Низкоуровневое программирование

Лекция 7

Кэш-память

Суперскалярная архитектура ЦП

Многоядерные ЦП

Программная модель ЦП

Под программной моделью ЦП (programming model, programmer’s model) описание правил функционирования ЦП, задаваемое в ISA.

В программную модель включаются набор регистров ЦП и ограничения на поведение ЦП в различных ситуациях (что должно происходить, и что не должно).

Микроархитектура ЦП, определяющая его внутреннее устройство и функционирование, не должна противоречить программной модели.

При этом фактический процесс функционирования ЦП может крайне существенно отличаться от описанного в программной модели – важно, чтобы совпадало видимое поведение ЦП.

Возможные расхождения с реальностью

К настоящему моменту может сложиться ложная уверенность, что:

- У процессора всего 14 регистров общего назначения;
- Инструкции выполняются последовательно;
- В каждый момент времени выполняется только 1 инструкция;
- Операции чтения/записи постоянны по времени;
- Время работы не зависит от обрабатываемых данных;

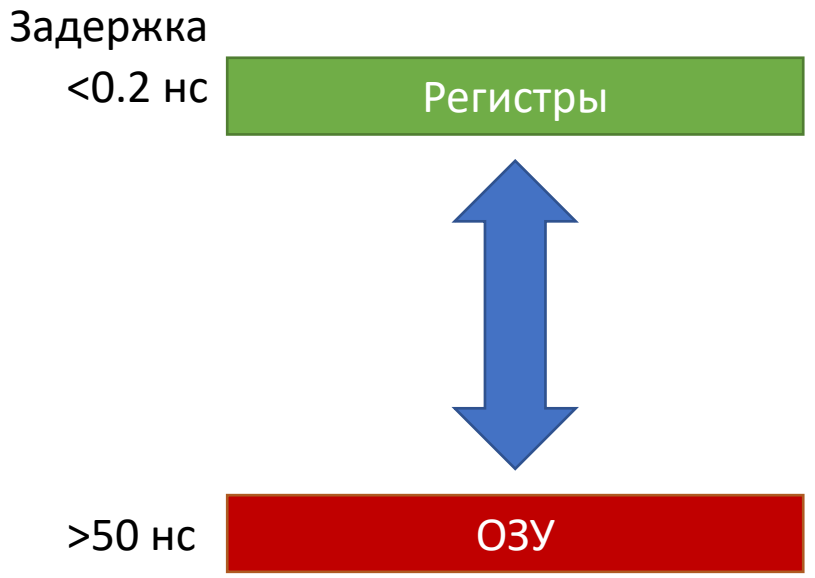
Любое из этих утверждений может быть ложным ☺.

Производительность памяти

С точки зрения программиста на ассемблере существует 2 уровня памяти – оперативная память и регистры.

Производительность регистров в сотни раз превосходит производительность памяти, но их суммарный объем крайне мал.

Оперативная память обладает большим объемом, но обращения к ней существенно «тормозят» процессор.

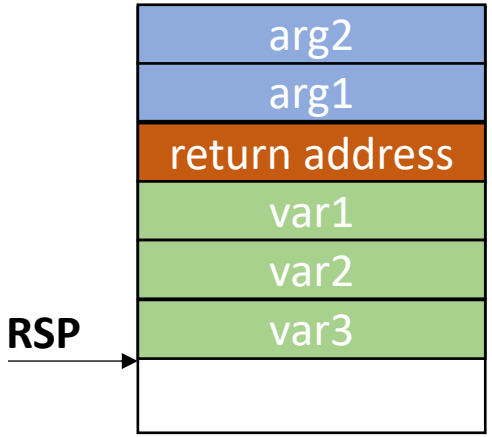


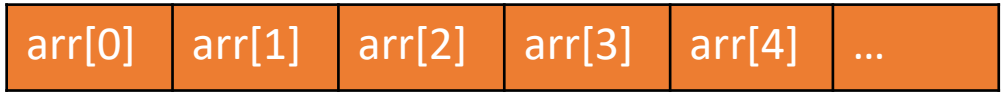
Локальность данных

Обычно данные, которые обрабатываются программой, обладают хорошей **локальностью**: они сгруппированы, а не разбросаны по адресному пространству.

Хорошим примером является стек: все локальные переменные расположены а текущем кадре стека.

Отсюда следует, что *если мы прочитали из памяти одну порцию данных, то скорее всего следующая порция будет прочитана из соседних адресов.*



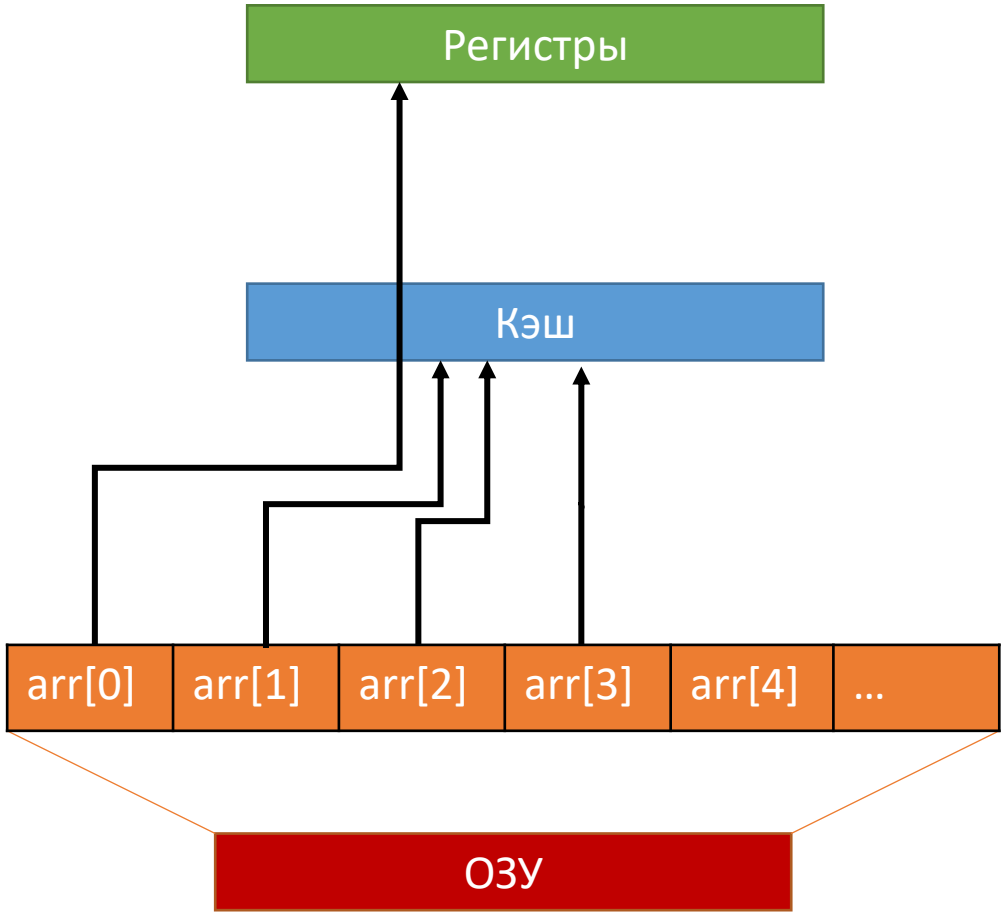


Кэш-память

Исходя из локальности данных, можно ускорить обращение с памятью, заранее читая не только запрошенные данные, но вообще все данные в окрестности запрошенного адреса (**упреждающее чтение**)

Для сохранения таких заранее прочитанных данных используется промежуточная память, которая работает медленнее регистров, но быстрее оперативной памяти. Данная память называется **кэш-памятью**.

Данные считываются в кэш блоками фиксированного размера – **кэш-линиями**. Обычно кэш-линия на современных ЦП имеет размер 64 байта. Кэш-линии не перекрываются, начало кэш-линии выровнено по ее размеру (т.е. по 64 байтам).



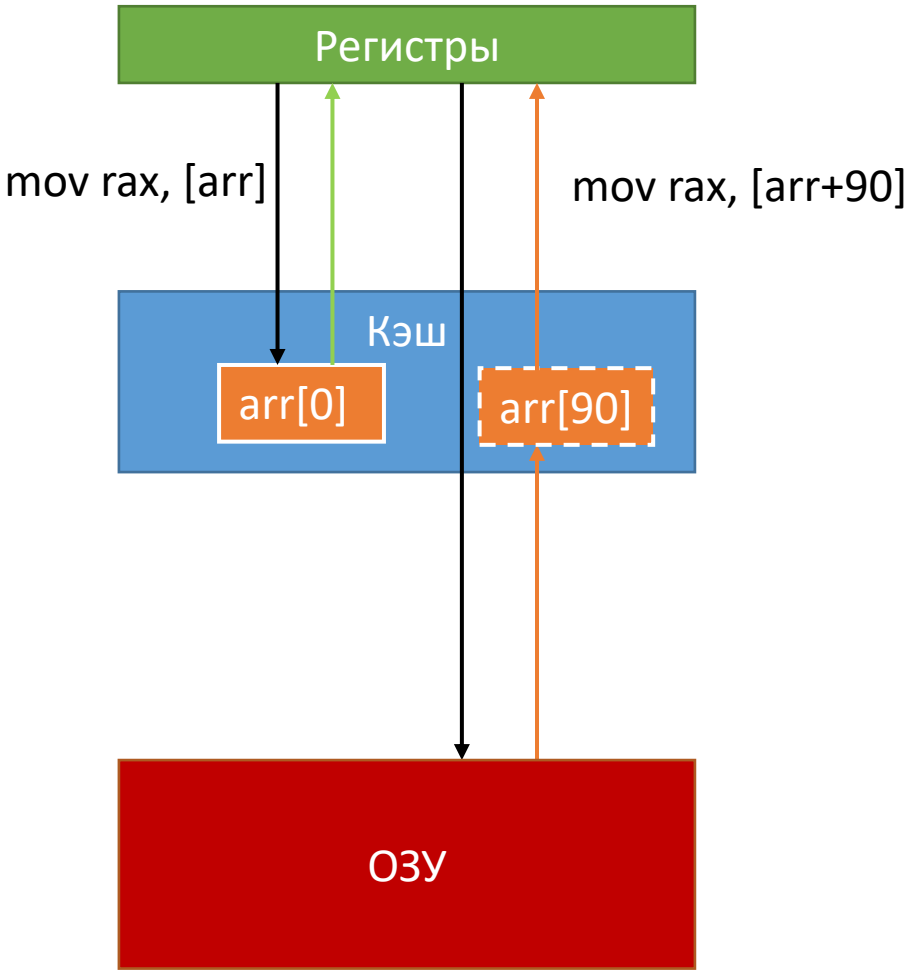
Попадания, промахи и вытеснения

Если при обращении в память окажется, что данные уже находятся в кэше (**попадание в кэш, cache hit**), то данные будут взяты из кэша.

Если запрошенных данных в кэше не окажется (**промах кэша, cache miss**), то произойдет чтение данных из оперативной памяти.

Если кэш заполнен, то при чтении новых данных производится **вытеснение** кэш-линии. Обычно вытесняется наиболее «старая» кэш-линия*.

*если при проектировании ЦП не была выбрана иная стратегия



Запись в кэш. Write-through и Write-back

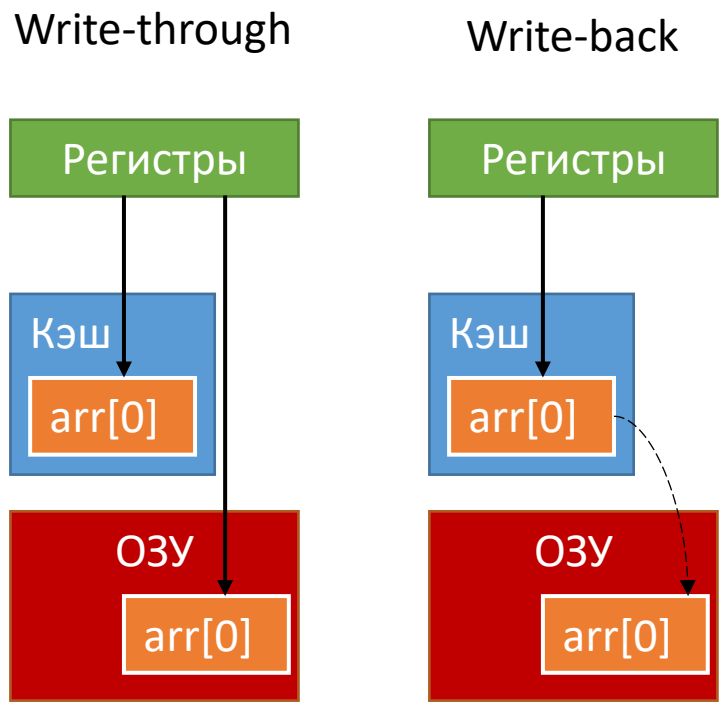
Если выполняется операция записи, и данные находятся в кэше, то изменяются данные в кэше. Момент изменения данных в ОЗУ зависит от используемой политики записи:

- Write-through кэширование – операция записи выполняется одновременно и в кэш, и в ОЗУ;
- Write-back кэширование – запись выполняется только в кэш, данные в ОЗУ обновляются при вытеснении измененной кэш-линии.

Плюсом write-through кэширования является простота реализации.

Плюсом write-back кэширования является производительность, минусом – сложность организации в многоядерных ЦП.

В ЦП x86-64 в большинстве случаев используется write-back кэширование.



Уровни кэш-памяти. Кэш L1

В современных процессорах используется **несколько уровней кэша**. Чем меньше уровень кэша, тем меньше его объем и тем выше его производительность.

Кэш L1 по производительности немногим уступает регистрам (задержка около 5 тактов).

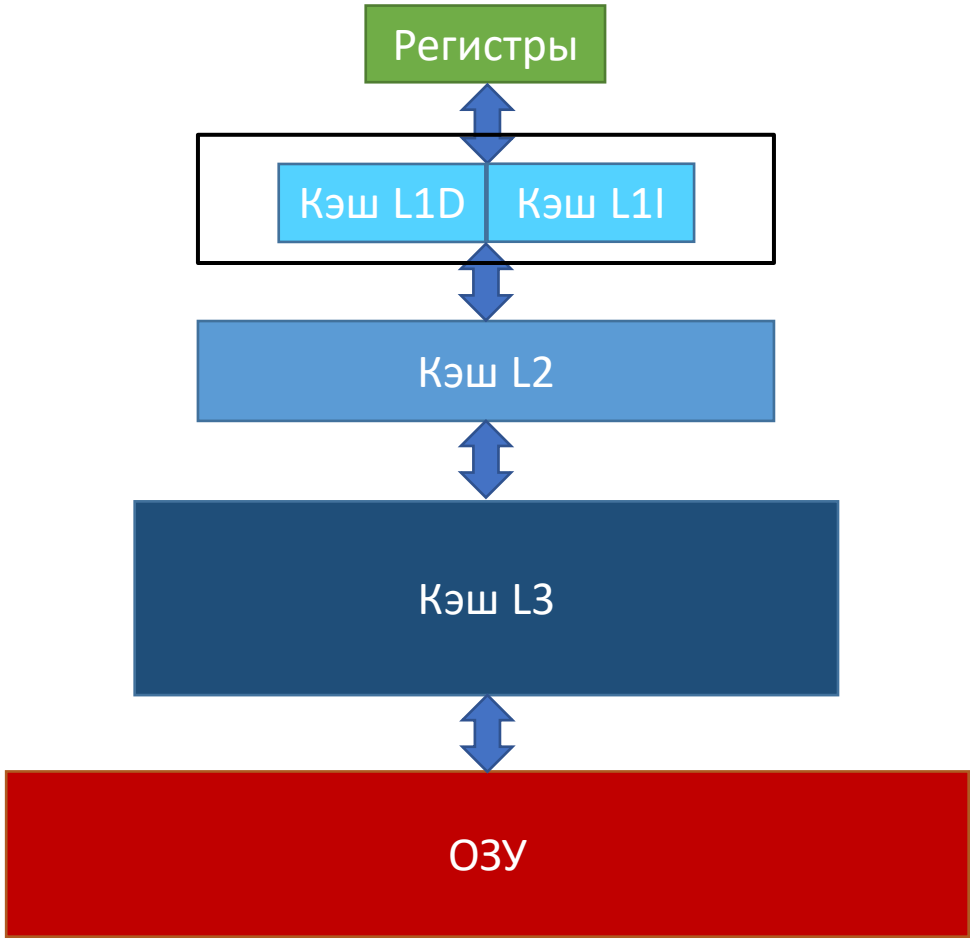
Типичный объем – 64 Кбайт на 1 ядро процессора.

Кэш уровня L1 состоит из 2 частей: **кэша инструкций** (L1I) и **кэша данных** (L1D).

При таком разделении память с инструкциями программы не будет вытесняться из кэша при интенсивном чтении данных.

Т.к. инструкции тоже подвергаются кэшированию, *код с высокой локальностью исполняется быстрее.*

По этой же причине компиляторы используют более короткие инструкции, если возможно (*xor eax, eax* вместо *xor rax, rax* или *mov eax, 0*)



См. также: ассоциативность кэша, виктимный кэш, атрибуты [[likely]] и [[unlikely]]

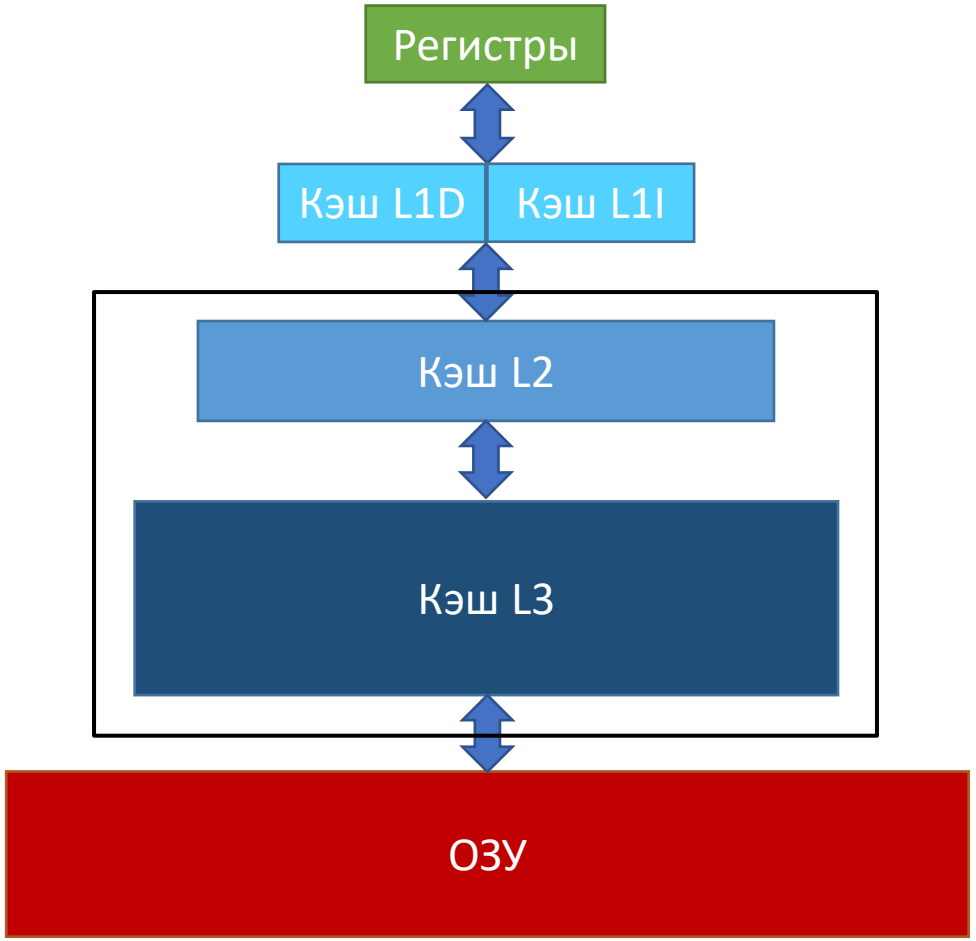
Кэши L2 и L3

Кэш L2 в современных процессорах имеет объем 256-1024 Кбайт *на ядро* процессора.

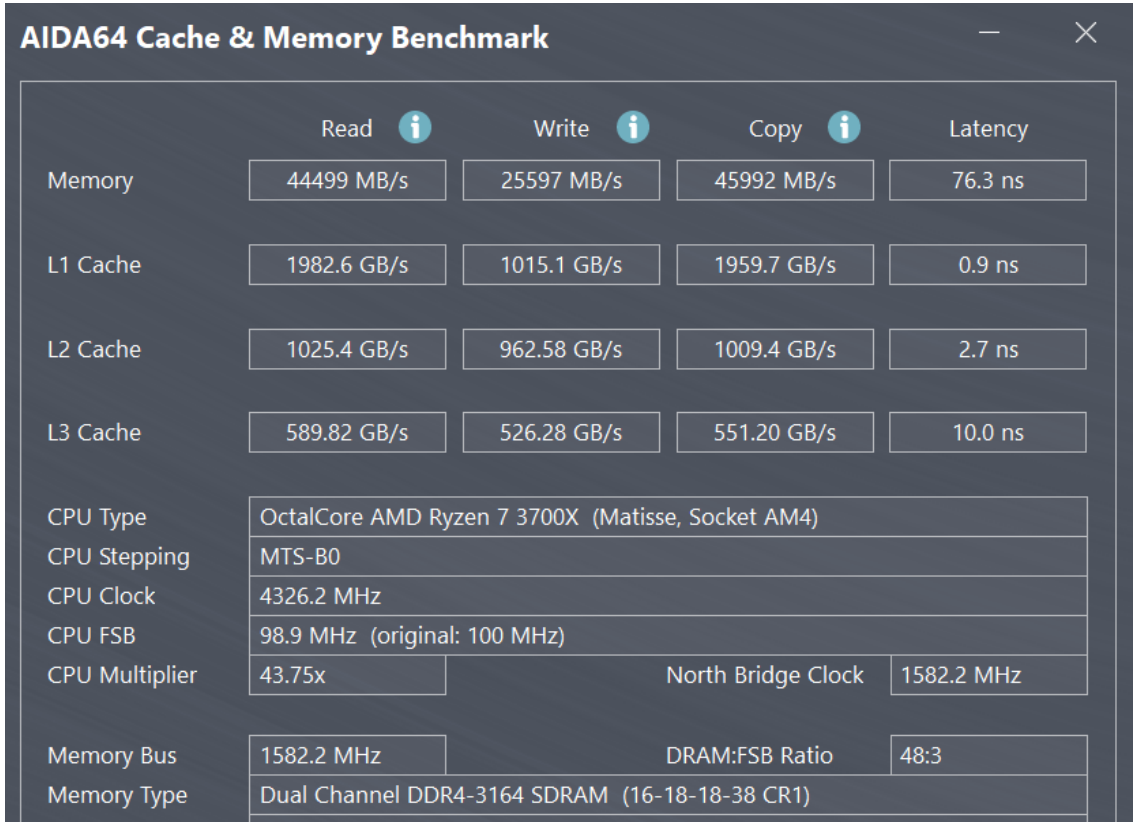
Задержка доступа – 10-12 тактов.

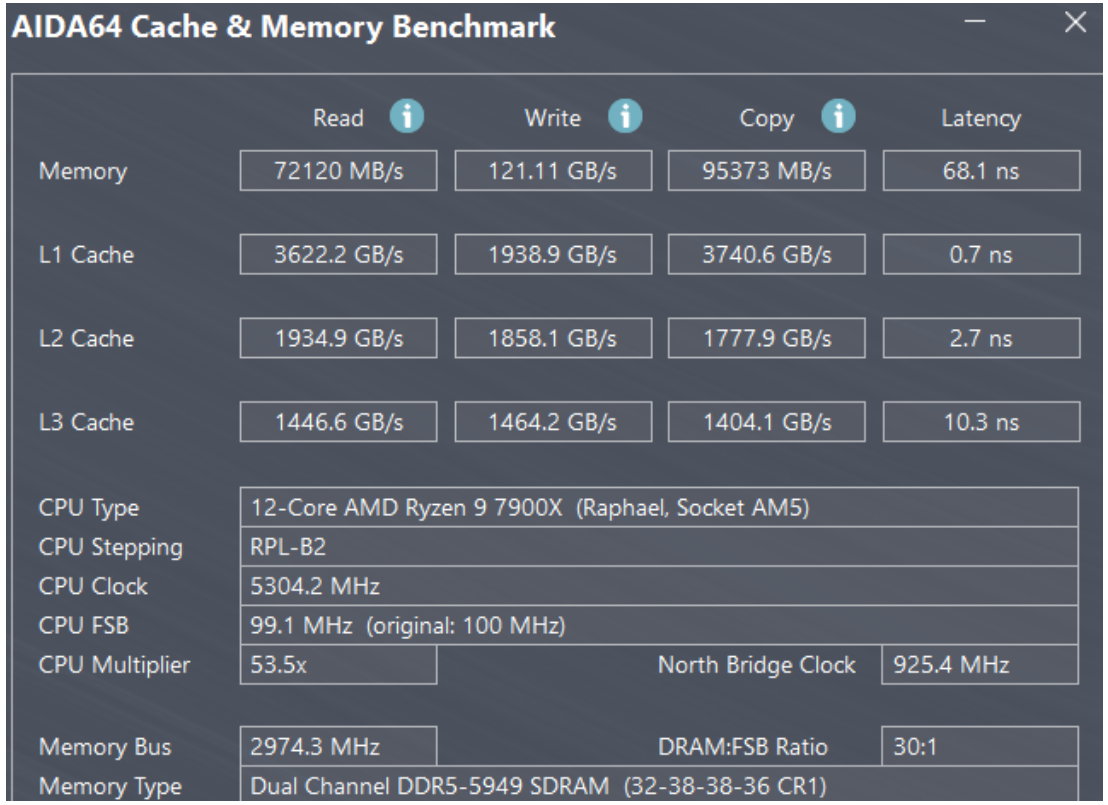
Кэш L3 - общий либо для всех ядер, либо для группы ядер, и имеет размер 1-128 МБ.

Задержка доступа – 30-40 тактов.



Кэш-память


AIDA64 Cache & Memory Benchmark
Read
Write
Copy
Latency
Memory
44499 MB/s
25597 MB/s
45992 MB/s
76.3 ns
L1 Cache
1982.6 GB/s
1015.1 GB/s
1959.7 GB/s
0.9 ns
L2 Cache
1025.4 GB/s
962.58 GB/s
1009.4 GB/s
2.7 ns
L3 Cache
589.82 GB/s
526.28 GB/s
551.20 GB/s
10.0 ns
CPU Type
OctalCore AMD Ryzen 7 3700X (Matisse, Socket AM4)
CPU Stepping
MTS-B0
CPU Clock
4326.2 MHz
CPU FSB
98.9 MHz (original: 100 MHz)
CPU Multiplier
43.75x
North Bridge Clock
1582.2 MHz
Memory Bus
1582.2 MHz
DRAM:FSB Ratio
48:3
Memory Type
Dual Channel DDR4-3164 SDRAM (16-18-18-38 CR1)


AIDA64 Cache & Memory Benchmark
Read
Write
Copy
Latency
Memory
72120 MB/s
121.11 GB/s
95373 MB/s
68.1 ns
L1 Cache
3622.2 GB/s
1938.9 GB/s
3740.6 GB/s
0.7 ns
L2 Cache
1934.9 GB/s
1858.1 GB/s
1777.9 GB/s
2.7 ns
L3 Cache
1446.6 GB/s
1464.2 GB/s
1404.1 GB/s
10.3 ns
CPU Type
12-Core AMD Ryzen 9 7900X (Raphael, Socket AM5)
CPU Stepping
RPL-B2
CPU Clock
5304.2 MHz
CPU FSB
99.1 MHz (original: 100 MHz)
CPU Multiplier
53.5x
North Bridge Clock
925.4 MHz
Memory Bus
2974.3 MHz
DRAM:FSB Ratio
30:1
Memory Type
Dual Channel DDR5-5949 SDRAM (32-38-38-36 CR1)

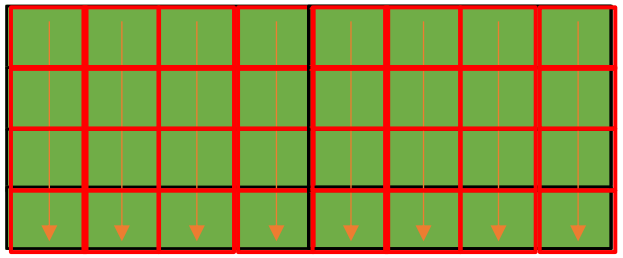
Кэш-память (примеры)

Обход матрицы по строкам

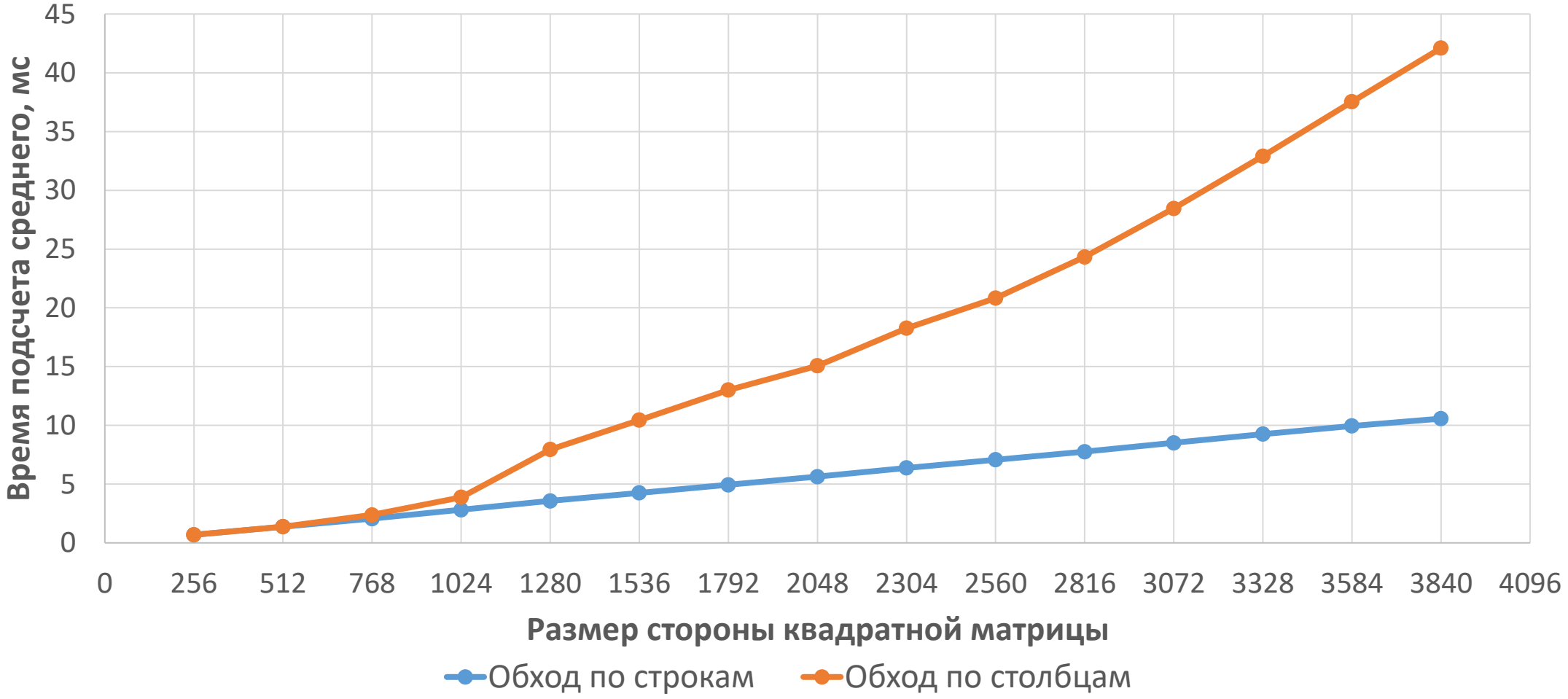
```
float avg_by_row(float* arr, int rows,
                                int cols) {
    float result = 0;
    for (int i = 0; i < rows; ++i)
        for (int j = 0; j < cols; ++j)
                result += arr[i*rows +j];
    return result / (rows * cols);
}
```

Обход матрицы по столбцам

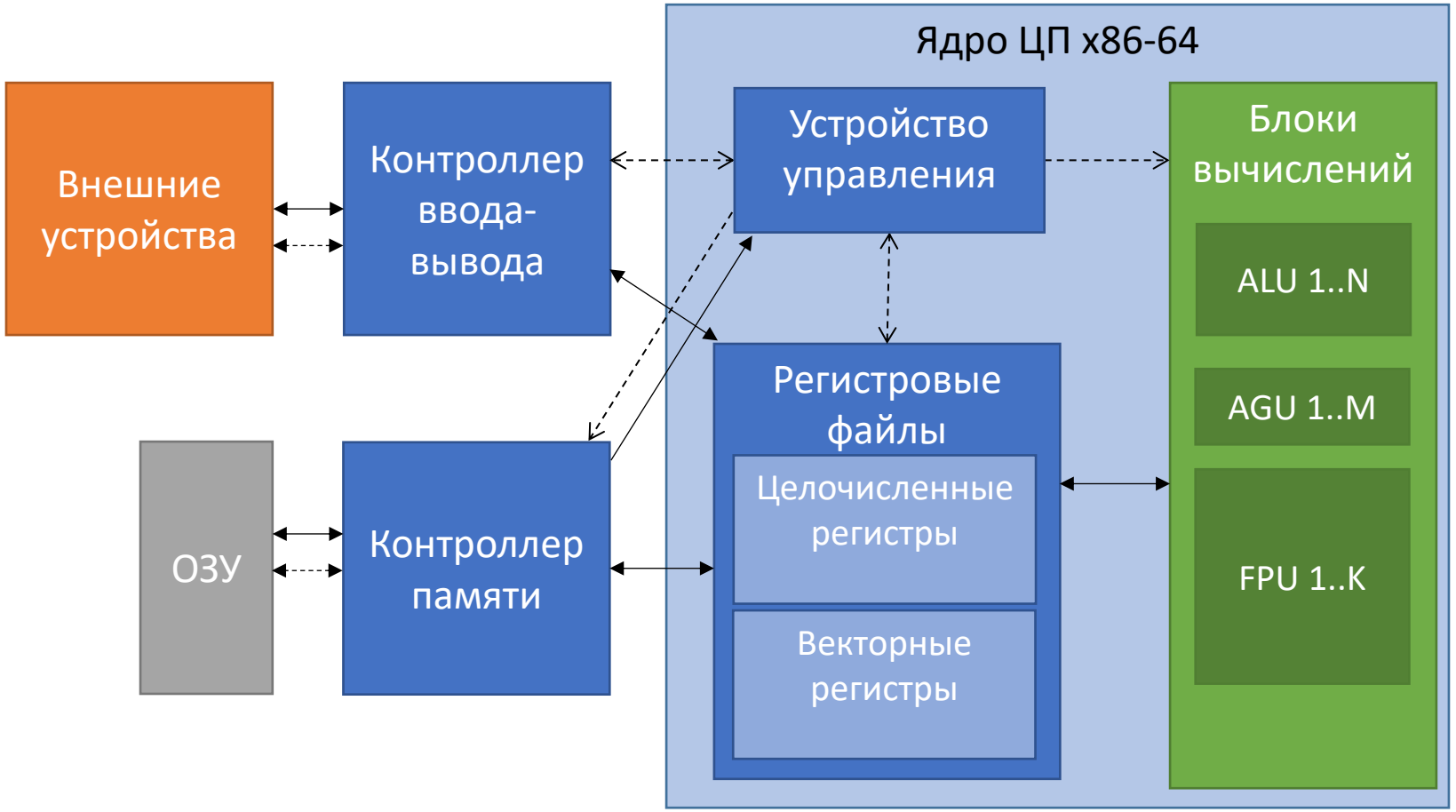
```
float avg_by_col(float* arr, int rows,
                                int cols) {
    float result = 0;
    for (int j = 0; j < cols; ++j)
        for (int i = 0; i < rows; ++i)
            result += arr[i*rows +j];
    return result / (rows * cols);
}
```



Кэш-память


Время подсчета среднего, мс
45
40
35
30
25
20
15
10
5
0
0
256
512
768
1024
1280
1536
1792
2048
2304
2560
2816
3072
3328
3584
3840
4096
Размер стороны квадратной матрицы
Обход по строкам
Обход по столбцам

Общая архитектура ядра ЦП



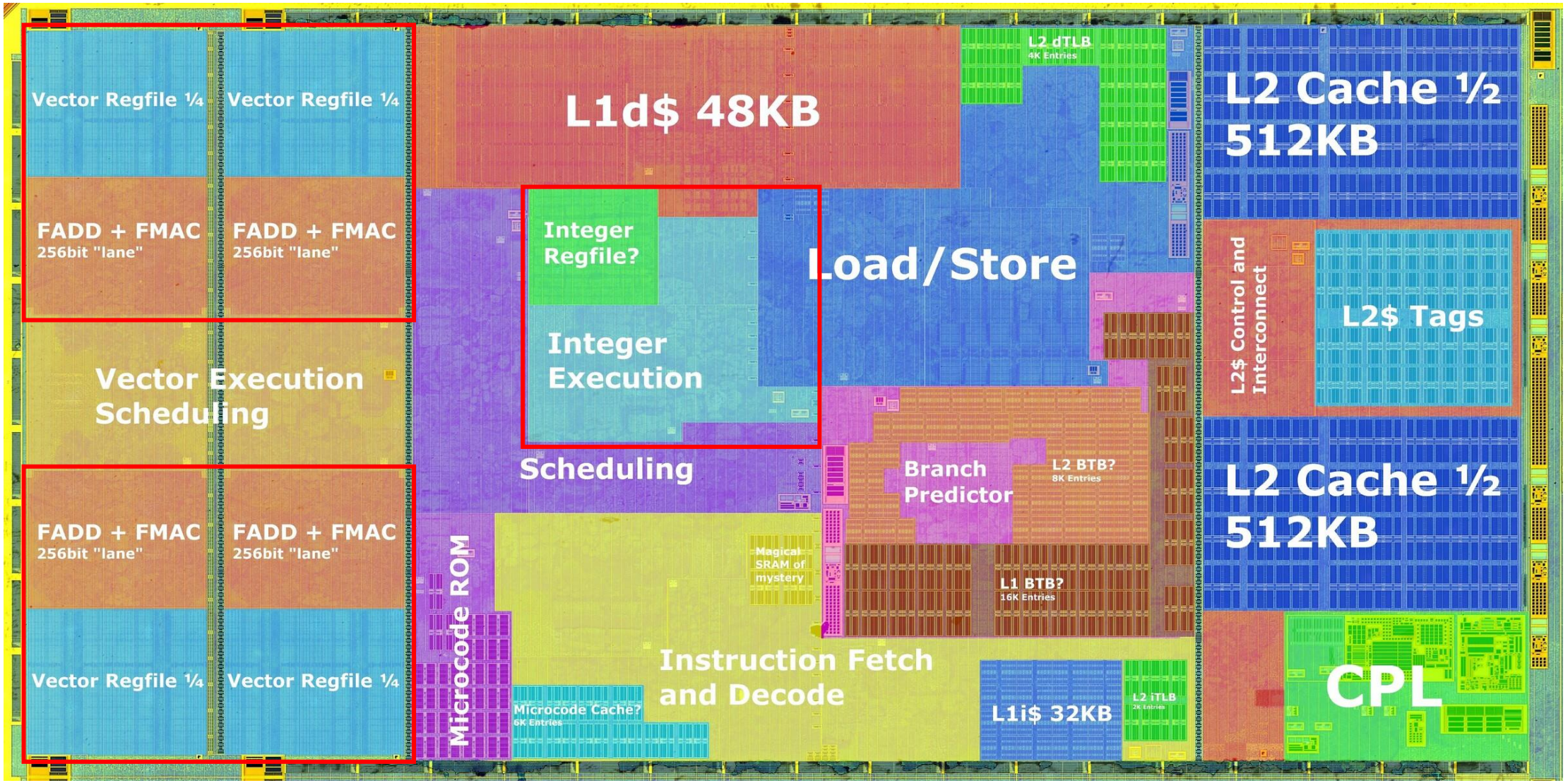
ALU
(Arithmetic Logic Unit)
отвечает за целочисленные вычисления

AGU
(Address Generation Unit)
отвечает за вычисление адреса в адресных выражениях

FPU
(Floating Point Unit)
отвечает за вещественные вычисления

Устройство ядра ЦП

Ядро Zen 5



За выполнение вещественных операций отвечают области, отмеченные FADD+FMAC в левой части;
за выполнение целочисленных операций – Integer Execution в центре.
чем занимается все остальное?

Выполнение инструкций процессором

Между считыванием инструкции из памяти и получением результата проходит несколько тактов (дискретных промежутков времени между импульсами тактового генератора).

Инструкция проходит несколько этапов:

1. **Чтение /выборка**(fetching - инструкция считывается из ОЗУ или кэша);
2. **Декодирование** (decoding – определение требуемого действия);
3. **Выполнение** (execution - выполнение операций соответствующими устройствами);
4. **Запись результата** (write-back/retire– запись результата в регистры/память).

Сколько тактов выполняется конкретная инструкция, определяется микроархитектурой ЦП. К примеру, на Zen 1 инструкция DIV выполняется за 14-47 тактов, а на Zen 4 – за 10-18 тактов (деление является одной из самых долгих инструкций целочисленной арифметики).

При этом, каждая инструкция, в зависимости от типа, выполняется на конкретном **исполнительном устройстве** – ALU, AGU, FPU. Инструкции, операндом которых является адресное выражение, сначала загружают значение из памяти, затем выполняют вычисления в устройстве выполнения (и затем, если необходимо, сохраняют результат в память).

https://agner.org/optimize/instruction_tables.pdf

Чтение инструкций

Чтение инструкций, против ожиданий, является достаточно сложным процессом.

В архитектурах семейства x86 инструкции имеют переменную длину. При этом чтение данных обычно выполняется блоками по 8 байт.

Когда процессор считывает очередной 8-байтовый блок с кодом программы он должен разбить блок на инструкции. При этом возможна ситуация, когда инструкция на конце блока считана не вся – в этом случае ее конец нужно взять из следующего блока.

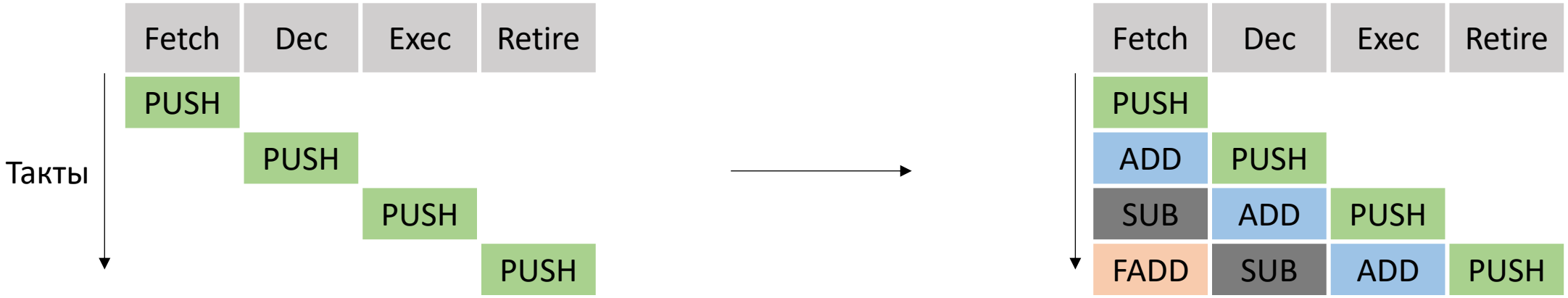
Конвейеризация.

Поскольку каждая инструкция проходит данные этапы независимо, возможно проводить считывание одной инструкции, пока происходит декодирование другой и т.д.

Подобная схема с одновременным выполнением этапов называется **конвейером**.

В простом конвейере 4 крупных этапа (Fetch-Decode-Execute-Retire), которые потенциально могут быть разбиты на более мелкие этапы (в ЦП Intel микроархитектуры Prescott – 31 этап!).

Конвейеризация позволяет ускорить выполнение участков последовательного кода.
Условные переходы и переходы по неконстантным значениям (`JMP [EAX]`) не ускоряются, т.к. заранее неизвестно, какая инструкция будет считана следующей.

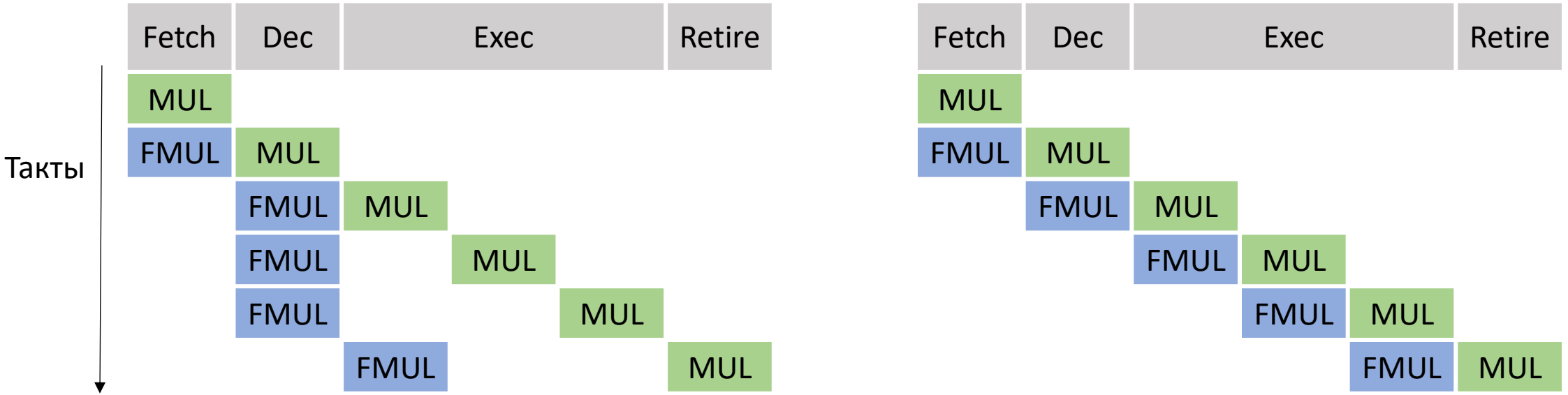


Суперскалярная архитектура (пример)

ЦП имеет несколько устройств управления, независимых друг от друга (например, ALU и FPU).

Если 2 инструкции используют разные исполнительные устройства и *не зависят* друг от друга (например, используют разные регистры), то они могут перекрываться по времени выполнения. Процессоры, построенные по такому принципу, называют **суперскалярными**, т.к. в них одно ядро может выполнять одновременно несколько операций за один такт.

В такой ситуации, логично продублировать АЛУ и FPU, чтобы процессор мог обрабатывать несколько однотипных операций одновременно.



Микрооперации

В архитектуру набора команд x86-64 входит более тысячи инструкций длиной от 1 до 15 байт. Часто одна инструкция может выполнять несколько действий.

Прямое декодирование сложной инструкции в управляющие сигналы затратно – потребуется огромное количество логических элементов. Кроме того, если инструкция осуществляет несколько действий, она занимает несколько исполнительных устройств.

Вместо этого в декодер современных ЦП разбивает инструкцию на набор элементарных действий (**микроопераций**). Каждая микрооперация затрагивает только 1 блок выполнения. После декодирования микрооперации обычно аккумулируются в **очереди микроопераций**.

Введение микроопераций позволяет упростить схемотехнику ЦП и повысить производительность.

```
ADD [RBX], RAX
```

затрагивает контроллер памяти и АЛУ

декодирование →

```
MOV TMP, [RBX] ; АЛУ свободно
ADD TMP, RAX   ; КП свободен
MOV [RBX], TMP ; АЛУ свободно
```

Микрокод ЦП. Кэш микроопераций

Если инструкция разбивается на 2-3 микрооперации, то декодер выполняет разбиение самостоятельно на аппаратном уровне.

Для более сложных инструкций разбиение на микрооперации хранится в специальном **микрокоде ЦП**. Данный микрокод может обновляться -> возможно закрывать некоторые «баги» и уязвимости, возникшие при проектировании ЦП, а также повышать его производительность. Обновление микрокода может осуществляться ОС.

Микрокод ЦП загружается в ЦП при старте компьютера.

Поскольку чтение микроопераций из микрокода занимает некоторое время, после декодирования ЦП заносит результат в специальный **кэш микроопераций**. Если спустя некоторое время та же инструкция возникнет вновь, соответствующие микрооперации будут считаны из кэша.

Внеочередное выполнение

Поскольку после декодирования микрооперации от разных инструкций накапливаются в очереди микрооопераций, возможно выполнить перестановку операций в очереди с целью оптимальной загрузки исполнительных устройств. *При этом фактический порядок выполнения операций будет отличаться от порядка операций в программе* (**out-of-order**).

Микрооперация из очереди отправляется на выполнение, если известны значения ее операндов. Очевидно, если значение операнда рассчитывается какой-либо предыдущей микрооперацией - текущая не может начать выполнение, вместо нее на выполнение отправляется одна из следующих в очереди микроопераций.

На некоторых архитектурах внеочередное выполнение может влиять на корректность многопоточных программ. При этом на уровне машинного кода инструкции будут располагаться в корректном порядке.

```
MOV RCX, [R8] ; чтение (долгая операция)
ADD RCX, 1    ; зависит от предыдущей
SUB R9, 7     ; не зависит, выполнится раньше
```

Внеочередное выполнение

```
MOV RCX, [R8] ; чтение (долгая операция)
ADD RCX, 1    ; зависит от предыдущей
SUB R9, 7     ; не зависит, выполнится раньше
```

Такт	Fetch	Dec	Exec	Retire
1	MOV			
2	ADD	MOV		
3	SUB	ADD	MOV	
4		ADD SUB	MOV	
5		ADD SUB	MOV	
6		ADD SUB		MOV
7		SUB	ADD	
8			SUB	ADD

Такт	Fetch	Dec	Exec	Retire
1	MOV			
2	ADD	MOV		
3	SUB	ADD	MOV	
4		ADD SUB	MOV	
5		ADD	MOV SUB	
6		ADD		MOV SUB
7			ADD	
8				ADD

Зависимость по данным

Зависимости по данным запрещают переупорядочивание микроопераций в определенных ситуациях. В роли данных могут выступать переменная в ОЗУ или регистр.

1. Read-after-write (RaW) – данные записываются операцией А и считываются операцией Б => переупорядочивание может повлиять на результат операции Б;
2. Write-after-read (WaR) – данные считываются операцией А и затем перезаписываются операцией Б => переупорядочивание может повлиять на результат операции А;
3. Write-after-write (WaW) – данные записываются операцией А и перезаписываются операцией Б => может измениться поведение инструкций, считывающих данные между ними .

Поскольку регистр флагов перезаписывается инструкциями арифметики, он тоже может быть источником неявной зависимости по данным.

Разрыв зависимостей по данным. Переименование регистров. (пример)

Для увеличения производительности используется механизм **переименования регистров**, который позволяет избавиться от WaR/WaW-зависимостей по регистрам.

Если регистр перезаписан полностью, но последующие инструкции точно не имеют зависимости от старого значения (т.е. зависимость разорвана). Как следствие, ЦП может выбрать один из доступных «безымянных» регистров (их доступно несколько сотен), и использовать его в качестве заданного именованного регистра. При этом, одновременно 2 аппаратных регистра могут иметь одно «имя».

При полной перезаписи *всего* регистра инструкциями `MOV(SX/ZX)`, `XOR` (регистры общего назначения), `MOVP*/VMOVP*`, `PXOR/VPXOR` (векторные регистры) на аппаратном уровне ЦП просто выбирает свободный регистр и начинает использовать его. Компиляторы генерируют данные инструкции там, где они вроде-бы не требуются, именно для разрыва зависимостей.

```
DIV RDX         ; 1
MOV [RDI], RDX ; 1
XOR EDX,EDX     ; 2 (RDX перезаписан)
ADD EDX,ECX     ; 2
```

```
MOVPS XMM1, XMM0 ; 1
PXOR XMM0, XMM0  ; 2 (XMM0 перезаписан целиком)
MOVSS XMM0, XMM2 ; 2
```

См. также: алгоритм Томасуло

Разрыв зависимостей по данным. Переименование регистров. (пример)

```
DIV RCX          ; 1
MOV [RDI], RDX   ; 1
XOR EDX, EDX     ; 2
ADD EDX, ECX     ; 2
```

Такт	Fetch	Dec	Exec	Retire
1	DIV			
2	MOV	DIV		
3	XOR	MOV	DIV	
4	ADD	XOR	DIV	
5		ADD XOR		DIV
6		ADD XOR	MOV	
7		ADD XOR	MOV	
8		ADD XOR	MOV	
9		ADD	XOR	MOV

Такт	Fetch	Dec	Exec	Retire
1	DIV			
2	MOV	DIV		
3	XOR	MOV	DIV	
4	ADD	XOR	DIV	
5		ADD		DIV XOR
6			MOV ADD	
7			MOV	ADD
8			MOV	
9				MOV

Ложные зависимости

В некоторых случаях WaW/WaR-зависимость, существующая с точки зрения ЦП, в реальности не влияет на исход работы программы.

Причиной ложных зависимостей является работа с частью регистра без перезаписи всего регистра (например, выполнение скалярных операций SSE без предшествующего PXOR или работа с AL/AX без перезаписи RAX) - ЦП не может понять, что данные в старшей части регистра не нужны, и разорвать зависимость.

На старых ЦП неполная перезапись арифметических флагов в FLAGS порождает неявную ложную зависимость.

```
MOV EDX, EAX
MOV AL, CL    ; RAX перезаписан частично
ADD AL, 3
```

```
MOV EDX, EAX
MOVZX EAX, CL ; RAX перезаписан полностью
ADD AL, 3
```

```
ADD EDX, 1
INC ECX    ; FLAGS перезаписан частично
```

```
ADD EDX, 1
ADD EXC, 1 ; FLAGS перезаписан полностью
```

```
MOVPS XMM1,XMM0
MOVSS XMM0,XMM2;XMM0 перезаписан частично
```

```
MOVPS XMM1,XMM0
XORPS XMM0,XMM0;XMM0 перезаписан полностью
MOVSS XMM0,XMM2
```

Переупорядочивание операций чтения/записи

Переупорядочивание зависимых операций чтения/записи (т.е. операций по перекрывающимся адресам), очевидно, невозможно.

Переупорядочивание независимых операций чтения/записи возможно. Правила подобных переупорядочиваний прописываются в ISA.

На x86-64:

- операции чтения не могут переупорядочиваться друг с другом;
- операции записи не могут переупорядочиваться друг с другом;
- операция чтения может быть переупорядочена с более ранней операцией записи, но не наоборот.

На ARM все переупорядочивания возможны => возможна ситуация, когда многопоточная программа без использования дополнительных мер корректно работает на x86-64, но работает с ошибками на ARM.

Барьеры памяти

В некоторых случаях (многопоточное/системное программирование) переупорядочивание операций с памятью является нежелательным. Если необходимо, чтобы операции чтения/записи были выполнены в определенном порядке, используются **барьеры памяти** (memory barriers).

Барьер памяти гарантирует, что операции определенного типа, находящиеся до барьера, полностью завершатся до барьера, а расположенные после барьера – начнут выполнение после барьера.

Часто барьеры описываются комбинацией слов Load и Store. Например, LoadStore-барьер гарантирует, что операции чтения до барьера выполнятся до барьера, а операции записи после барьера начнутся после барьера.

На x86-64 барьеры памяти реализуются с помощью 3 инструкций:

- `LFENCE` = LoadStore + LoadLoad барьер*;
- `SFENCE` = StoreStore барьер**;
- `MFENCE` = полный барьер***.

В силу строгой модели памяти x86 барьеры, памяти используются редко. На ARM/RISC-V барьеры используются очень широко.

* дополнительно запрещает переупорядочивание вообще любых микроопераций относительно себя
** не имеет смысла для обычных операций чтения (см. MOVNTQ) *** сериализует только операции чтения/записи, в отличие от LFENCE

Зависимость по управлению. Предсказание переходов (пример)

Условные переходы являются проблемой для архитектур с конвейером, т.к. от результата условного перехода зависит, какие инструкции будут считываться/декодироваться дальше – возникает **зависимость по управлению**.

В этом случае, применима «простая» оптимизация – процессор может сохранить текущее состояние, предположить исход условного перехода и продолжить выполнение одной из веток (**спекулятивное выполнение**). В момент, когда результат перехода вычислится окончательно, проверка проводится повторно. Если результат проверки не совпал с предположением, то процессор восстанавливает сохраненное состояние, «откатывая» те изменения, которые он уже сделал.

Отсюда следует, что производительность ЦП зависит от того, насколько предсказуемы данные и насколько хорошо ЦП умеет определять закономерности.

При спекулятивном выполнении ЦП не может выполнять действия, которые влияют на состояние ЦП и программы в целом. В частности, результаты спекулятивных операций записи сохраняются в очередь записи, но не отправляются в ОЗУ, пока операция не будет признана действительной.

Если результатом спекулятивной операции было аппаратное исключение (например, из-за выхода за границы массива/деления на 0), то оно задерживается, пока выполнение исходной инструкции не будет подтверждено

Буферизация записи

Поскольку ОЗУ является общим ресурсом, ЦП не имеет права проводить спекулятивную запись в нее.

Кроме того, даже не спекулятивная запись производится долго. С другой стороны, запись производится реже, чем чтение и прочие операции.

Логичной оптимизации является создание **очереди записи** (write-back queue), в которую отправляются значения, которые должны быть записаны в ОЗУ. Данные из этой очереди считываются контроллером памяти.

Очередь записи может быть источником данных для операций чтения.

Сериализация микроопераций

В ряде случаев переупорядочивание микроопераций и спекулятивное выполнение нежелательны и должны быть запрещены.

Некоторые инструкции (`LFENCE` на ЦП Intel и AMD, `MFENCE` на ЦП AMD, `CPUID`) запрещают переупорядочивание микроопераций относительно себя и при этом дожидаются выполнения всех предыдущих инструкций.

Данные инструкции используются в основном в системном программировании для обеспечения корректного порядка выполнения действий.

Аппаратные исключения и прерывания (см. следующую лекцию) также запрещают переупорядочивание инструкций. *Т.к. аппаратные исключения используются при отладке, программисту не видны эффекты внеочередного выполнения.*

Многоядерные процессоры

Широкодоступные 2-ядерные процессоры x86-64 появились в мае 2004 г. В настоящее время число ядер в ЦП варьируется от 2 до 24, а потребительские ЦП с 1 ядром не выпускаются вовсе.

Каждое ядро считывает и выполняет инструкции независимо от других ядер. Общими для ядер обычно являются L3-кэш, контроллер памяти и контроллер ввода вывода.

Обычно ядра общаются между собой по специальной шине и информируют друг друга о действиях, которые могут затронуть другие ядра.

Наличие полностью независимых ядер позволяет выполнять несколько программ (или несколько участков кода одной программы) одновременно. С другой стороны, из независимости возникают проблемы синхронизации.

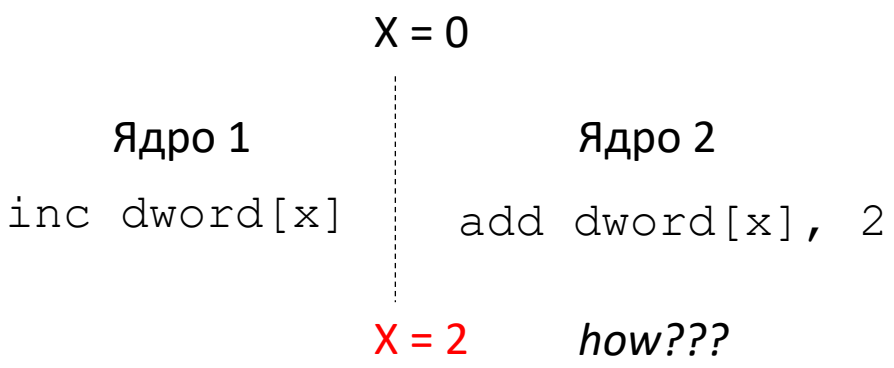
Атомарные операции

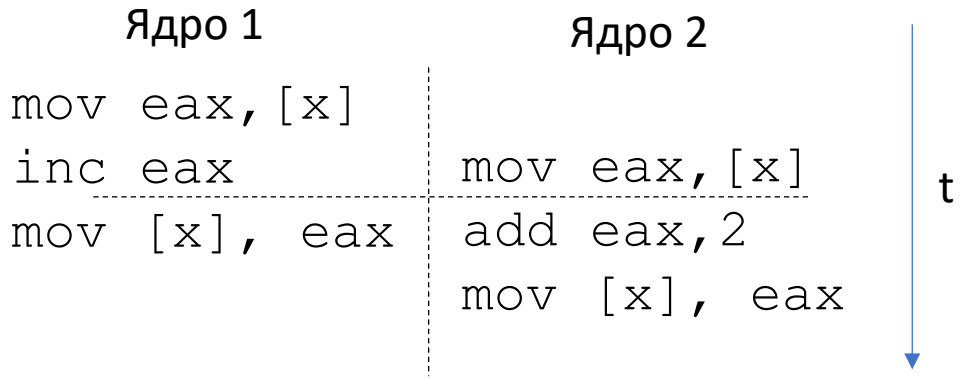
Инструкции при декодировании могут разбиваться на несколько микроопераций, которые выполняются отдельно.

Данный факт может влиять на корректность многопоточных программ, поскольку операции не являются атомарными.

Операция является **атомарной**, если сторонний наблюдатель (в т.ч. другое ядро) не может получить доступа к данным, над которыми операция выполняется в данный момент (т.е. начата, но не завершена).

Операции чтения и записи выровненных значений размером не более 8 байт всегда атомарны в x86-64.



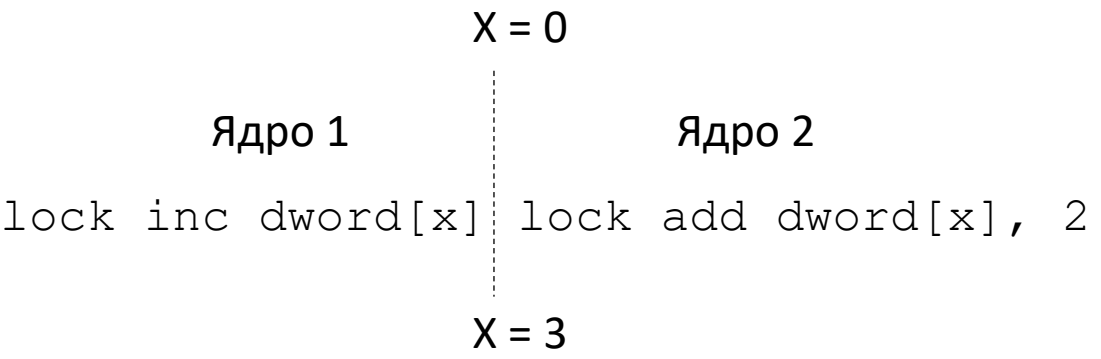


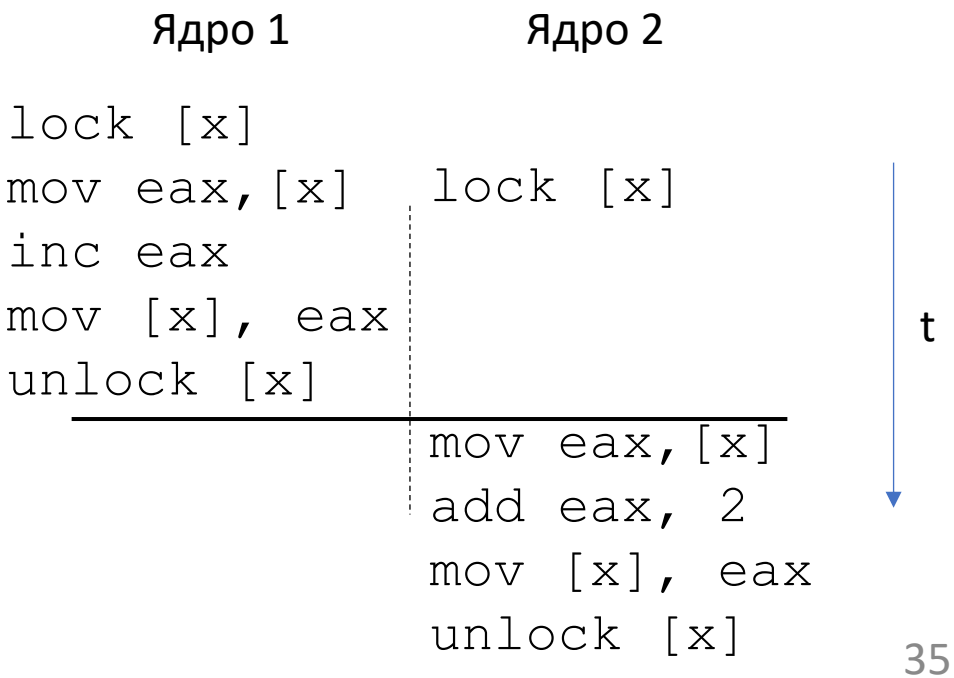
Префикс LOCK (пример)

Для выполнения атомарных операций используется префикс `LOCK`. Данный префикс может быть использован не со всеми инструкциями*.

При использовании данного префикса область памяти, над которой выполняется операция, блокируется на время ее выполнения.

Операции с префиксом `LOCK` являются неявными барьерами памяти – все операции чтения-записи не могут быть переупорядочены через `LOCK`.





* ADD, ADC, AND, BTC, BTR, BTS, CMPXCHG, CMPXCH8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XOR, XADD, XCHG

Кэширование в многоядерных системах

Поскольку обычно каждое ядро ЦП имеет свои кэши L1/L2, возникает проблема согласованности данных разных ядер.

Наиболее простой вариант – использование **некогерентного кэша**. В этом случае содержимое кэшей не обязано быть синхронизированным => ядра могут видеть одну и ту же область памяти по-разному. В этом случае для синхронизации состояния используются спец. инструкции.

В случае **когерентного кэша** ядра ЦП обмениваются информацией об операциях чтения/записи.

В самом простом варианте, если одно из ядер производит запись в память, соответствующая кэш-линия в кэшах других ядер помечается, как невалидная – ядра запросят ее из ОЗУ повторно. Никаких спец. инструкций синхронизации состояния не требуется.

В более сложных схемах управления кэшем ядра могут обмениваться кэш-линиями, избегая лишних запросов в кэш нижнего уровня/ОЗУ.

Все ЦП x86 используют когерентные кэши.

См. также: https://en.wikipedia.org/wiki/MESI_protocol

False sharing (пример)

Если 2 *независимые* переменные, используемые несколькими ядрами, располагаются на одной кэш-линии, то производительность работы с ними существенно падает:

- запись в любую из переменных инвалидирует кэш-линию в других ядрах => они будут запрашивать кэш-линию заново;
- префикс LOCK обычно приводит к блокировке *всей* кэш-линии, содержащей переменную, что исключает доступ к ней со стороны других ядер даже для операций чтения.

Если переменные располагаются на разных кэш-линиях, такого не произойдет. С этой целью независимые переменные или группы переменных, участвующие в многопоточном взаимодействии, выравниваются по значению, превышающему размер кэш-линии (см. std::hardware_destructive_interference_size).

Simultaneous Multithreading

Несмотря на наличие спекулятивного выполнения, полностью загрузить исполнительные устройства одного ядра проблематично => часто в ядре есть свободные вычислительные ресурсы.

Эти ресурсы можно использовать, если загрузить ядро инструкциями не из 1, а из нескольких (обычно 2) потоков выполнения (одновременная многопоточность, Simultaneous Multithreading, Hyperthreading).

В этом случае ЦП должен считывать и выполнять инструкции из разных потоков с соблюдением их изоляции. Достигается это частичным дублированием управляющих блоков ЦП.

SMT позволяет поднять производительность системы в целом за счет повышения количества доступных «ядер». Оборотной стороной является повышение конкуренции за общие ресурсы ЦП (кэши, шина ОЗУ, ввод/вывод) и исполнительные устройства ядра, что в некоторых случаях приводит к понижению производительности некоторых приложений