**Кэш** - небольшая быстрая память, расположенная близко к процессору (обычно на одном с ним кристалле). Обычно SRAM.

# 1 Зачем?

Потому что доступ к памяти долгий, а часто имеющиеся данные хотелось бы получать за быстро.

# 2 Политика чтения

## 2.1 Look-aside

CPU отправляет запрос на чтение в память. Кэш подсматривает. Если не может ответить на запрос - не отвечает, если может - отвечает, а в память посылает отмену.
Данные, которые еще не закэшированны читаются быстрее.

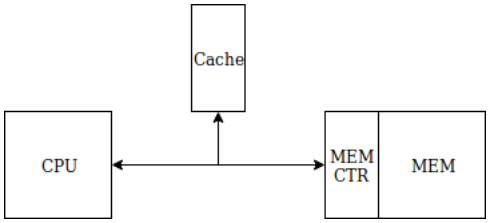


Рис. 1: Look-aside

## 2.2 Look-through

CPU отправляет запрос на чтение в кэш. Если у кэша есть нужная линия, он отвечает. Если нет, то запрашивает линию у основной памяти, а её ответ сохраняет себе. Если

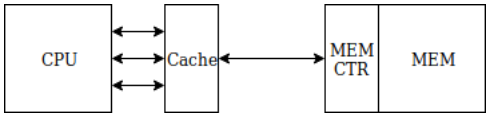


Рис. 2: Look-through

у нас кэш умный, у нас cache-hit в 90% запросов. А значит скорость обращения к памяти нам не очень важна, а вот скорость обращения к кэшу - очень. При look-through можем сделать **жирную** шину до процессора (т.к. они с кэшем на одном кристалле) и выиграть в скорости. А при look-aside - нет.

# 3 Политика записи

## 3.1 При cache-hit

### 3.1.1 Write-through

Пишем и в кэш, и в память. Проще реализовать, но очередь к памяти забивается не очень полезными запросами (на запись в одну и ту же ячейку, например).

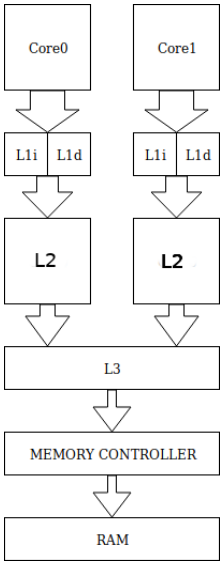


Рис. 3: Примерная схема. Чем ниже, тем тоньше шина

#### 3.1.2 Write-back

Пишем только в кэш, когда линия вытесняется - записываем в память. Реализовать сложнее, т.к. нужно как-то запоминать, являются ли данные копией памяти (clean) или изменены (dirty). Эта штука быстрее, потому что в очереди к памяти не стоят бесполезные последовательные запросы на запись вида: записать в ячейку X значение 1, записать в ячейку X значение 2 и т.д.

### 3.2 При cache-miss

#### 3.2.1 Write allocate или fetch on write

При кэш-промахе нужная кэш-линия сначала загружается в кэш, а потом в неё происходит запись. В таком случае промахи при записи аналогичны промахам при чтении.

#### 3.2.2 No-write allocate или write around

При кэш-промахе пишем сразу в память, не подгружая ничего в кэш.

### 3.3 Уровни кэша

В современных архитектурах кэш делится на уровни, обычно на 3.

- **L1** - кэш первого уровня. Содержит то, что активно используется, но не уместилось в регистры. Обычно делится на кэш для данных **L1d** и кэш для инструкций **L1i**. Обычный объём 32КБ. Время доступа 4 такта.
- **L2** - кэш второго уровня. Обычно общий для данных и для инструкций. Содержит часто используемые данные. Обычный объём 256КБ. Время доступа 12 тактов.
- **L3** - кэш третьего уровня. Содержит то, что еще успели закэшировать. Обычный объём 8МБ. Время доступа 50 тактов. Обычно общий для всех ядер.

### 3.4 Ещё про кэш

Кэш - это быстрая SRAM память, а она занимает много места.
Кроме того, ближе к реальности схема, когда у нас L3-кэш - это несколько маленьких

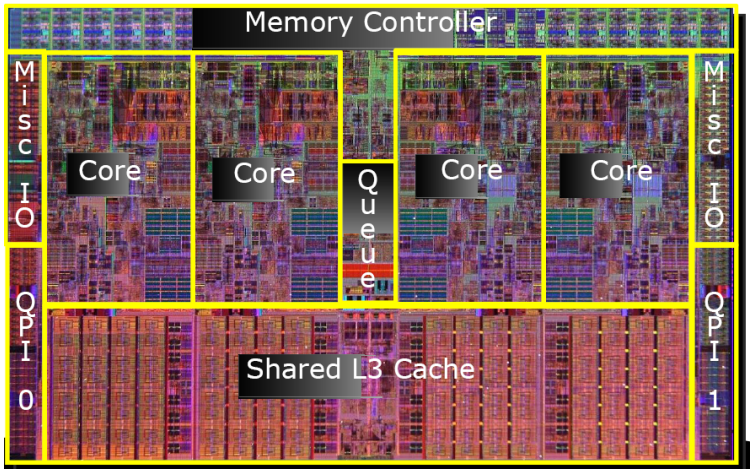


Рис. 4: Core i7

кэшей, сидящих на одной шине, а не один большой. Таким образом повышается маштабируемость: у нас есть блок из CPU-L1-L2-L3, который можно размножать копипастой.

## 4 Кэширование данных

Если бы кэш кэшировал побайтово, то вместе с каждыми 8 битами данных мы хранили бы еще 32 бита адреса. Не очень выгодно, особенно учитывая что кэш - дорогое удовольствие (занимает до четверти кристалла, ы).
Данные кэшируются кэш-линиями. Обычно одна кэш-линия - 64 байта. Соотвественно адрес можно хранить один на кэш-линию. Кэш-линии не пересекаются! Т.е адрес кэш-линии должен быть кратен 64. Следовательно можем не хранить для линии последние 6 бит адреса - это всё равно нули.

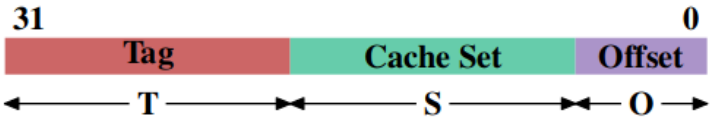


Рис. 5: Как-то так разделяется адрес. Offset - адрес данных внутри кэш-линии

## 5 Политика кэширования

Поиск кэш-линии ведется по тэгу (блок T в адресе). Существует несколько способов организовать адрес.

### 5.1 Полная ассоциативность

Можем любой блок данных положить в любую строку кэша. Т.е. размер блока S в адресе равен 0.

Это значит, что нам нужно провести $2^T$ сравнений, где $T$ - размер тэга. Это очень много. Мы не можем делать сравнения последовательно (это долго, следовательно теряется весь смысл кэша), значит нам нужно $2^T$ аппаратных компараторов. Что-то многовато.

### 5.2 Direct-mapping

Другая крайность: пусть теперь размер тэга равен нулю.
Встречаем другую проблему: если мы обращаемся к блокам памяти, которые попадают в одну кэш-линию, то работа с ними получается даже немного медленнее, чем без кэша: мы постоянно вытесняем одни данные другими.

### 5.3 Групповая ассоциативность

Разделим кэш на $N$ групп по $M$ кэш-линий в каждой.
Пусть поступил запрос с адресом ячейки X. Последние $O$ ячеек - адрес внутри кэш-линии. Следующие с конца $\log N$ бит - адрес группы. Внутри группы ищем кэш-линию по тэгу.
Сейчас в основном распространена 8-ассоциативность.

## 6 Когда кэш полезен?

Идея кэша опирается на идею **пространственно-временной локальности** (в близкие промежутки времени обращаемся к близко лежащим данным). Соответственно если алгоритм эту идею не использует (обращается к памяти более-менее случайно), то кэш не особенно поможет.

## 7 Кэш с точки зрения программиста

Фактически программно кэш не виден. Но есть некоторое количество команд, позволяющих к кэшу обратиться:

- flush кэш-линии в память
- команды-рекомендации вида: "я тут собираюсь использовать вот те данные, если можешь, закэшируй". Эффективно использовать за 200 тактов до начала взаимодействия с искомыми данными.