Лекция 3. Архитектурные особенности современных компьютеров. Исполнение инструкций.

Содержание

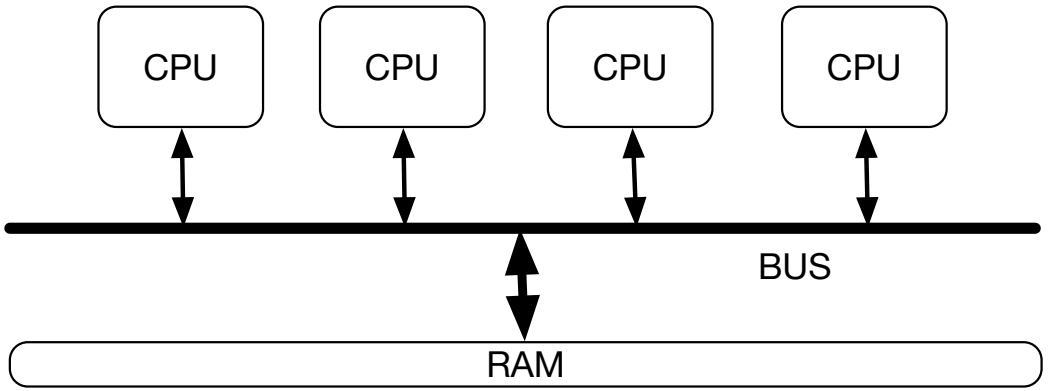
- Протокол работы кэша
- Особенности программирования, учитывающие влияние кеша.
- Особенности системы команд x86/x64.
- Видимость результатов работы команды.

Кэш: Использование нескольких вычислительных ядер

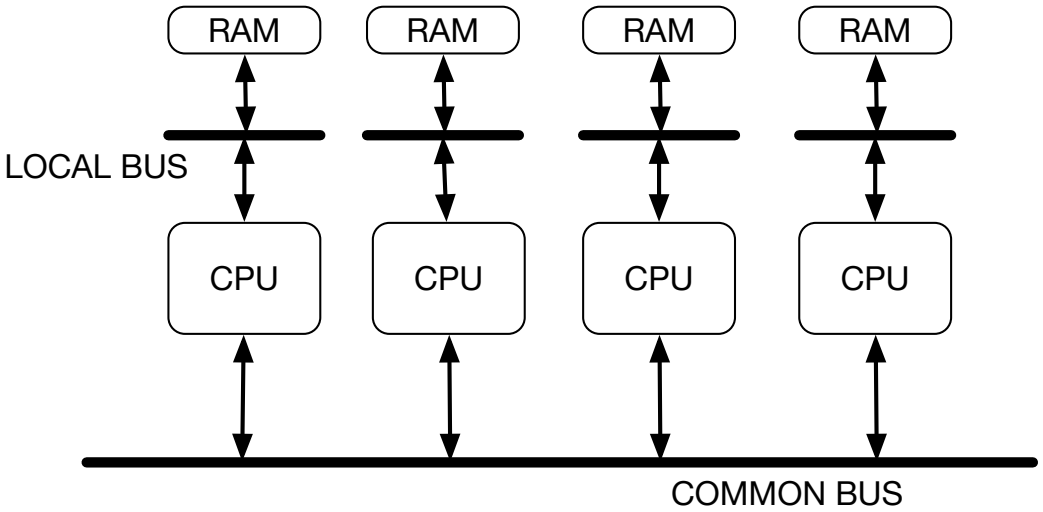
- Если каждому процессору выделить собственную память, то каким образом они будут взаимодействовать?
- Если множество процессоров будет работать с общей памятью, то каким образом избежать конфликтов?

Кэш: использование нескольких вычислительных ядер

Взаимодействие нескольких процессоров.



Симметричная архитектура (SMP)



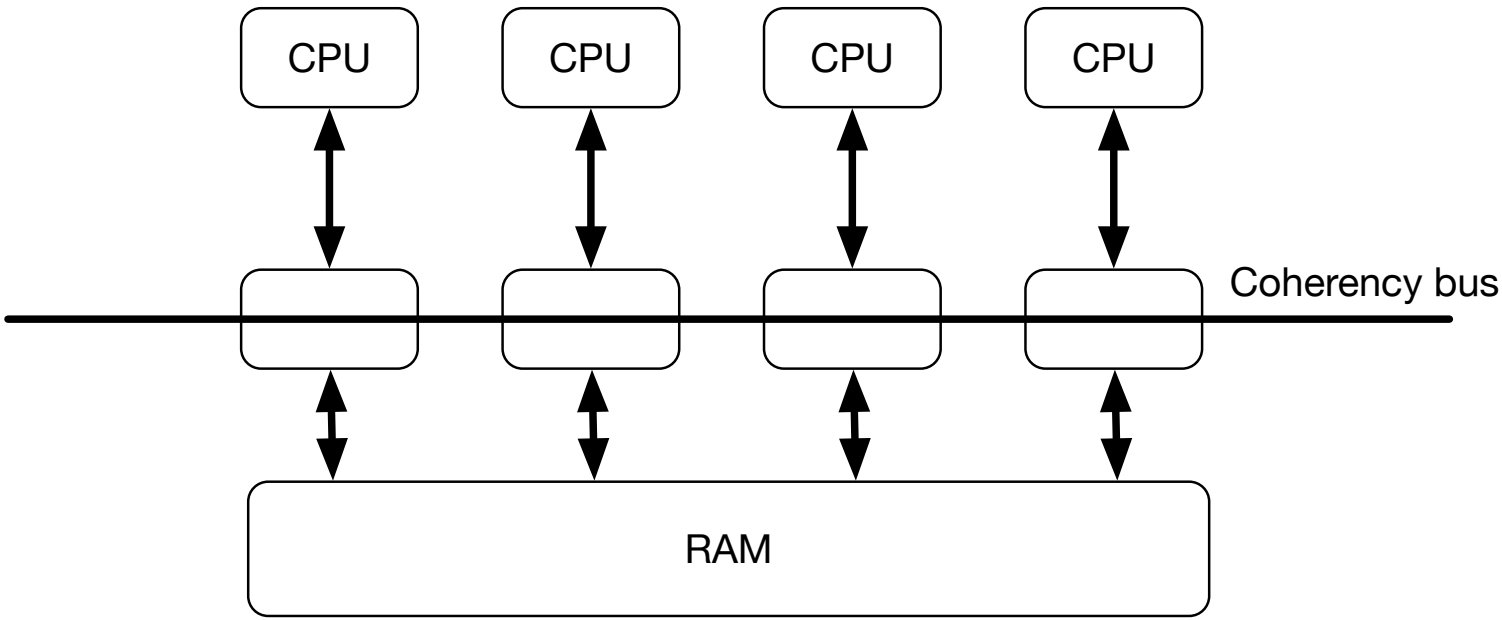
Несимметричная архитектура (NUMA)

Кэш: разделение между потоками

- Пусть два ядра работают с одной переменной.
- Разные ядра обращаются к одним участкам памяти.
- Как несколько потоков могут иметь одновременный доступ к одним и тем же линейкам кэша?
- Если два ядра работают с одной и той же переменной, то эта переменная должна принадлежать какой-то из линеек.
- Чтении переменной $\rightarrow$ каждый кэш содержит копию линейки.
- Первое ядро хочет записать переменную.
- Две линейки станут различными.
- Второе ядро может использовать свою копию.
- А если и оно захочет изменить линейку?
- Требуется *согласование*, когерентность кэша.

Кэш: когерентность

- Современные процессоры используют абстракцию *шины когерентности*, *coherency bus*.
- Когерентность кэша важна для поддержания *консистентности* памяти.
- Термин: *фрейм* — фрагмент RAM, соответствующий линейке.



Кэш: протоколы синхронизации: MOESI

- Операция поддержания когерентности — *синхронизация*.
- Имеется 5 состояний линейки кэша:

M `Modified`. Линейка не совпадает с фреймом. Кто-то обратился к памяти, линейка загрузилась в кэш и её модифицировали.

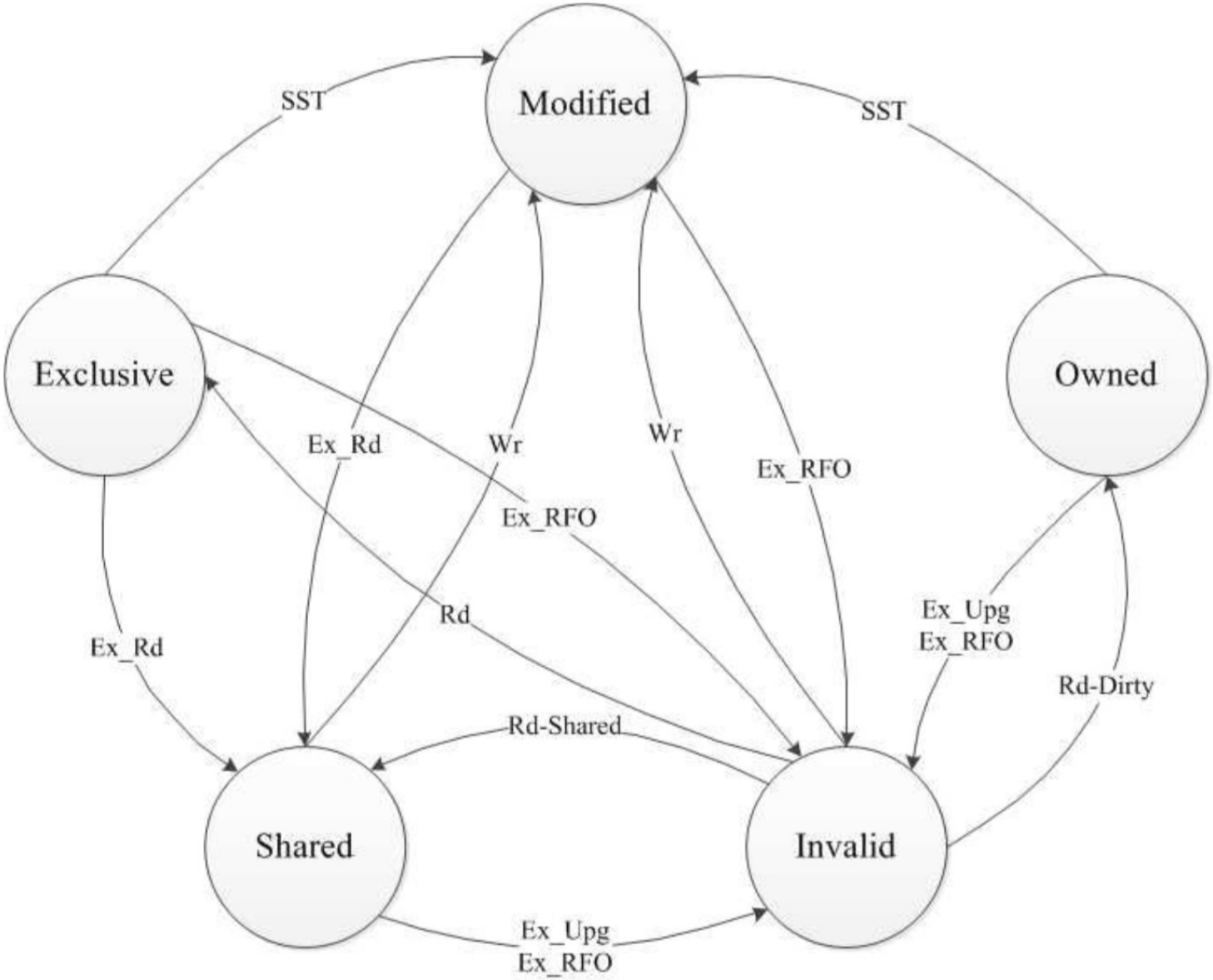
O `Owned`. В линейке актуальные данные. Другие тоже могут иметь копии фрейма, они должны быть `Shared` и могут отличаться от того, что в `Owned`. Владелец записал линейку в память, состояние $\leftarrow$ `Shared`. Если остальные состояния $\leftarrow$ `Invalid`, то $\rightarrow$ `Modified`.

E `Exclusive`. Линейка содержит наиболее свежие данные, совпадающие с фреймом. Никто не может получить копию фрейма.

S `Shared`. Фрейм используется совместно на чтение. Линейки совпадают с фреймом. Никто не может модифицировать. Хочешь записать — переведи в `Exclusive`, остальные в `Invalid`.

I `Invalid`. Линейка не соответствует фрейму. Корректная информация либо в фрейме, либо в другой линейке.

Диаграмма переходов протокола MOESI



Кэш: разделение

- L1 не разделяется между ядрами и между HT ядрами. Применяется модифицированная Гарвардская архитектура, кэш инструкций (только на чтение) и кэш данных (на чтение/запись).
- L2 разделяется попарно ядрами.
- L3 разделяется всеми ядрами.

Кэш: общий протокол работы

Резюмируем общий протокол работы с кэшем:

1. Если запрошенная информация в кэше отсутствует (*cache miss*), то исполняются следующие операции:
   - обращение к кэшу;
   - обращение к оперативной памяти;
   - освобождение места в кэше, поиск жертвы;
   - загрузка из оперативной памяти в кэш;
   - передача из кэша в процессор.
2. Если запрошенная информация в кэше имеется (*cache hit*), то:
   - обращение к кэшу;
   - передача из кэша в процессор.

Особенности программирования, учитывающего влияние кэша.

Кэш: гранулярность

- Размер кэш-линеек превосходит размер привычных для всех объектов в языках программирования.
- Для повышения производительности нужно учитывать гранулярность кэша.
- Локальность: одна из продуктивных стратегий. Полагается, что если мы используем адрес p скоро будем использовать еще раз или его, или соседние адреса.
- p берем из кэша. Если соседний адрес в том же фрейме, то информация берется не из памяти, а из линейки.
- Как оценить коэффициент попадания в кэш? Даже небольшой процент непопаданий увеличивает эффективное время доступа.
- Больше локальности $\rightarrow$ больше эффективность $\rightarrow$ меньше эффективное время доступа.

Кэш: гранулярность

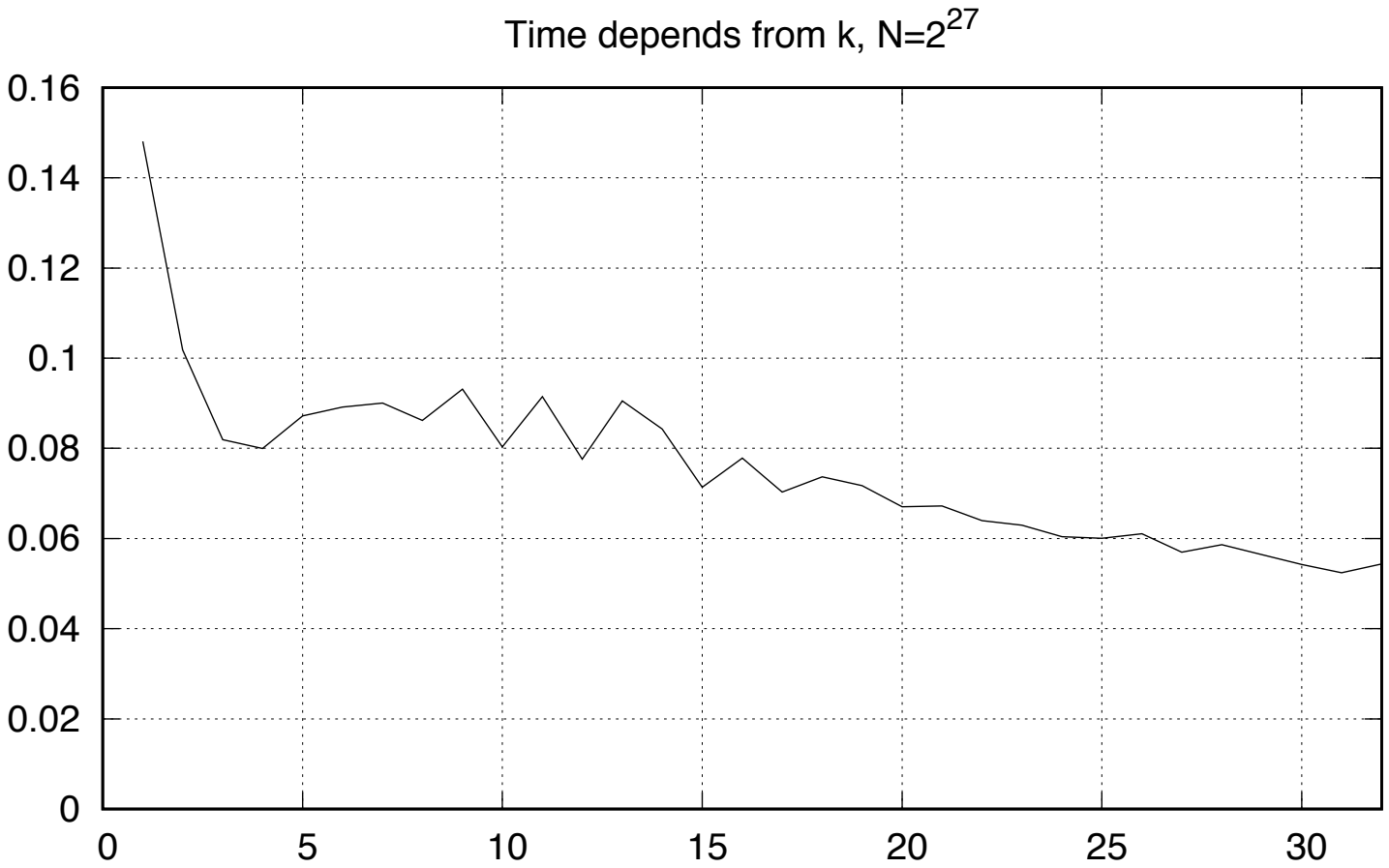
Гранулярность — также источник проблем. При считывании одного слова читается вся линейка.

```
//Example 1.
double *arr;
double sum = 0;
const int k = 1;
for (int i = 0; i < N; i += k) {
   sum += arr[i];
}
```

```
//Example 2.
double *arr;
double sum = 0;
const int k = 4;
for (int i = 0; i < N; i += k) {
   sum += arr[i];
}
```

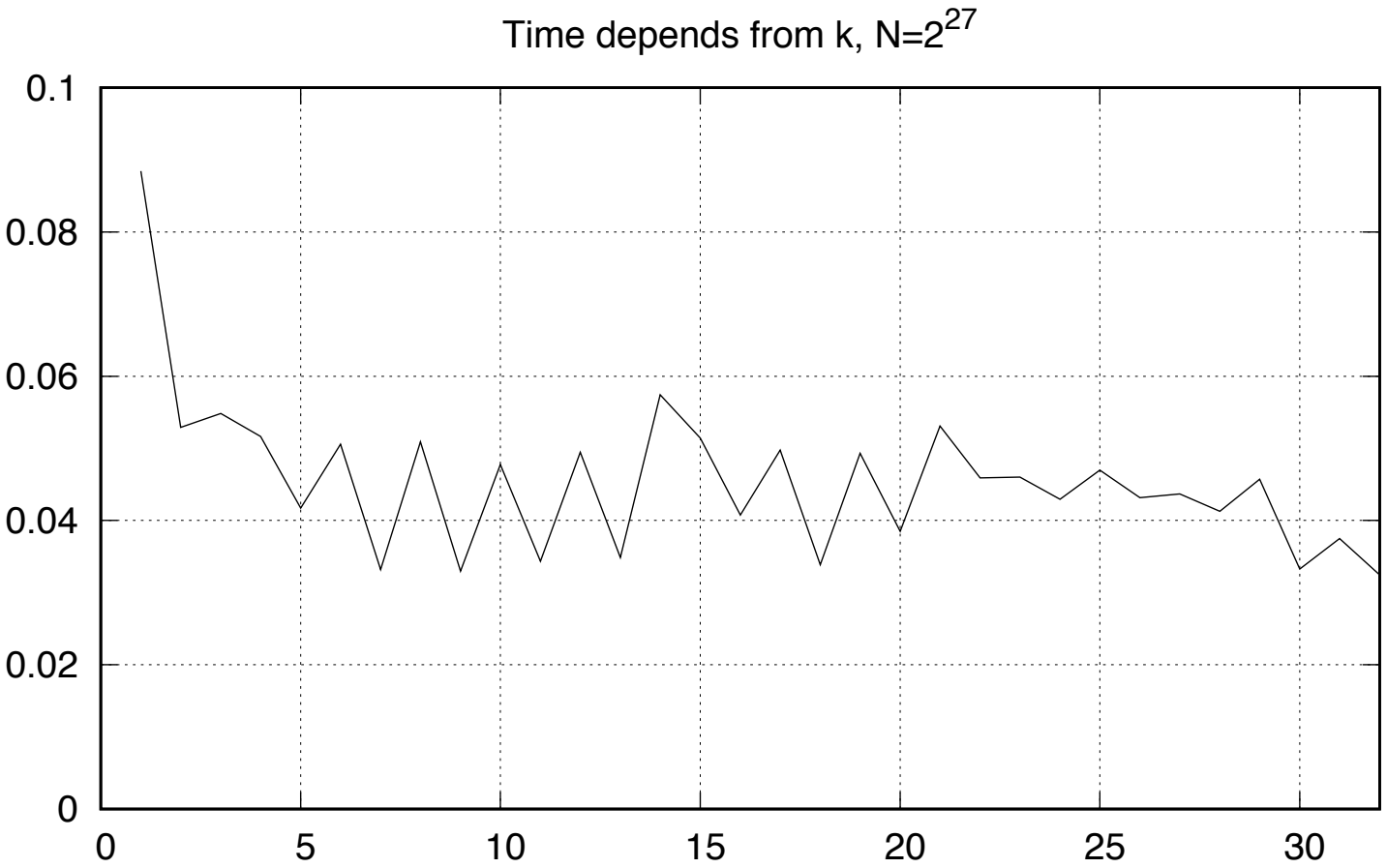
Чем больше k, тем формально меньше обращений к оперативной памяти. Однако...

Кэш: гранулярность



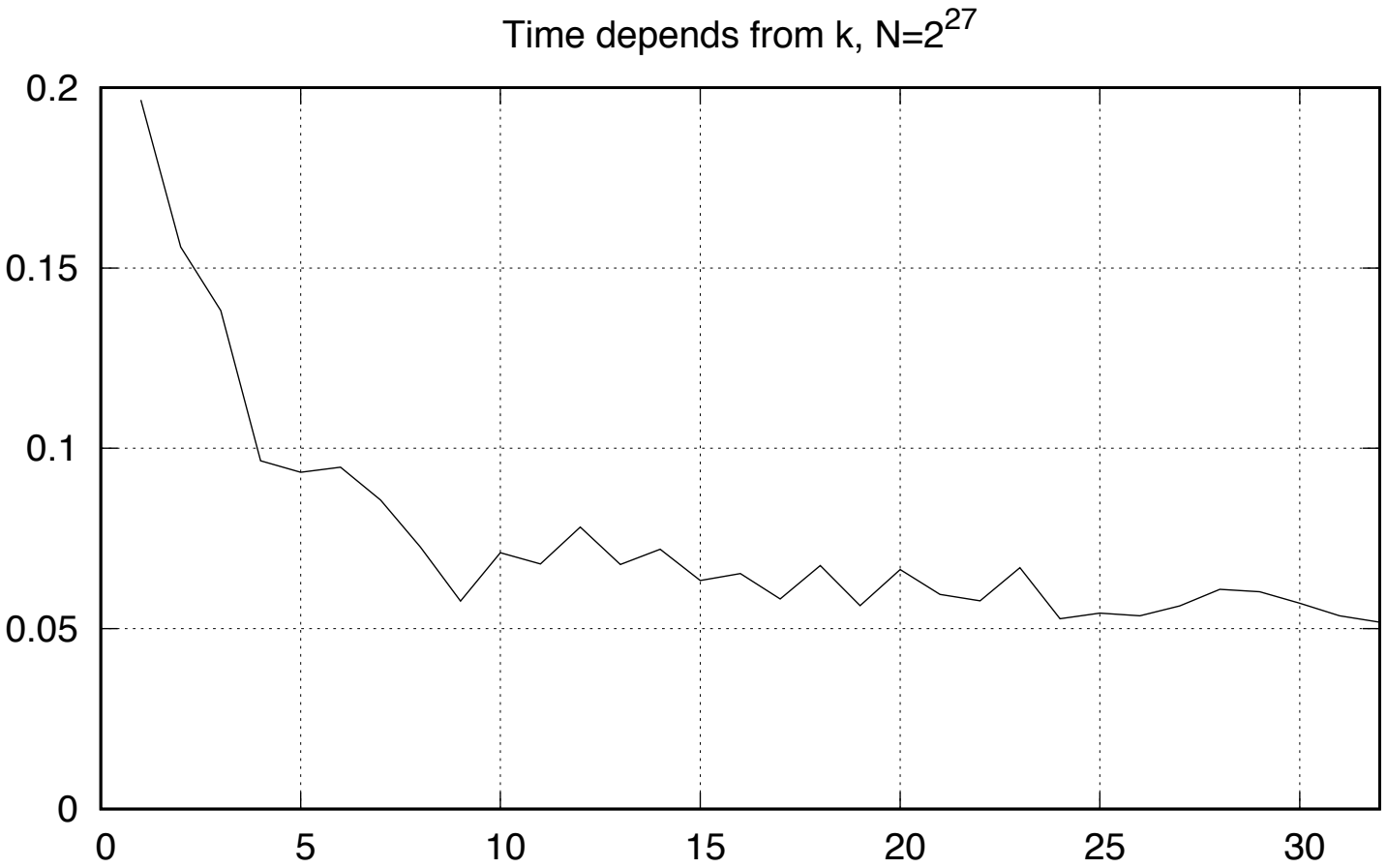
Результаты прогона на Intel i7 с кэшем L2 в 256 килобайт на ядро и общим кэшем L3 в 6 мегабайт, размер double массива 1 гигабайт.

Кэш: гранулярность



Результаты прогона на Intel i7 с кэшем L2 в 256 килобайт на ядро и общим кэшем L3 в 6 мегабайт, размер int массива 512 мегабайт.

Кэш: гранулярность



Результаты прогона на Intel Core M с кэшем L2 в 256 килобайт на ядро и общим кэшем L3 в 4 мегабайта, размер int массива 512 мегабайт.

Кэш: фальшивое разделение

- Неожиданная проблема: многопоточная программа написана как будто корректно, но при этом совершенно неоптимально с точки зрения использования кэш-памяти.
- Это случается, когда одна линия кэша содержит данные, относящиеся к разным потокам.
- Каждая модификация данных одним потоком вызывает сброс кэша для второго потока (и его постоянное перечитывание из памяти).
- Данные реально не разделяются, но накладные расходы как при разделении. Проблема не в неверном алгоритме, а именно в гранулярности кэшей и протоколе синхронизации.

Кэш: фальшивое разделение

```
int sumLocal[N_THREADS];
void ThreadFunc(thread_data *p) {
    int id = p->threadId;
    sumLocal[id] = 0;
    ...
    for (i=0; i<N; i++)
        sumLocal[id] += p->q[i];
}
```

- Поток 1 пишет в свой элемент своей копии линейки.
- Эта линейка инвалидируется во потоке 2.
- Поток 2 пишет в свою линиейку. Текущая — `Invalid`, перед записью она считывается из памяти,они какое-то время `Shared`.
- Поток 2 записывает в свою линейку. Для потока 1 она становится `Invalid`.
- Каждая операция записи в каждый кэш приводит к инвалидизации другой и перечитывании этой линейки другим потоком. Cache trashing.

Ещё раз про Interconnect и NUMA

```
//Thread 1.
void *threadFunc1(void *arg) {
    ...
    int *pFlag = getFromRemote();
    ...
    while (*pFlag == 0)
        ;
    ...
}
```

```
//Thread 2.
void *threadFunc2(void *arg) {
    int flag = FALSE;
    ...
    giveToRemote(&flag);
    ...
    flag = TRUE;
}
```

Первый поток получает адрес переменной, принадлежащей второму потоку.
Второй поток исполняется на ядре в другом процессорном домене.
Информация должна пройти через:

1. общую шину;
2. локальную шину второго ядра;
3. кэш второго ядра;
4. память второго ядра;
5. общую шину;
6. кэш первого ядра и только затем процессор первого ядра

Особенности NUMA

- Означает ли это, что NUMA архитектуры не подходят для практического использование?
- NUMA может быть на той же элементной базе производительнее архитектуры SMP.
- Если исполняющийся поток в основном использует данные только своей, локальной памяти, то накладные расходы на поддержание консистентности кэшей будут отсутствовать и время доступа к локальной памяти в NUMA системах можно сделать меньше, чем в SMP системах.
- Программы должны быть NUMA-aware!
- Эффективное программирование на NUMA системах сходно с программированием распределённых систем.
- Каждое ядро можно рассматривать как процесс распределённой системы, каждое обращение к памяти другого ядра может трактоваться как посылка сообщения между процессами.

Основные узкие места в вычислительных системах

Узкие места в вычислительных системах: ретроспекция

- 1985 год. Средняя вычислительная система обладает сравнительно небольшим количеством оперативной памяти. Только что для массовых вычислительных систем появилась виртуальная память, но даже специалисты не понимают принцип локальности и она используется плохо.
- 1995 год. Оперативной памяти много. Узким местом становится процессор, количество одновременно запускаемых инструкций. Кэш был интегрирован в процессор, появились зачатки суперскалярности (Pentium), память оказывалась способной выполнять запросы со стороны процессора с приемлемой скоростью.
- 2005 год. Процессор стал быстрым, суперскалярным и конвейерным. Тактовая частота стала выше, количество исполняемых за такт инструкций тоже возросло (Athlon, Core2). Латентность памяти почти не изменилась, теперь требуется больше тактов! Опять узкое место — память. Наибольшая проблема — промахи кэшей, низводящие скорость до значений

Проблемы производительности современных программ

- Много косвенной адресации.

  ```
  a->b->c = d->e;
  ```

  Следствие — много промахов кэша.
- Много уровней абстракции.
  Классический пример — протокол SOAP. Транспортный уровень — XML. Уровень представления в приложении — DOM, много уровней указателей. Много уровней преобразования. Всё плохо отражается на кэш.
  Объектно-ориентированное программирование, возможно, экономит время на разработку программ, но замедляет их исполнение.

Особенности системы команд x86/x64

Система команд x86/x64

Много внутреннего параллелизма на уровне инструкций. Особенности:

- много исполнительных блоков. `AGU`,`ALU`,`BP`.
- длинный конвейер.
- суперскалярное исполнение.
- спекулятивное исполнение.
- умозрительное исполнение инструкций. Наиболее важен не факт обязательного исполнения конкретной машинной инструкции, а факт видимости получившихся результатов.

Система команд x86/x64

- Большие буфера для переупорядочивания и регистры переименования. Исполняется следующая последовательность инструкций (сумма элементов массива arr):

```
    xor ecx,ecx
    mov eax,offset arr
    xor ebx,ebx
@loop:
        mov edx, [eax+ecx]
        add ebx, edx
        add ecx, 4
        cmp ecx, 1024
        jl @loop
@loopend:
```

Переупорядочивание запросов

- Пусть в кэш-памяти элементы с 16-го по 31-й, размер линейки равен 64 а остальных элементов в кэше нет. Тогда в очереди на исполнение будут следующие команды:

```
mov edx, [eax+0]
add ebx, edx
...
mov edx, [eax+4]
add ebx, edx
...
...
mov edx, [eax+64]
add ebx, edx
...
```

Переупорядочивание запросов

- Первой инструкции обращается к памяти $\rightarrow$ результат задержан.
- Задержаны и следующие 15 команд (та же линейка кэша).
- А вот 16-я команда данная команда может поступить в исполнение *перед* тем, как завершатся предыдущие 15 команд.
- Каждая из команд будет использовать свою копию регистра `edx` и инструкции будут переупорядочены для более оптимального исполнения.
- Для этого потребуется буфера переупорядочения и пул регистров переименования.
- Иногда применяется термин *теневые регистры*, он не совсем точен.

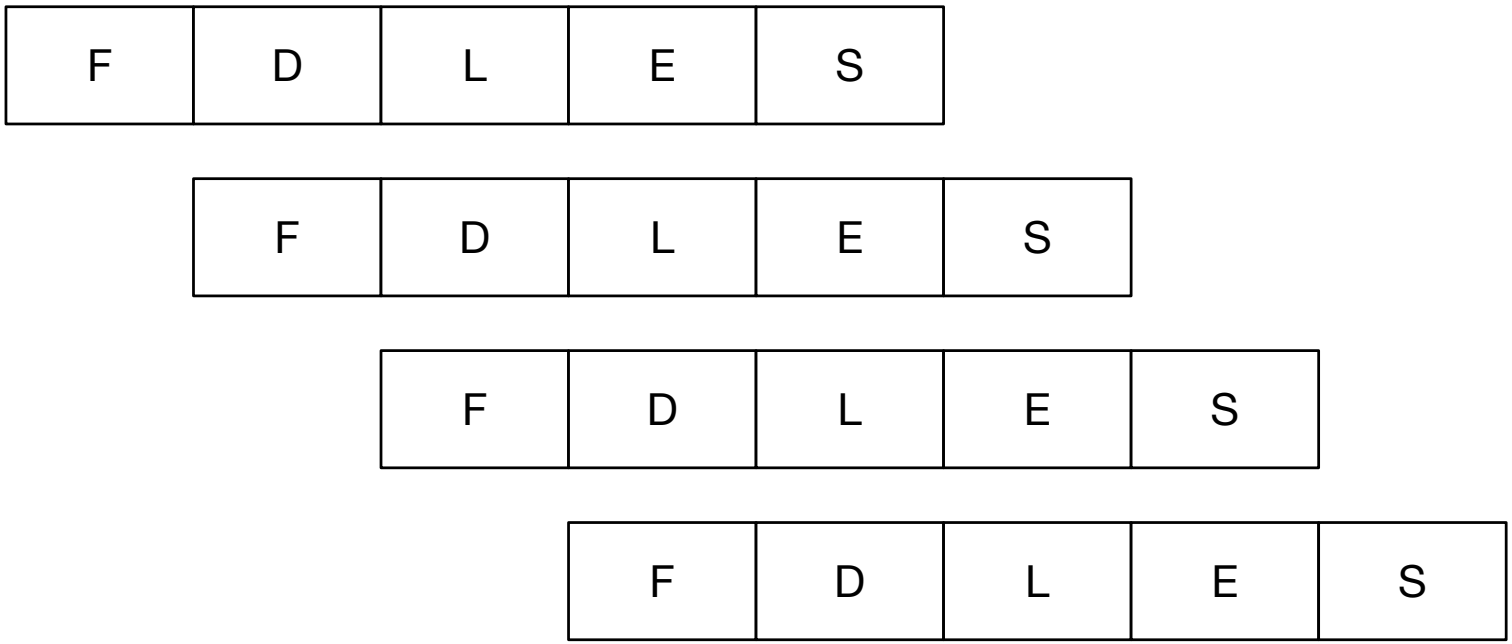
Система команд x86/x64: особенности

- многоуровневое кэширование.
- низкий уровень кэш-промахов. При кэш-памяти до 24 мегабайт количество кэш-промахов в относительно прилично написанной программе не может быть очень велико.
- высокая цена промаха.
- разные задержки при обращении к инструкциям и данным в различных уровнях кэша:
  - регистры $\leqslant 1$
  - L1 кэш $\approx 2 - 3$
  - L2 кэш $\approx 7 - 10$
  - L3 кэш $\approx 25 - 35$
  - RAM ≈ 200

CPU: конвейер

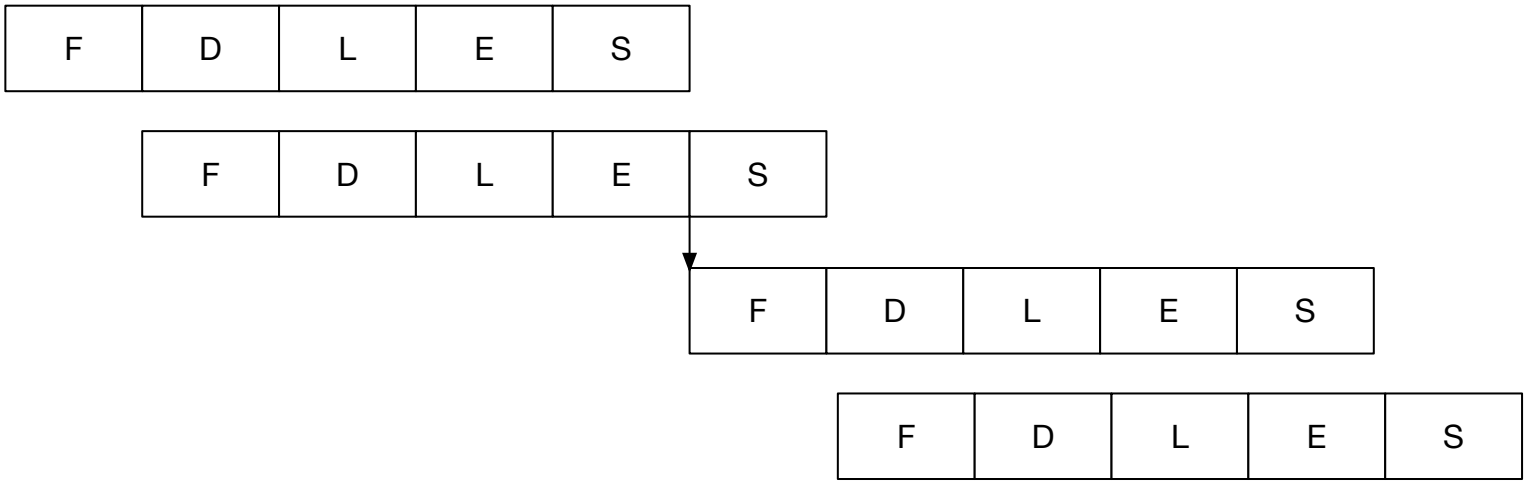
- У `Intel` традиционно много ступеней конвейера.
- Больше ступеней $\rightarrow$ больше достижимая тактовая частота.
- Традиционные ступени:
  1. Выборка команды
  2. Декодирование команды
  3. Выборка операндов из памяти
  4. Исполнение
  5. Занесение результатов в память.

CPU: конвейер: последовательное исполнение



Конвейер команд при последовательном исполнении.

CPU: конвейер: исполнение переходов



Конвейер команд при наличии команд перехода.

- Переходы очень плохо влияют на производительность конвейера.
- Процессор вынужден прервать конвейерную цепочку.
- Для борьбы с этим применяют предсказание переходов. Производится выборка команды по наиболее ожидаемому маршруту. Если переход не осуществился — откат.

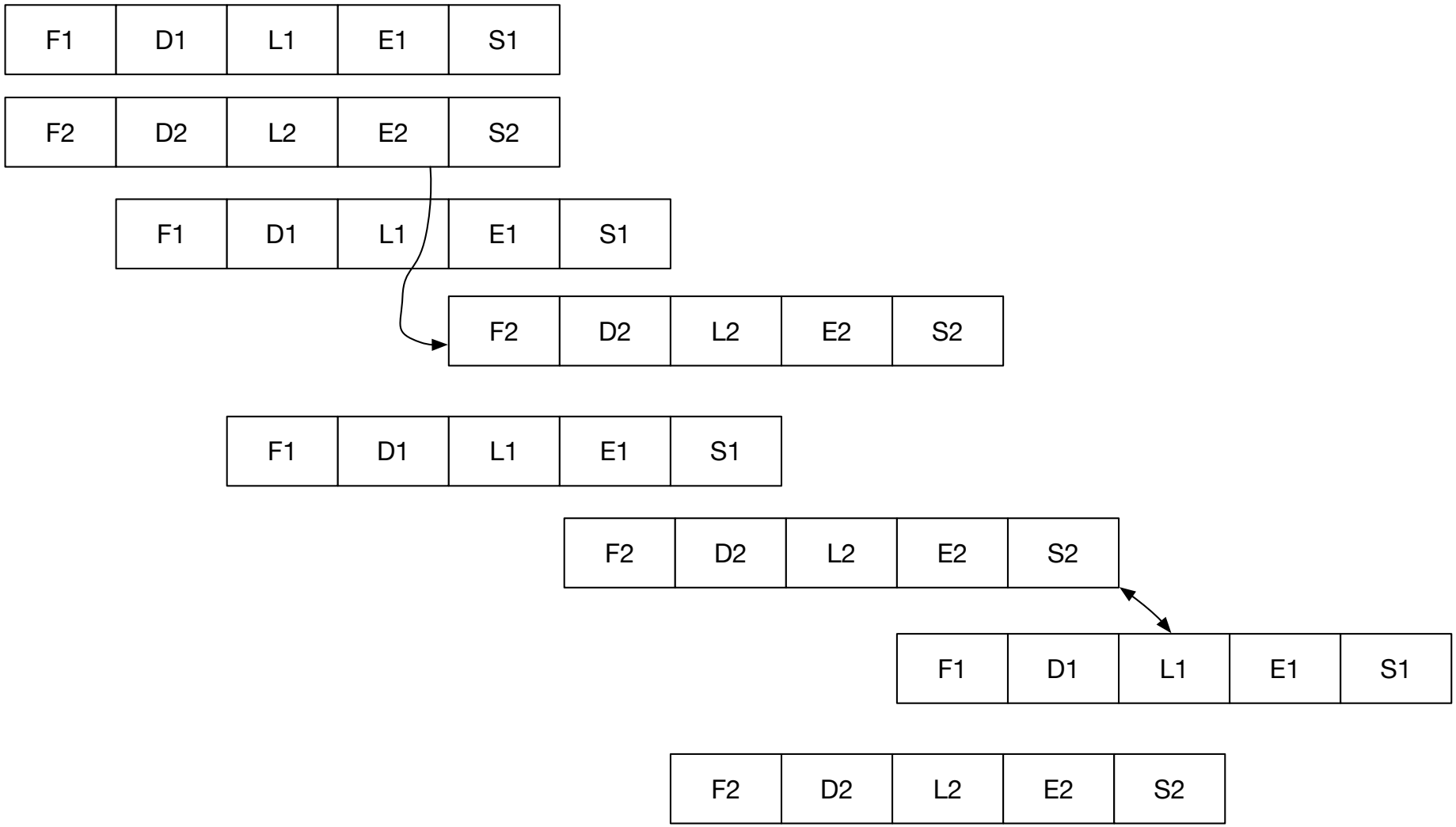
CPU: суперскалярное исполнение

- На разных исполнительных устройствах в один момент времени могут исполняться различные машинные команды.
- В начале очередного такта на исполнение могут быть переданы несколько команд.
- Процессор `Pentium` имел два конвейерами `u` и `v`.
- Конвейер `u` мог исполнять любую команду.
- При определённых условиях, следующую команду мог взять на исполнение конвейер `v`.
- Сейчас имеется 3-4 конвейера и одновременно могут запускаться до 4 команд, исполняющихся параллельно.

CPU: суперскалярное исполнение

- Возможны несколько экземпляров каждого исполнительного устройства.
- Они могут исполнять работу одновременно.
- Зависимость по данным: одно устройство должно дожидаться завершения другого.

CPU:суперскалярное исполнение с зависимостью по данным



Суперскалярное исполнение с зависимостью по данным.

CPU: спекулятивное исполнение

- В обычном коде примерно пятая часть команд — команды перехода, безусловного или условного (вызовы функций тоже являются командами перехода).
- При длинном конвейере команды перехода — зло.
- Если не планировать исполнение, то каждая команда перехода приводила бы к сбросу конвейера и колоссальным задержкам в исполнении.
- При условной операции (`if`) конвейер заранее не знает, осуществится ли переход.
- Исполняются сразу две ветви, одна — соответствующая истинному значению, вторая — ложному.
- После того, как код условия получен, спекулятивное исполнение соответствующей ветви исполнения прекращается и регистры актуализируются.

CPU: умозрительное исполнение инструкций

- Наиболее важен факт видимости результата команды. Обязательного исполнения конкретной машинной инструкции не требуется.

```
mov ebx, [esp+8] ; a
mov eax, ebx
mov ebx, [esp+12] ; b
```

результат — помещение значения переменной a в регистр eax, а переменной b в регистр ebx. Флаги процессора остаются неизменными.

CPU: умозрительное исполнение инструкций

- Процессор при исполнении может заменить первые две инструкции на

```
mov eax, [esp+8] ; a
mov ebx, [esp+12] ; b
```

- видимый результат исполнения окажется тем же самым.
- Это важно — регистров мало.

Особенности системы команд Intel

- Много этапов в историческом развитии.
- Система команд 16-разрядного процессора 8086 проектировалась для лёгкого переноса кода с 8-битного процессора 8080.
  - Крайне малое количество регистров
  - Разбиение 16-битных регистров на 8-ми битные подрегистры.
  - Сегментные регистры — кошмар для программистов.
  - Не все регистры одинаково полезны, косвенно адресоваться можно было только для `bx`,`bp`,`si`,`di`
- Введение 32-битной архитектуры. Опять совместимость.
  - Количество регистров осталось 7.
  - Они стали более-менее равноправными.
- 64-х битная архитектура. Количество регистров стало 15. Сохранилась возможность использовать 32-разрядные регистры, так же как и 16 и 8 разрядные.

Спасибо за внимание.

Следующая тема — архитектурные особенности современных компьютеров. Упорядоченность памяти. Атомарность.