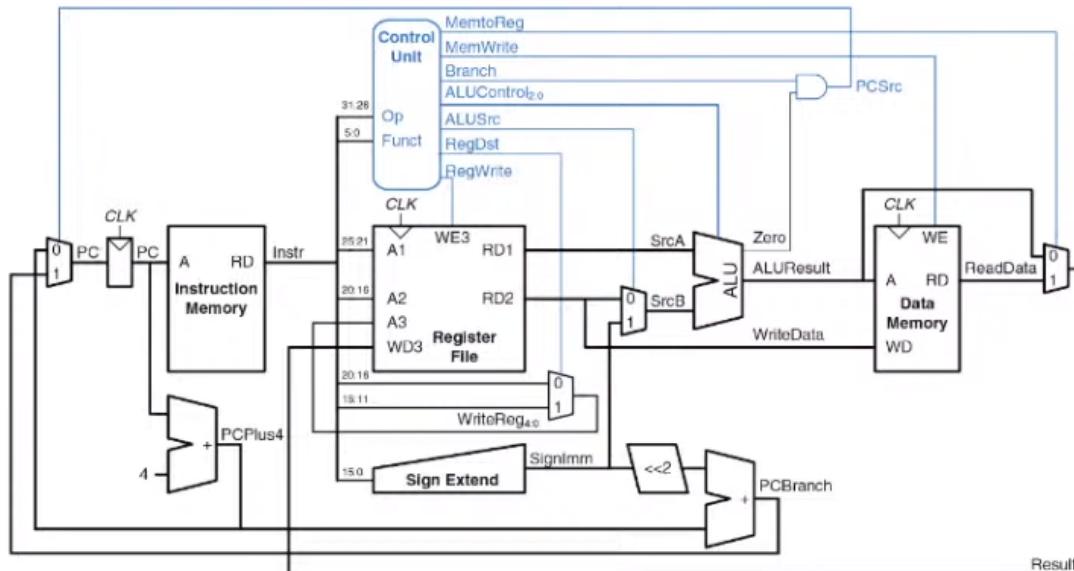


Обозначение на схемах для Data Memory.

Память Данных (Data Memory) — память устройства, использующего данный процессор, предоставляет интерфейс для чтения и записи данных. Чаще всего представляет из себя память с различными эвристиками доступа — набор из кэш и определенной памяти.

## 10.2. Однотактовый и многотактовый процессоры

### Однотактовый процессор



Легко заметить, внутреннее устройство Однотактовый процессор элементарно и крайне понятно. Если вы по каким то причинам не понимаете, как работает данная логическая схема, авторы статьи коллективно рекомендуют вам пойти и написать свой 32-х разрядный многотактовый процессор на архитектуре MIPS, используя язык verilog .

Главным минусом однотактовой реализации процессора является то, что все инструкции занимают одинаковое время, равное времени выполнения самой долгой команды (чаще всего это команды, связанные с доступом к оперативной памяти).

### Многотактовый процессор

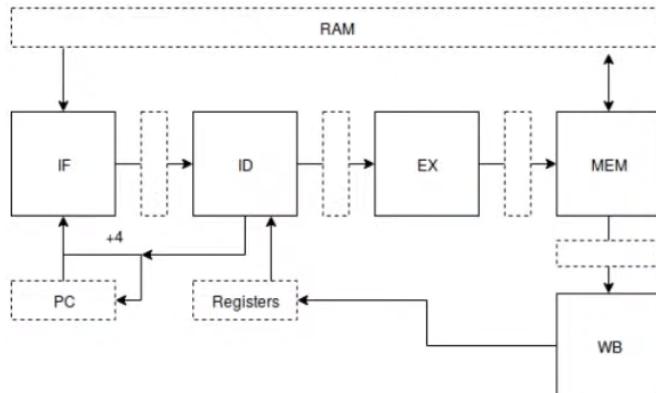
Главной задачей многотактного процессора является корреляция недостатка однотактового процессора – команды могут занимать различное количество тактов (время одного такта значительно уменьшается), в зависимости от времени их выполнения. Так же к плюсам можно отнести уменьшение количества сумматоров, по сравнению с однотактовой реализацией.

Главный минус – очень сложное устройство *Control Unit*, который отвечает за управлением выполнения команд, в зависимости от их типа.

### 10.3. Конвеер MIPS

Несложно заметить, что выполнение инструкций в процессорах (на примере однотактового) разбивается на 5 условных стадий (в реальности стадий намного больше):

- **IF (Instruction Fetch)** – вычисление адреса следующей инструкции.
- **ID (Instruction Decode)** – декодирование инструкции по ее коду, получение операндов из регистров.
- **EX (Execution)** – вычисление на АЛУ/вычисления адреса памяти для записи/вычисление условий команд ветвления.
- **MEM (Memory Access)** – обращение к памяти (для команд I-типа).
- **WB (Write-Back)** – запись результатов вычислений в регистры (для команд R/I-типа).



Схематичное представление конвеера MIPS с разделением на 5 стадий.

Разделение команд на этапы позволяет выполнять этапы разных инструкций параллельно, что значительно уменьшает *throughput* (время между выполнением двух команд).



Схема параллельного выполнения нескольких инструкций.

Для достижения параллельного выполнения инструкций необходимо значительное усложнение устройства процессора и Control Unit (хранение данных между стадиями одной инструкции, управлением состояния выполнения параллельных этапов).

### Проблемы в случае параллельного выполнения этапов

В случае параллельного выполнения команд возникает ряд проблем (конфликтов):

1. **Data Hazards (конфликт по данным).** Возникает, когда подряд идущие инструкции не независимы, то есть инструкция с номером  $n + 1$  использует результат инструкции с номером  $n$  (которая еще не завершилась полностью).

*Пример (результат выполнения первой команды произойдет на 5-ом такте – WB1, а во второй команде он нужен уже на 3-ем – ID2):*

```
add $s0, $s1, $s2
add $s3, $s0, $s4
```

*Способы решения:*

- Подождать выполнения первой инструкции, например добавив операций-заглушек (операция NOP).
- Задокументировать проблему, сказав, что это не баг, а фича.
- Forwarding – делать доступными для использования значение, полученное в результате окончания стадии EX и MEM, сразу же (дополнительно перенаправлять результат выполнения куда-то).

2. **Structural Hazards (конфликт по ресурсам).** Две параллельные стадии одновременно используют один и тот ресурс.

*Пример (на 4-ом такте первая инструкция записывает в память, четвертая инструкция читает из этой же ячейки памяти):*

```
lw $s0, $s1
add $s2, $s3, $s4
add $s5, $s6, $s7
addi $s2, 15
```

*Способы решения:*

- Подождать выполнения первой инструкции, например добавив операций-заглушек (операция NOP).
- Задокументировать проблему, сказав, что это не баг, а фича.
- Добавить ресурсов (т.е. использовать Гарвардскую архитектуру, храня инструкции, регистры и данные в разных модулях памяти).

3. **Control Hazards (конфликты управления).** Точно не известно как изменится РС (условные переходы, циклы).

*Пример (на 2-ом такте необходимо начать вторую инструкцию, но ее адрес будет получен только на 3-ем такте – EX1):*

```
bne $s1, $s2, else
add $s0, $s1, $s2
j end

else:
sub $s0, $s2, $s3
end:
```

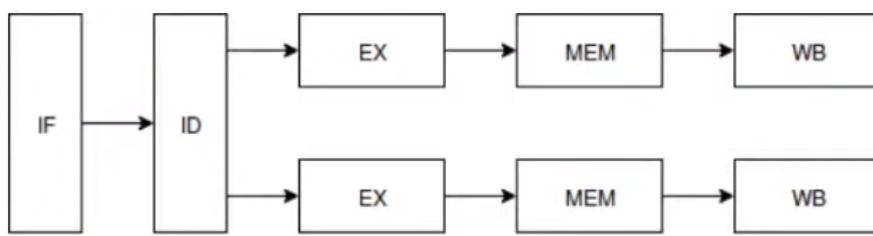
*Способы решения:*

- Подождать выполнения первой инструкции, например добавив операций-заглушек (операция NOP).
- Задокументировать проблему, сказав, что это не баг, а фича.
- Branch prediction.* Выполняем одну из веток, если мы не угадали и требуется выполнить другую, просто отменяя результат выполнения первой. Несмотря на то, что этот способ значительно усложняет устройство, именно этот способ используется, т.к. процессор постоянно должен что-то выполнять и просто ждать не выгодно.

## 10.4. Развитие конвейерной архитектуры

### VLIW (Very Long Instruction Word)

Одна команда в данном подходе – является конкатинацией нескольких обычных команд, при получении сконкатенированной команды она разбивается на обычные, которые выполняются одновременно. Таким образом производительность полностью зависит от компилятора (того, насколько хорошо он склеит команды).



Пример организации 2-pipeline VLIW (2 инструкции параллельно) конвеера.

**Плюсы**

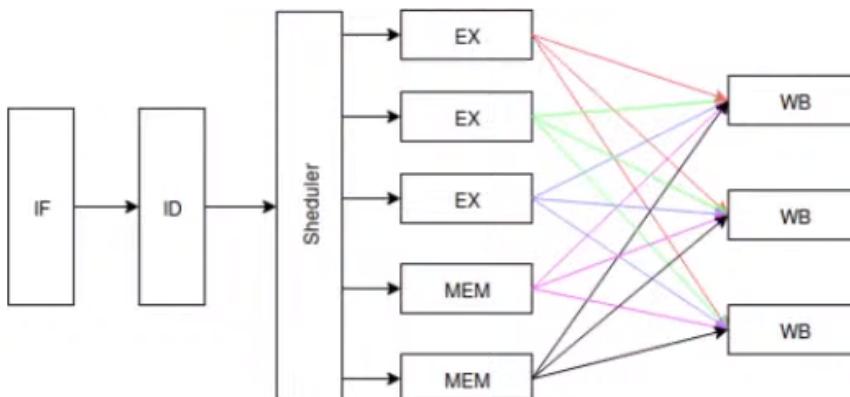
- Просто реализовать в железе.
- Производительность зависит от компилятора (можно использовать различные эвристики для оптимизации).

**Минусы**

- Производительность зависит от компилятора (их тяжело реализовать).
- ISA зависит от микроархитектуры (чего быть не должно, из-за этого, например, придется перекомпилировать код):
  1. Различный размер сконкатенированных команд в зависимости от кол-ва конкатенируемых команд.
  2. Количество конкатенируемых команд может быть различно, так, например при выполнении команд скомпилированных для 2-pipeline VLIW на 4-pipeline VLIW могут возникнуть ошибки (рассчитано, что будут выполняться только 2 команды параллельно).
- Значительное увеличение времени выполнения стадии ID.

**Superscalar**

Стадии IF, ID выполняются последовательно, за ними идет «Планировщик», который отвечает за распаралеливание этапов EX, MEM, WB.

**Плюсы**

- Упрощение интерфейса для программиста.
- Компактный код (не требуется не каких оптимизация в расположении инструкций).
- ISA не зависит от реализации микроархитектуры.

**Минусы**

- Планировщик реализуется в железе и должен учитывать огромное количество различных ситуаций.
- Большая вероятность создания конфликтов.
- Необходима синхронизация после выполнения какого-то количества паралельных инструкций.

## 11. Внешние запоминающие устройства.

*Внешние запоминающие устройства (Устройство магнитных и оптических накопителей. Что такое дорожка, сектор? Что хранится в секторе? RAID. Флеш накопители и SSD и как они работают, в чем отличия от HDD?).*

### 11.1. Внешние запоминающие устройства

Внешнее запоминающее устройство – устройство, предназначенное для записи и хранения данных. В основе работы запоминающего устройства может лежать любой физический эффект, обеспечивающий приведение системы к двум или более устойчивым состояниям.

### 11.2. Магнитные накопители

Магнитный накопитель – это внешнее запоминающее устройство, использующее какую-то поверхность, которая может быть намагниченна, и какое-то устройство, которое может по заданому участку поверхности считать его намагниченность и изменить её. В качестве накопителя могут выступать: лента, алюминиевая поверхность, стеклянная поверхность и так далее.



Магнитный жёсткий диск



Ленточный накопитель



Дискета

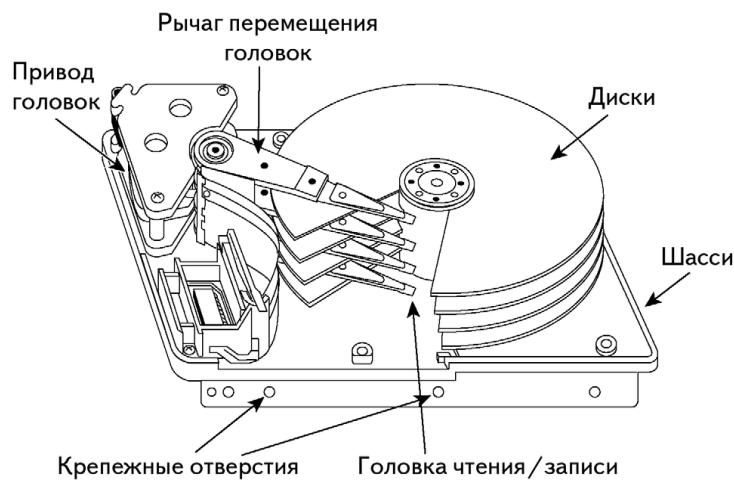
#### Чтение и запись данных в HDD

В случае с дисками (которые врачаются порядка 7200/3600RPM), круговые участки намагниченности представляют собой последовательности битов, называемые дорожками, 1 бит информации хранится на участке шириной порядка  $0.1\mu m$ . Дорожка делится на сектора фиксированного размера (512 байт, сейчас 4 Кб). Состояние каждого участка считывается и записывается специальной головкой, в зависимости от намагниченности. Такой сектор является минимально адресуемой единицей памяти на диске.



Важно понимать, что каждый сектор жесткого диска разбит на 3 части:

- Преамбула, хранящая данные о новом секторе.
- Данные сектора.
- Исправляющие коды, которые способны улавливать и устранять ошибки (например, коды Рида-Соломона, которые). Коды Рида-Соломона, наподобие кодов Хемминга, но оперируют блоками данных, исправляя в них ошибки.



В реальности, жёсткий диск состоит даже не из одной пластины, а из нескольких. В наше время на поверхности одного диска может храниться до 1Тб данных.

### Проблемы HDD

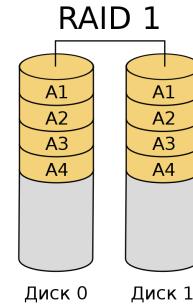
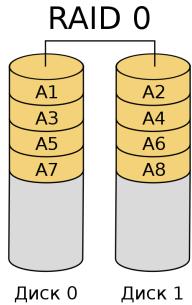
- Перестановка считывающей головки на произвольный сектор занимает в худшем случае 5-10 милисекунд. (долго).
- Из-за того, что угловая скорость вращения постоянна, плотность записи информации на диске неравномерна: чем дальше от центра, тем менее плотная запись.

К счастью, второе можно исправить, сделав так, что количество секторов на разных дорожках будет увеличиваться от центра к краю.

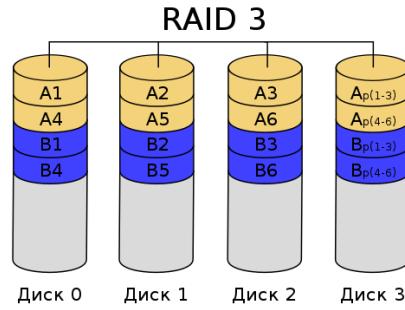
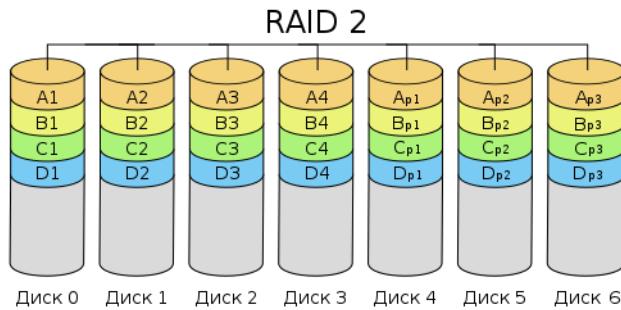
Взаимодействие в жесткими дисками происходит через контроллер, — микросхему, которая получает команды на чтение/запись/форматирование, управляет перемещением головок, обнаруживает и исправляет ошибки, передает данные наружу и даже следит за здоровьем жесткого диска. Угловая скорость вращения диска **не меняется**.

Для того чтобы увеличить объем хранимых данных и обеспечить их сохранность используются 2 подхода: RAID (*Redundant Array of Independent Disks*) и SLED (*Single Large Expensive Disk*).

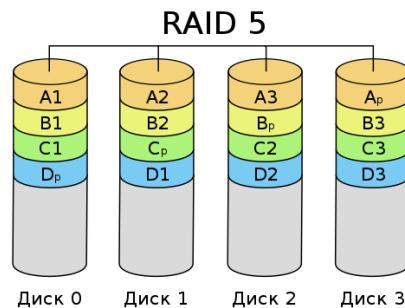
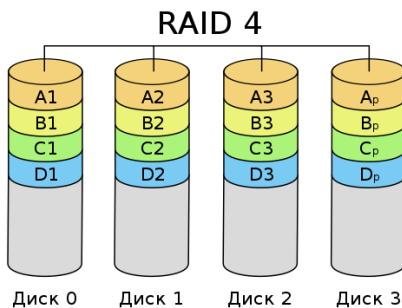
## RAID (Redundant Array of Independent Disks)



- Разбиваем объем  $n$  жестких дисков на  $m$  равных блоков, связанных между собой
- Можно читать  $n$  блоков параллельно с  $n$  дисков
- Скорость увеличивается существенно, если данные разбиты равномерно между дисками
- Шанс потерять данные в  $n$  раз выше

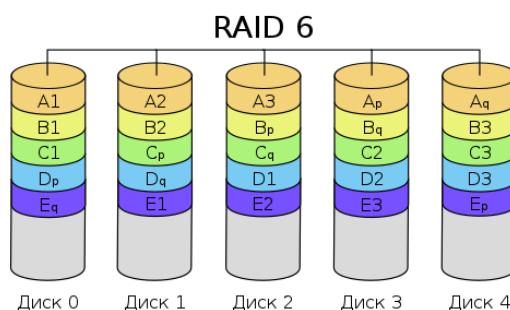


- Одна часть жестких дисков для хранения информации, другая для битов исправления хранимых данных
- Оперируем блоками или байтами
- Используем код Хэмминга, можем исправлять ошибки на лету
- Нужно минимум 7 дисков, быстрый износ

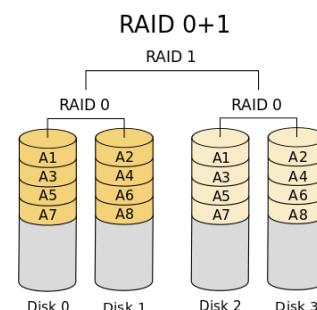


- То же самое, что и RAID 3, но вместо байт - блоки

- Храним байт чётности
- Хорошо читать
- Плохо писать (перезапись 3 диска)
- Исправление ошибок дольше



- То же самое, что и RAID 5, но теперь все диски изнашививаются равномерно



- То же самое, что и RAID 5, только мы храним 2 блока чётности

- Просто 2 RAID 0, соединённые в RAID 1

## Оптические накопители

*Оптические накопители* похожи на магнитные. Они имеют одну длинную закручивающуюся спираль на весь диск. Информация хранится с помощью дырок и площадок (переход дырка-площадка или площадка-дырка это единичка, плоская поверхность - нолик). В них всё так же есть преамбула и коды коррекции.



Выгодно использовать тогда, когда информация читается непрерывно, полностью (музыка, фильмы).

Для того, чтобы перезаписать информацию приходится срезать слой и нанести новые углубления на диске.

Угловая скорость вращения диска **меняется** от центра к краю, чтобы линейная скорость при считывании оставалась постоянной.

## Флеш-накопители (SSD и флешки)



SSD-диск на 480 Гб



флешка

*Флеш-накопители* для хранения информации используют транзисторы с плавающим затвором. В обычном транзисторе данные теряются при отключении питания, а в транзисторе с плавающим затвором заряд на затворе сохраняется. Благодаря этому, на них можно хранить и перезаписывать информацию.

Транзисторы с плавающим затвором могут хранить в себе 2 (SLC, Single Level Cell), 4 (MLC, Multi Level Cell), 8 (TLC, Tripple Level Cell), 16 (Quadripple Level Cell) состояний. В жизни вы чаще всего встречаетесь 4 или 8 состояний на транзистор.

С точки зрения внутренней реализации SSD и флешки очень сильно похожи на оперативную память. SSD на самом деле представляет из себя флешку с умным контроллером, который равномерно распределяет по нему нагрузку.

### Плюсы

- Нулевое время поиска данных по сравнению с HDD.
- Не страдают от резких перемещений в пространстве.

### Минусы

- Заряд на транзисторе может пропадать со временем.
- Для перезаписи данных сначала необходимо записать 0, а затем данные.

## 12. Параллелизм.

*Параллелизм (MIMD, что такое SMT, SIMD, почему GPU это SIMD).*

### 12.1. Вводная терминология

Процесс — часть программы, запущенная на выполнение. Процессы в общем случае не имеют общего кода и общей памяти. Они достаточно независимы друг от друга.

Поток — наименьшая последовательность инструкций внутри процесса, которой может независимо управлять планировщик.

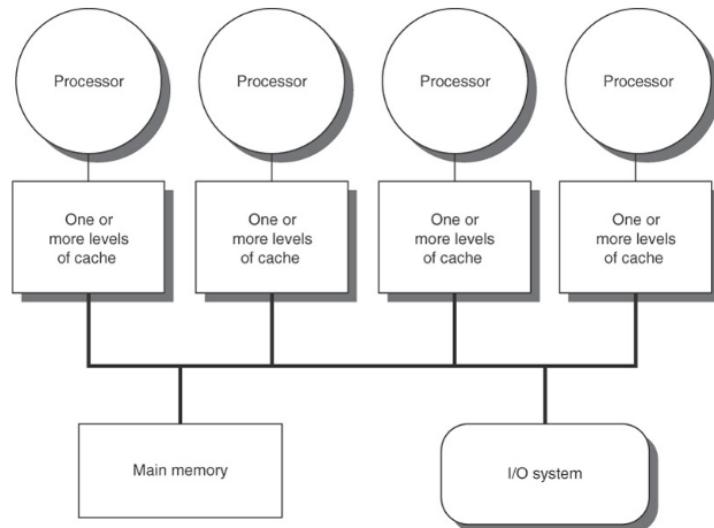
Многоядерный вычислитель — несколько вычислительных ядер на одном кристалле процессора. При этом L1 кэш уникальный для каждого ядра, L2 уникальный или общий, а L3 - общий для всех.

Многопроцессорный вычислитель — физически несколько процессоров (многоядерных или одноядерных).

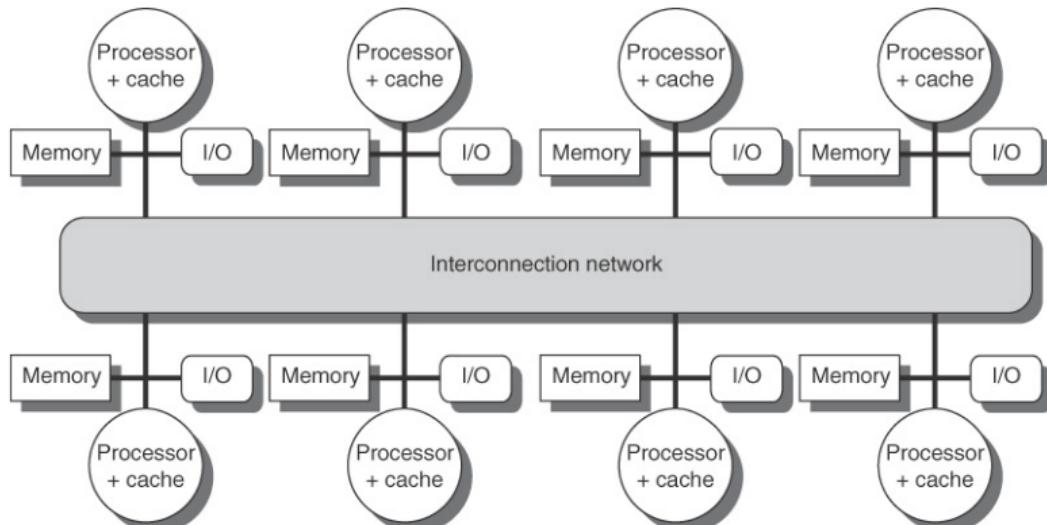
### 12.2. Виды параллельных архитектур (параллелизм по инструкциям и по данным)

1. SISD (Single Instruction stream Single Data stream) - простой одноядерный процессор.
2. SIMD (Single Instruction Multiple Data) - архитектура, характерная для видеокарт и векторных процессоров.
3. MISD (Multiple Instructions Single Data) - не имеет практического применения.
4. MIMD (Multiple Instructions Multiple-Data) - многоядерный процессор.

#### MIMD



Архитектура MIMD используется для многопроцессорных архитектур общего назначения. При небольшом количестве ядер можно поддерживать работу с памятью в форме модели UMA.



При большом количестве вычислителей общую память поддерживать сложно, поэтому используется модель разделения памяти NUMA.

## SMT и HyperThreading

*SMT (Simultaneous Multithreading / Одновременная многопоточность)* - подход при котором один процессор выполняет несколько потоков операций.

Hyperthreading - конкретная реализация SMT от Intel.

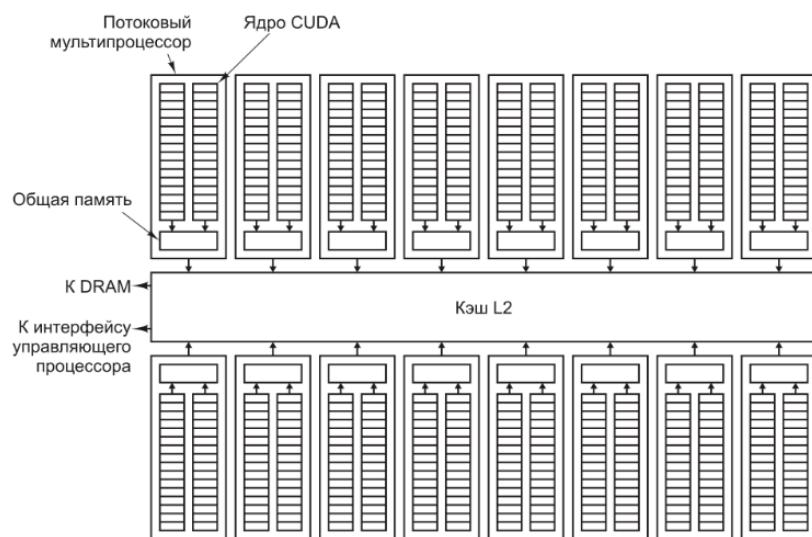
Алгоритм работы:

1. Каждое ядро может хранить состояние двух потоков, используя два набора регистров и два контроллера прерываний.
2. Количество реальных вычислителей не меняется (один), однако вычислитель постоянно переключается между двумя потоками, которые обрабатывает текущее ядро
3. Данная модель возможна и эффективна при кэш промахах одного из процессов. Поскольку в данном случае оптимизируется время, потраченное на поиск данных в оперативной памяти (пока один процесс делает запрос, над данными второго начинается работа)

## SIMD

Векторные процессоры - первые, кто реализовывал архитектуру SIMD. Основное отличие в том, что операндами могут выступать целые массивы данных. Однако векторные процессоры не увенчались успехом, более того, почти все современные микропроцессоры могут производить векторные вычисления (семейство расширений SSE).

## Почему GPU это SIMD?



ГПГПУ — техника использования графического процессора видеокарты, предназначенного для компьютерной графики, в целях производства математических вычислений, которые обычно проводит центральный процессор. Например, достаточно удобно рассчитывать задачи связанные с машинным обучением, к примеру, перемножение матриц, поскольку в данном случае можно хорошо использовать параллельные вычисления.

На примере архитектуры, представленной выше, у нас есть множество мультипроцессоров, у них у всех есть кэш второго уровня. Видеокарта может одновременно использовать все имеющиеся мультипроцессоры, выполняя на каждом разные инструкции. Непосредственно подход SIMD реализуется внутри данных потоковых мультипроцессоров. Смотря на картинку выше, нетрудно описать реализацию, которая состоит в том, что на одном столбце можно выполнить какую-то одну операцию над различными данными.

