

# Лекция 3

## Оптимизация доступа к памяти (memory access optimization)

**Курносков Михаил Георгиевич**

E-mail: [mkurnosov@gmail.com](mailto:mkurnosov@gmail.com)

WWW: [www.mkurnosov.net](http://www.mkurnosov.net)

Курс «Высокопроизводительные вычислительные системы»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

# Организация подсистемы памяти (история)

---

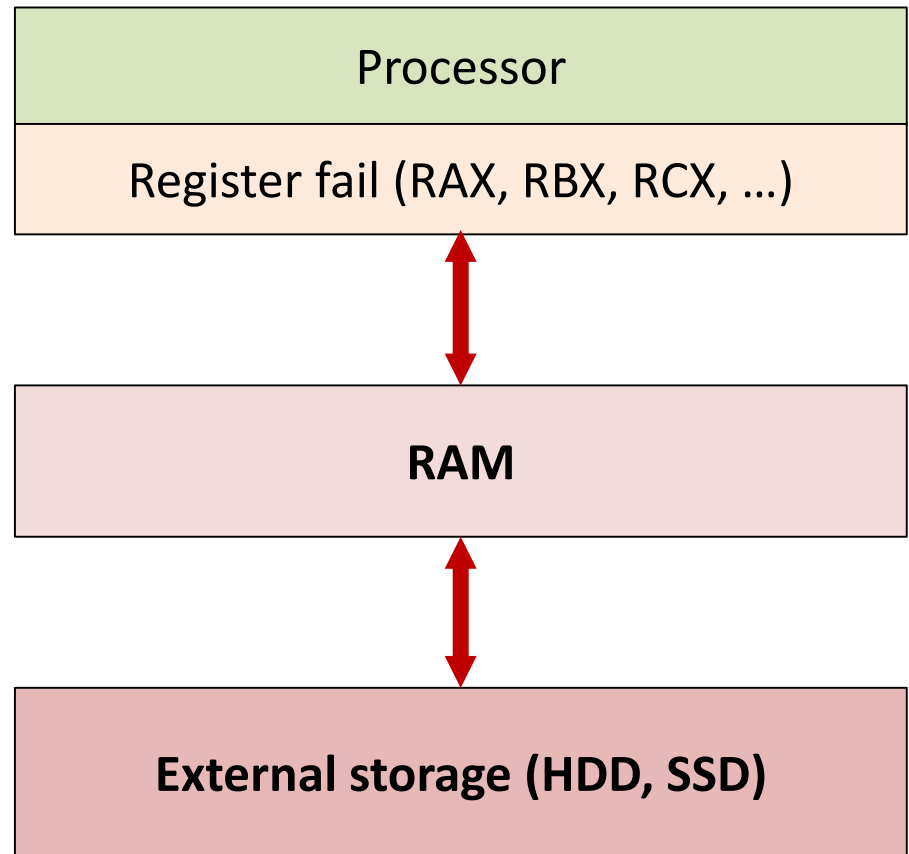
$v[i] = v[i] + 10$

```
addl $0xa, v(,%rax,4)
```

1. Load  $v[i]$  from memory
2. Add 2 operands
3. Write result to memory

Время доступа к памяти  
(load/store) для многих  
программ является  
критически важным

*(memory bound application,  
memory intensive application)*



# Доступ к памяти

```
int a[100];

int sum = 0;
for (int i = 0; i < 100; i++) {
    sum += a[i];
}
```

```
$ gcc -o prog ./prog.c --save-temps
```

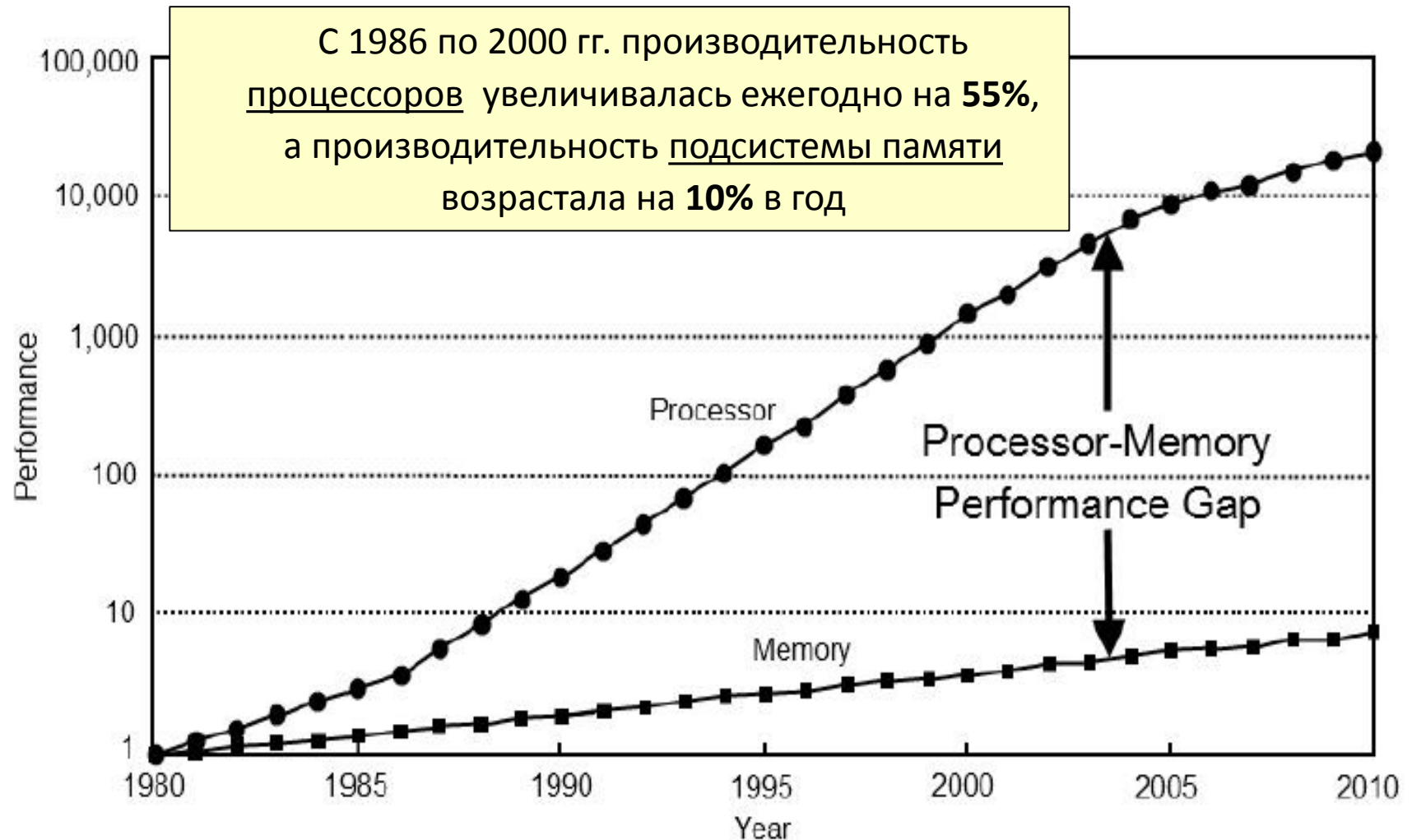


```
        movl    $0, -4(%rbp)           // sum = 0
        movl    $0, -8(%rbp)           // i = 0
        jmp     .L2

.L3:
        movl    -8(%rbp), %eax
        cltq
        movl    -416(%rbp,%rax,4), %eax // Convert Long To Quad
        addl    %eax, -4(%rbp)           // a[i] -> %eax
        addl    %eax, -4(%rbp)           // sum = sum + %eax
        addl    $1, -8(%rbp)            // i++

.L2:
        cmpl    $99, -8(%rbp)
        jle     .L3
```

# Стена памяти (memory wall)



[1] Hennessy J.L., Patterson D.A. **Computer Architecture: A Quantitative Approach (5th ed.)**. – Morgan Kaufmann, 2011. – 856 p.

# Латентность памяти (memory latency)

---

	<b>VAX-11/750</b> 1980 г.	<b>Modern CPU</b> 2004 г.	<b>Improvement</b> since 1980
<b>Clock speed</b> (MHz)	6	3000	<b>+500x</b>
<b>Memory size</b> (MiB)	2	2000	<b>+1000x</b>
<b>Memory bandwidth</b> (MiB/s)	13	7000 (read) 2000 (write)	<b>+540x</b> <b>+150x</b>
<b>Memory latency</b> (ns)	225	70	<b>+3x</b>
<b>Memory latency</b> (cycles)	1.4	210	<b>-150x</b>

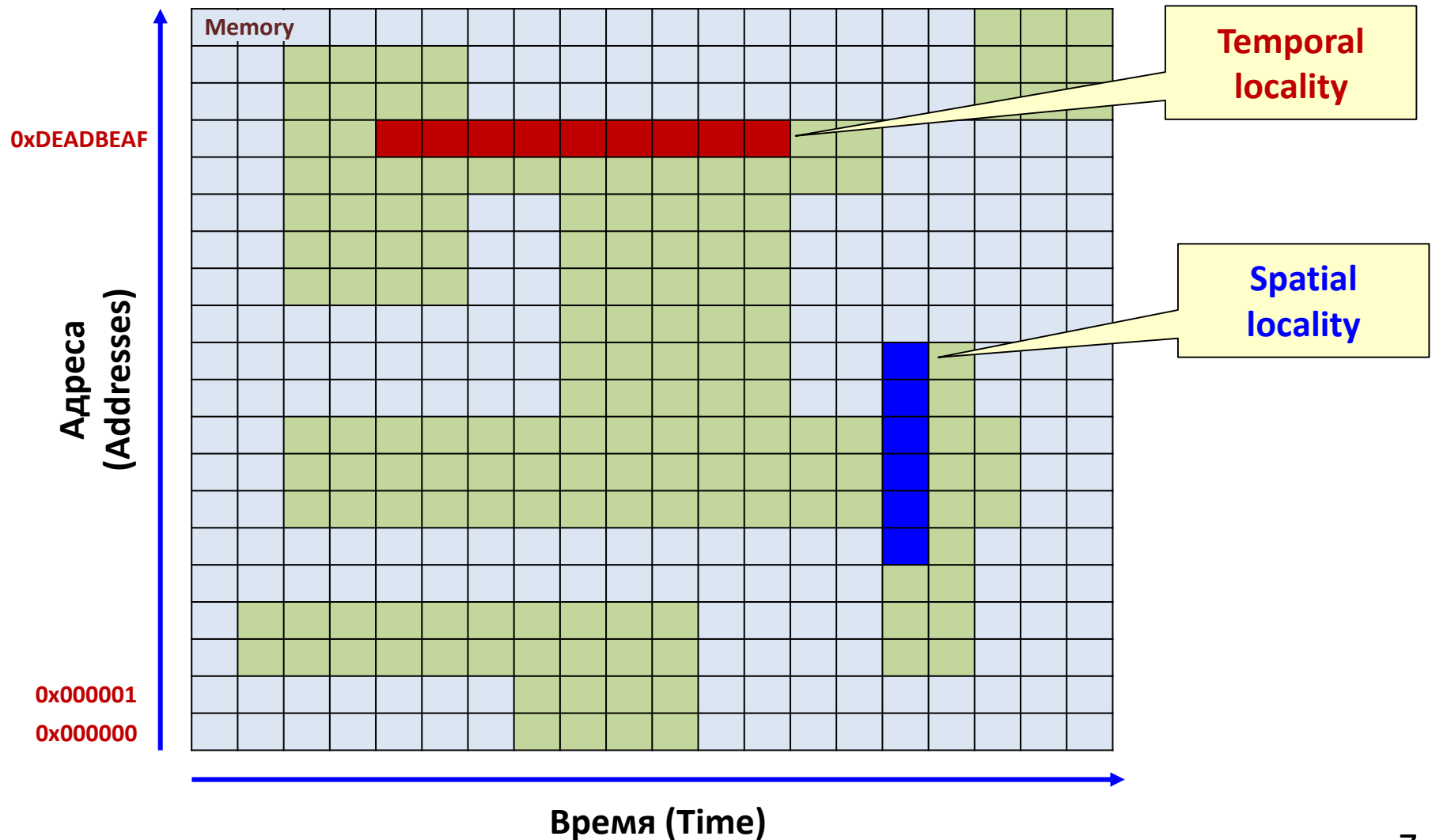
Herb Sutter. **Machine Architecture: Things Your Programming Language Never Told You** // [http://www.nwcpp.org/Downloads/2007/Machine\\_Architecture\\_-\\_NWCPP.pdf](http://www.nwcpp.org/Downloads/2007/Machine_Architecture_-_NWCPP.pdf)

# Локальность ссылок

---

- **Локальность ссылок (locality of reference)** – свойство программ повторно (часто) обращаться к одним и тем же адресам в памяти (данным, инструкциям)
  
- **Формы локальности ссылок:**
  - ❑ **Временная локализация (temporal locality)** – повторное обращение к одному и тому же адресу через короткий промежуток времени (например, в цикле)
  
  - ❑ **Пространственная локализация ссылок (spatial locality)** – свойство программ повторно обращаться через короткий промежуток времени к адресам близко расположенным в памяти друг к другу

# Локальность ссылок



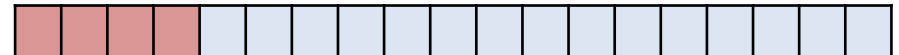
# Локальность ссылок

## Структура (шаблон) доступа к массиву (reference pattern)

```
int sumvec1(int v[N])
{
    int i, sum = 0;
    for (i = 0; i < N; i++)
        sum += v[i];
    return sum;
}
```

Address	0	4	8	12	16	20
Value	v[0]	v[1]	v[2]	v[3]	v[4]	v[5]
Step	1	2	3	4	5	6

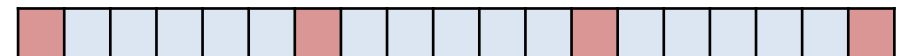
*stride-1 reference pattern (good locality)*



```
int sumvec2(int v[N])
{
    int i, sum = 0;
    for (i = 0; i < N; i += 6)
        sum += v[i];
    return sum;
}
```

Address	0	24	48	72	96	120
Value	v[0]	v[6]	v[12]	v[18]	v[24]	v[30]
Step	1	2	3	4	5	6

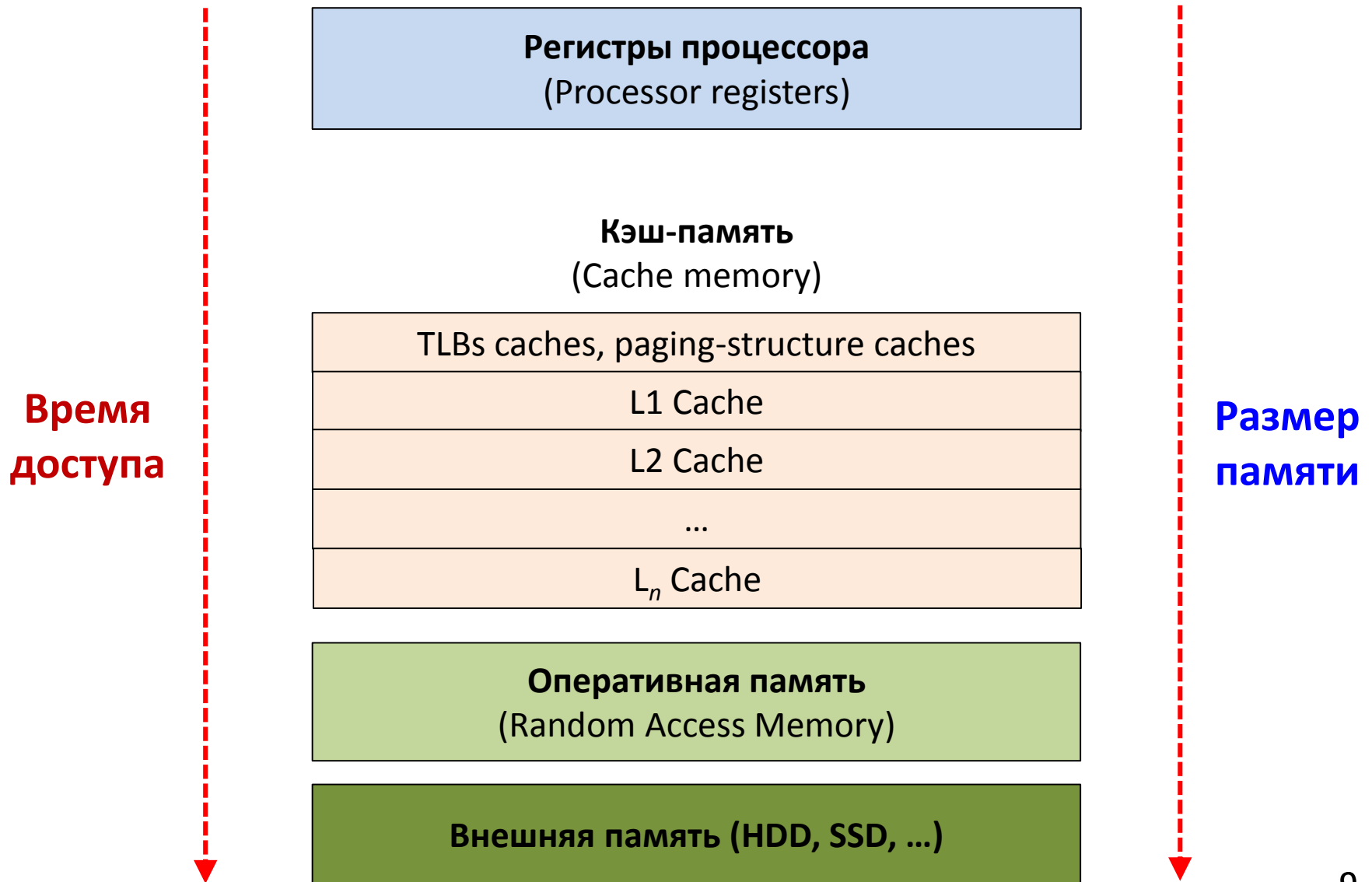
*stride-6 reference pattern*





# Иерархическая организация памяти

---



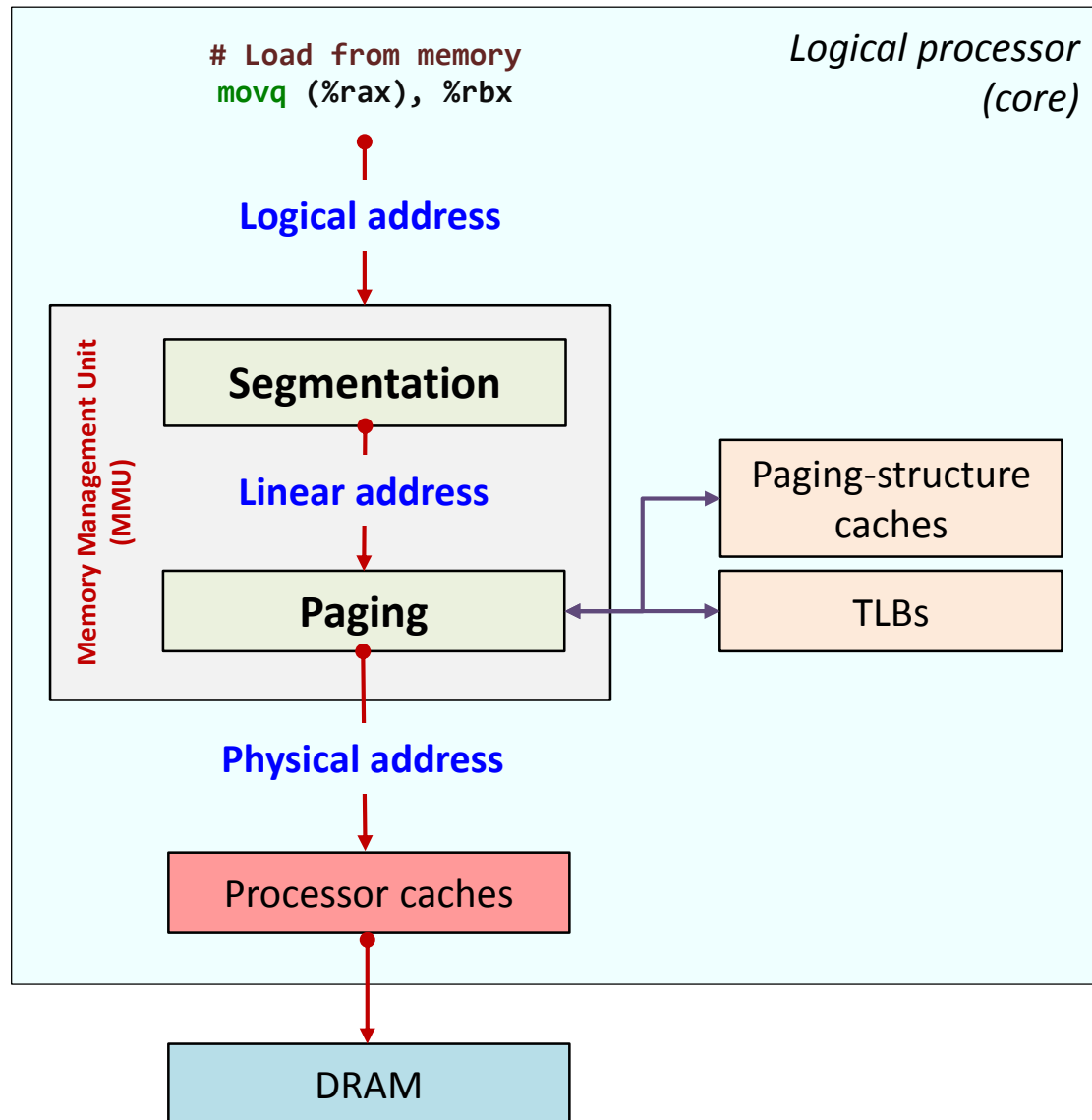
# Стена памяти (memory wall)

---

## Как минимизировать латентность (задержку) доступа к памяти?

- **Внеочередное выполнение** (out-of-order execution) – динамическая выдача инструкций на выполнение по готовности их данных
- **Вычислительный конвейер** (pipeline) – совмещение (overlap) во времени выполнения инструкций
- **Суперскалярное выполнение** (superscalar) – выдача и выполнение нескольких инструкций за такт ( $CPI < 1$ )
- **Одновременная многопоточность** (simultaneous multithreading, hyper-threading)

# Страничная организация памяти

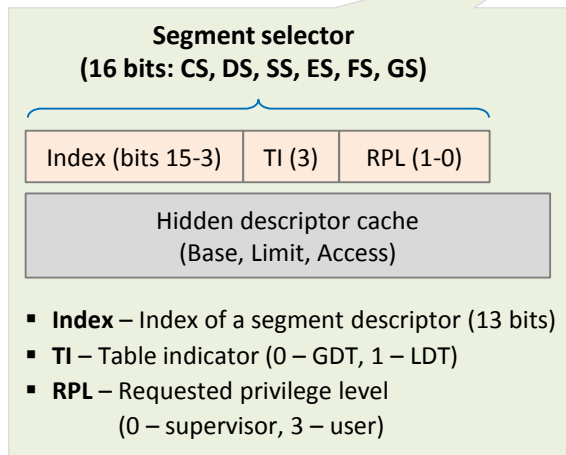


Vol. 3A Intel 64 and IA-32 Architectures Software Developer's Manual  
Chapter 3 Protected Mode Memory Management (Intel 64, IA-32e mode)

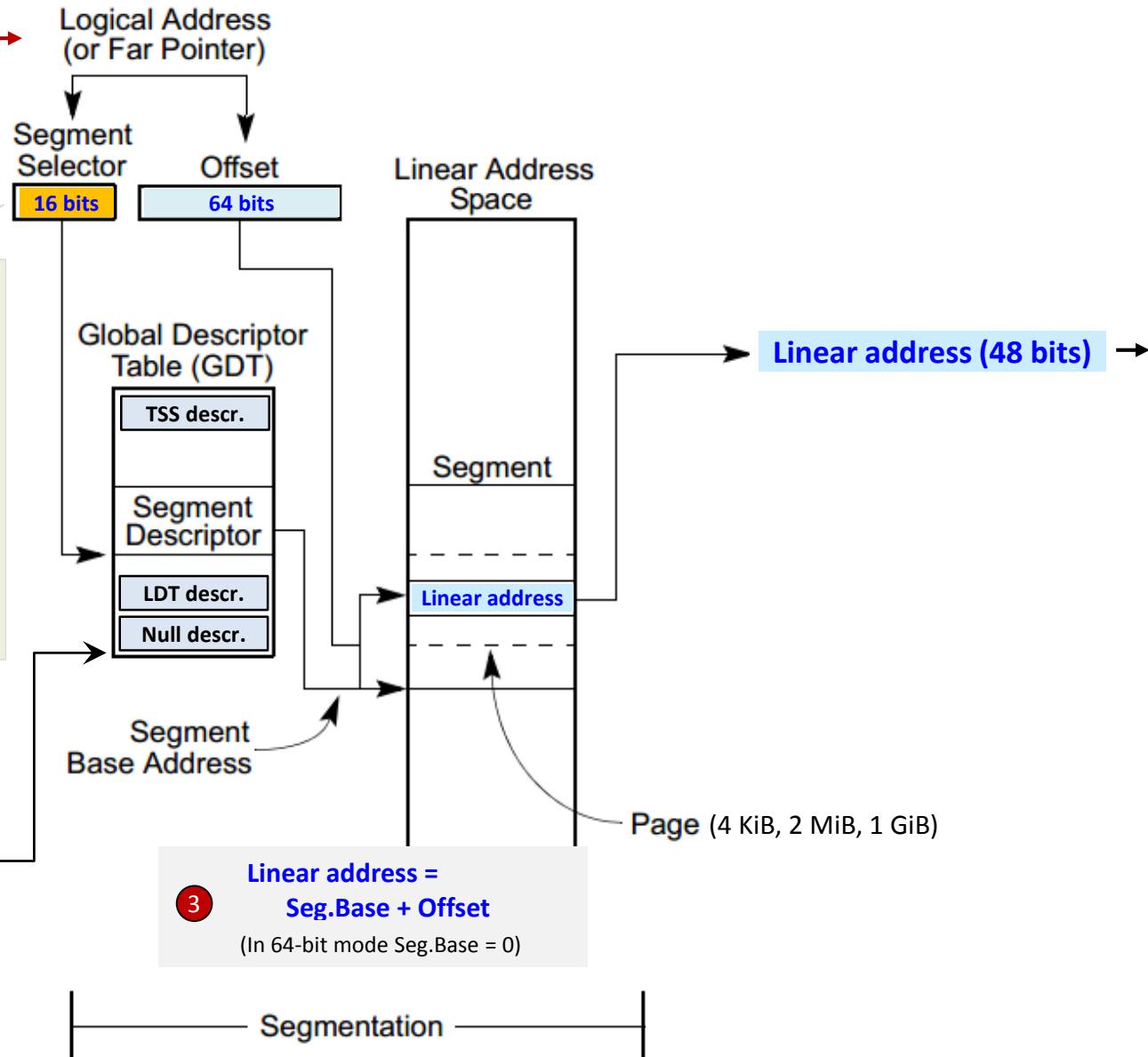
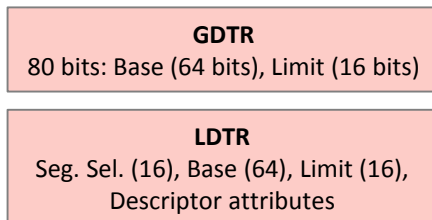
Logical address (SegSel : Offset) --> Linear address (48 bits)

# Load from memory  
`movq (%rax), %rbx`

- 1 Select descriptor table  
(TI = 0 ? GTD : LDT)



- 2 Read segment descriptor  
Check rights access and limits

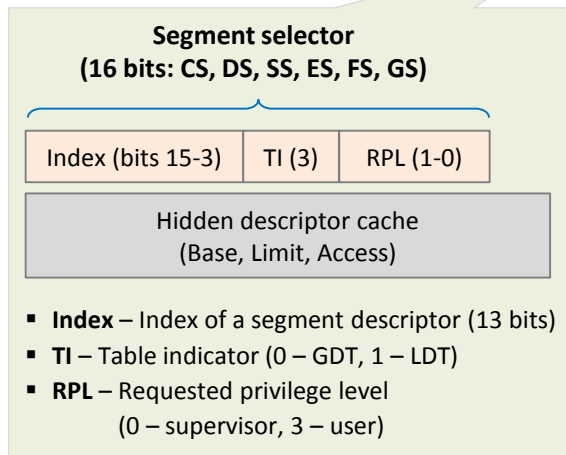


Vol. 3A Intel 64 and IA-32 Architectures Software Developer's Manual  
Chapter 3 Protected Mode Memory Management (Intel 64, IA-32e mode)

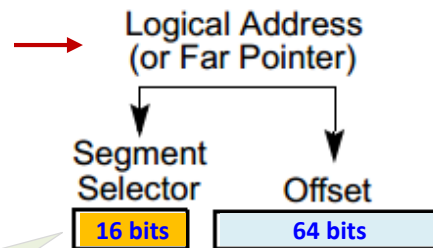
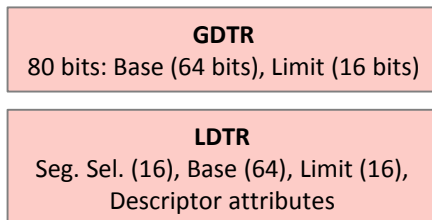
Logical address (SegSel : Offset) --> Linear address (48 bits)

# Load from memory  
`movq (%rax), %rbx`

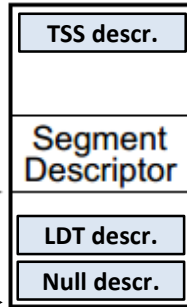
- 1 Select descriptor table  
(TI = 0 ? GTD : LDT)



- 2 Read segment descriptor  
Check rights access and limits



Global Descriptor Table (GDT)



Segment Base Address

3 Linear address =  
Seg.Base + Offset  
(In 64-bit mode Seg.Base = 0)

▪ **Average/best case**

Поля дескриптора сегмента читаются из скрытого регистра (Seg.Base, ...)

▪ **Worst case**

Дескриптор сегмента загружается из оперативной памяти в скрытый регистр (требуется трансляция адреса)

▪ **В 64-битном режиме сегментация не используется (Seg.Base = 0)**

▪ Таблицы дескрипторов сегментов (GDT и LDT) хранятся в оперативной памяти

▪ Трансляция адреса GDTR/LDTR осуществляется при установке сегментного регистра и загрузке соответствующего ему дескриптора в скрытый регистр

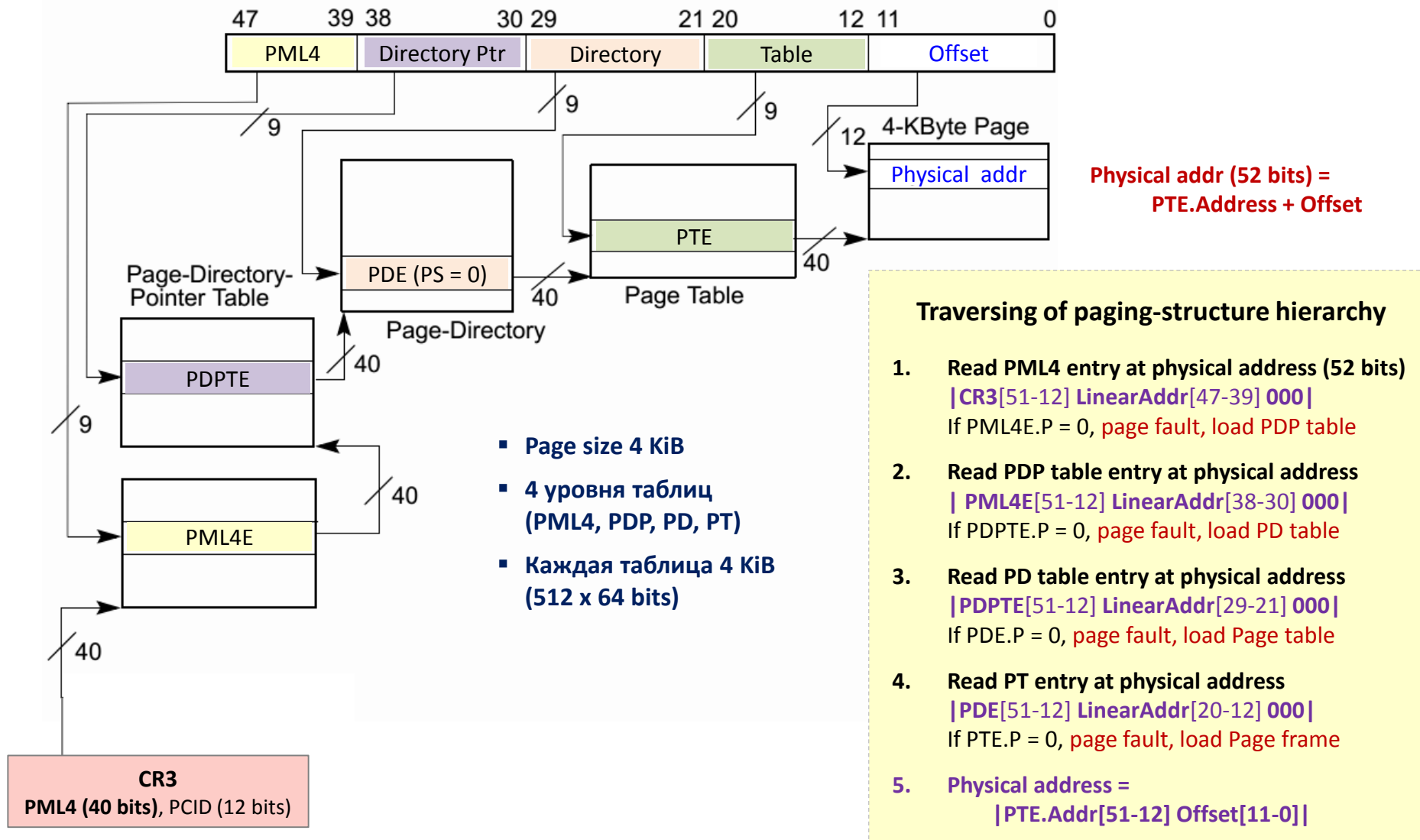
Segmentation

# Vol. 3A Intel 64 and IA-32 Architectures Software Developer's Manual

## Chapter 4 Paging (Intel 64, IA-32e mode)

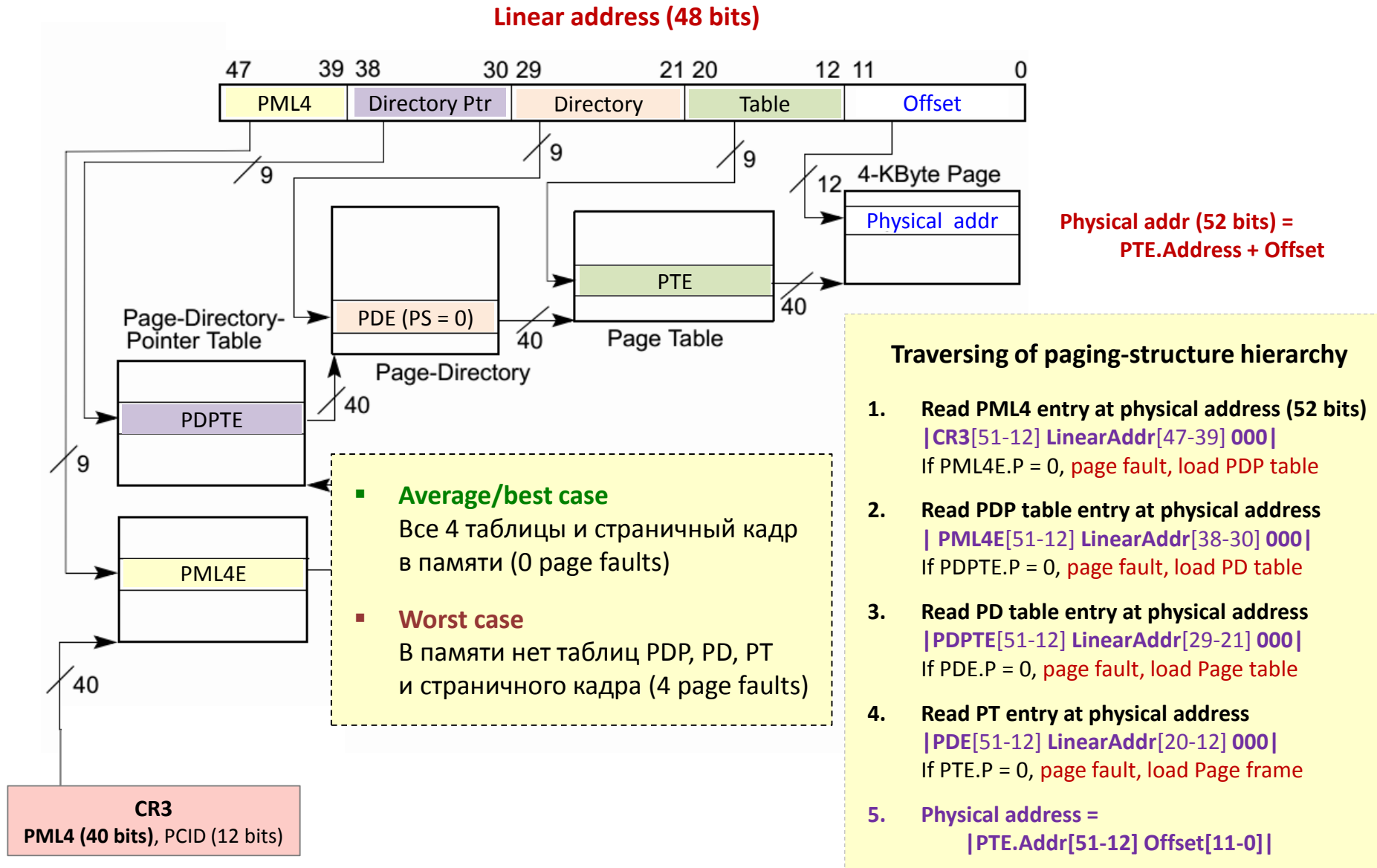
Linear address (48 bits) --> Physical address (52 bits)

Linear address (48 bits)



Vol. 3A Intel 64 and IA-32 Architectures Software Developer's Manual  
Chapter 4 Paging (Intel 64, IA-32e mode)

Linear address (48 bits) --> Physical address (52 bits)



## 4.10 Caching translation information (Intel 64, IA-32e mode)

Linear address (48 bits)



TLB (ITLB/DTLB, levels 1, 2, ...)

Page number (bits 47-12)	PCID (12 bits)	Physical address (40 bits)	Access rights (R/W, U/S, ...)	Attributes (Dirty, memory type)

Linear address  
(48 bits)

PML4 cache

PML4 index (bits 47-39)	Physical address (PML4E.Addr, 40 bits)	Flags (R/W, U/S, ...)

PDPTE cache

PML4 ind.   PDPT ind.   (bits 47-30)	Physical address (PDPTE.Addr, 40 bits)	Flags (R/W, U/S, ...)

PDE cache

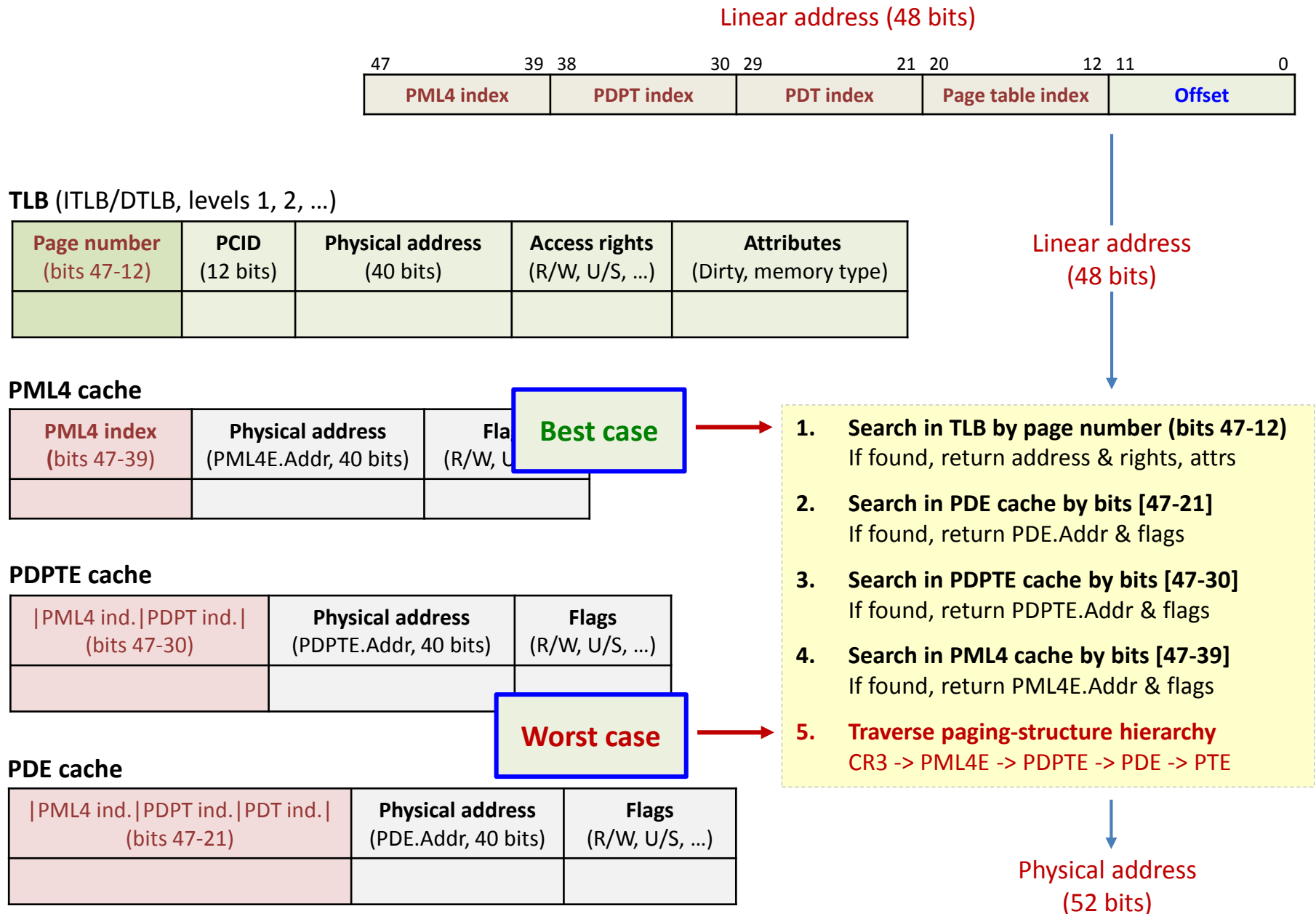
PML4 ind.   PDPT ind.   PDT ind.   (bits 47-21)	Physical address (PDE.Addr, 40 bits)	Flags (R/W, U/S, ...)

- Search in TLB by page number (bits 47-12)**  
If found, return address & rights, attrs
- Search in PDE cache by bits [47-21]**  
If found, return PDE.Addr & flags
- Search in PDPTE cache by bits [47-30]**  
If found, return PDPTE.Addr & flags
- Search in PML4 cache by bits [47-39]**  
If found, return PML4E.Addr & flags
- Traverse paging-structure hierarchy**  
CR3 -> PML4E -> PDPTE -> PDE -> PTE

Physical address  
(52 bits)



## 4.10 Caching translation information (Intel 64, IA-32e mode)



# Invalidation of TLBs and Paging-Structure Caches

---

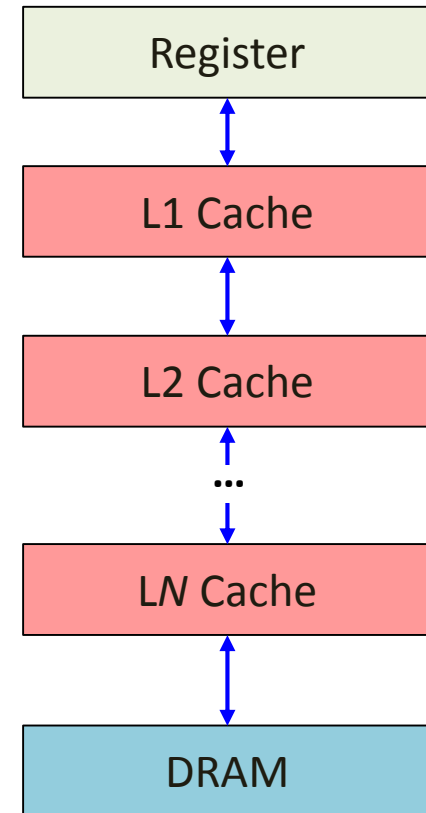
- Процессор создает записи в TLB и paging-structure caches (PSC) при трансляции линейных адресов
- Записи из TLB и PSC могут использоваться даже, если таблицы PML4T/PDPT/PDT/PT изменились в памяти
- Операционная система должна делать недействительными (invalidate) записи в TLB и PSC, информация о которых изменилась в таблицах PML4T/PDPT/PDT/PT
- Инструкции аннулирования записей в TLB и PCS:
  - **INVLPG** LinearAddress
  - INVPCID
  - MOV to CR0 – invalidates all TLB & PCS entries
  - MOV to CR3 – invalidates all TLB & PCS entries (if CR4.PCIDE = 0)
  - MOV to CR4
  - Task switch
  - VMX transitions

# Поиск записи в кеш-памяти процессора

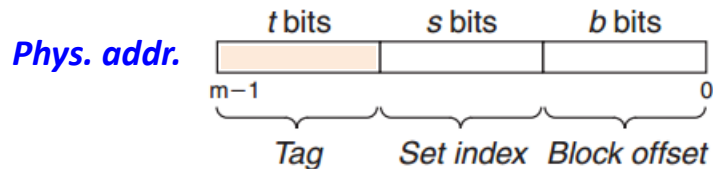
Physical address



- **Поиск записи в L1:** нашли запись (L1 cache hit) – возвращаем данные
- **[L1 cache miss]** L1 кеш обращается в L2 и замещает полученной строкой одну из своих записей (replacement), данные передаются ниже
- **Поиск записи в L2:** нашли запись (L2 cache hit) – возвращаем запись в L1
- **[L2 cache miss]** L2 кеш обращается в L3 и замещает полученной строкой одну из своих записей (replacement), запись передается в L1
- ...
- **Поиск записи в LN:** нашли запись (LN cache hit) – возвращаем запись в L{N-1}
- **[L{N-1} cache miss]** LN кеш обращается в DRAM и замещает одну из своих строк (replacement), запись передается в L{N-1}



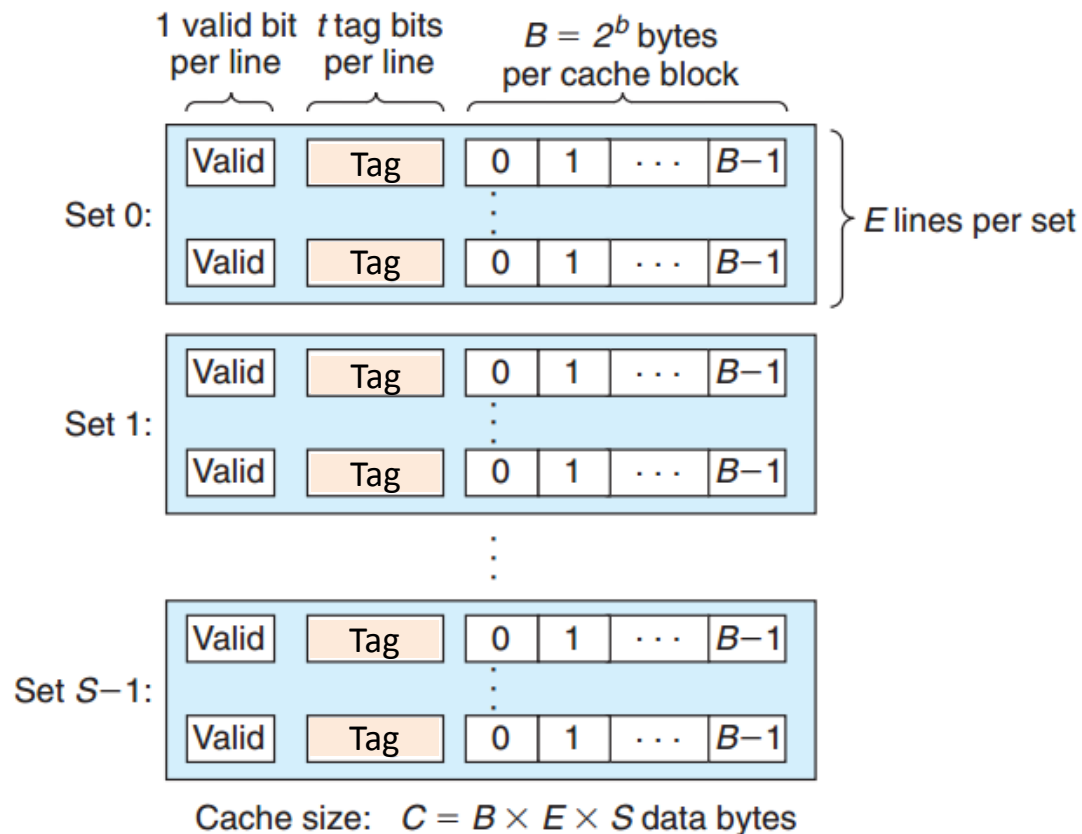
# Структурная организация кеш-памяти



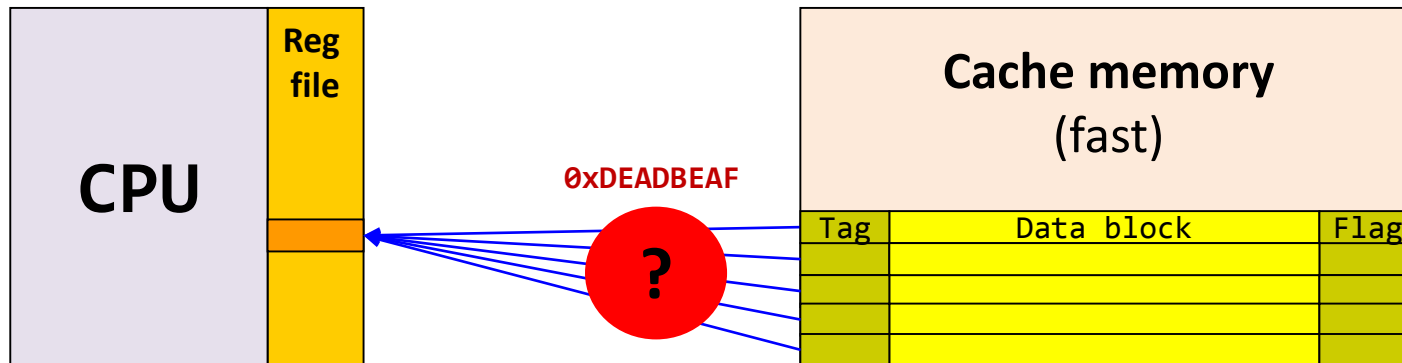
- Кеш содержит  $S = 2^s$  множеств (**sets**)
- Каждое множество содержит  $E$  строк/записей (**cache lines**)
- Каждая строка содержит поля *valid bit*, *tag* ( $t$  бит) и *блок данных* ( $B = 2^b$  байт)
- Данные между кеш-памятью и оперативной памятью передаются блоками по  $B$  байт (cache lines)
- “Полезный” размер кеш-памяти

$$C = S * E * B$$

## Множественно-ассоциативная кеш-память



# Методы отображения адресов (mapping)

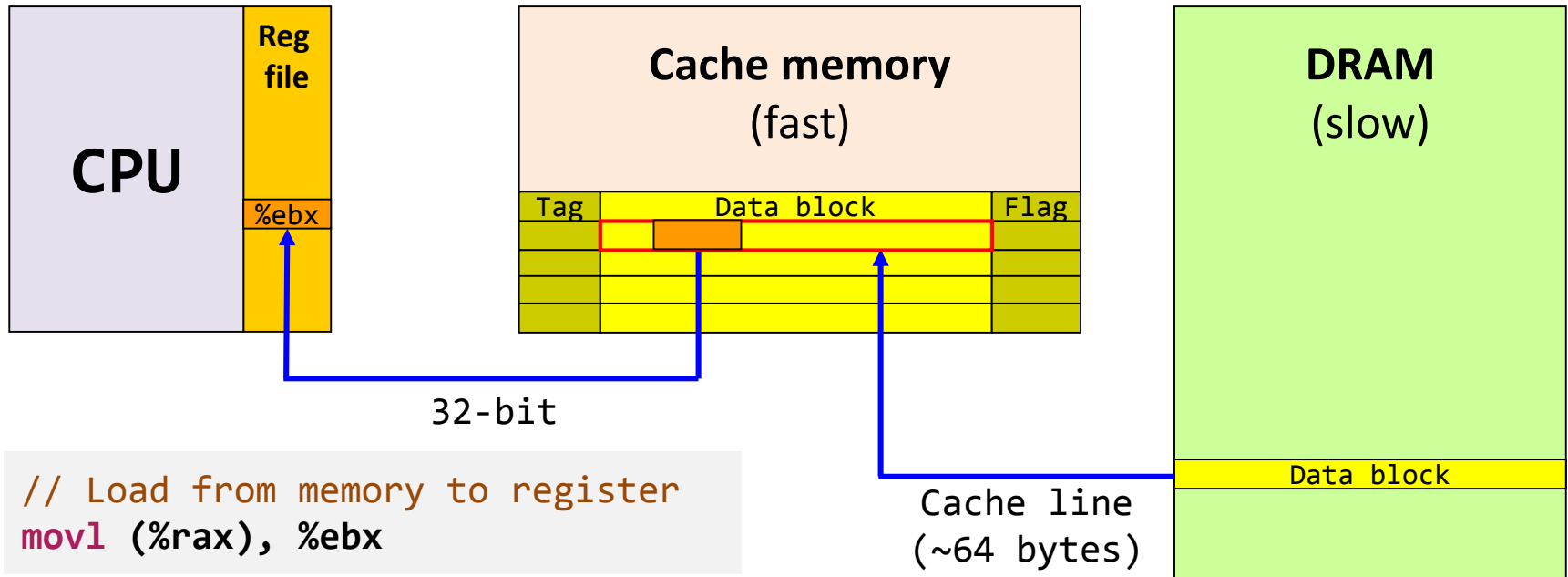


```
// Read: Memory -> register  
movl (0xDEADBEEF), %eax
```

В какой записи кеш-памяти  
размещены данные с адресом  
`0xDEADEAF`?

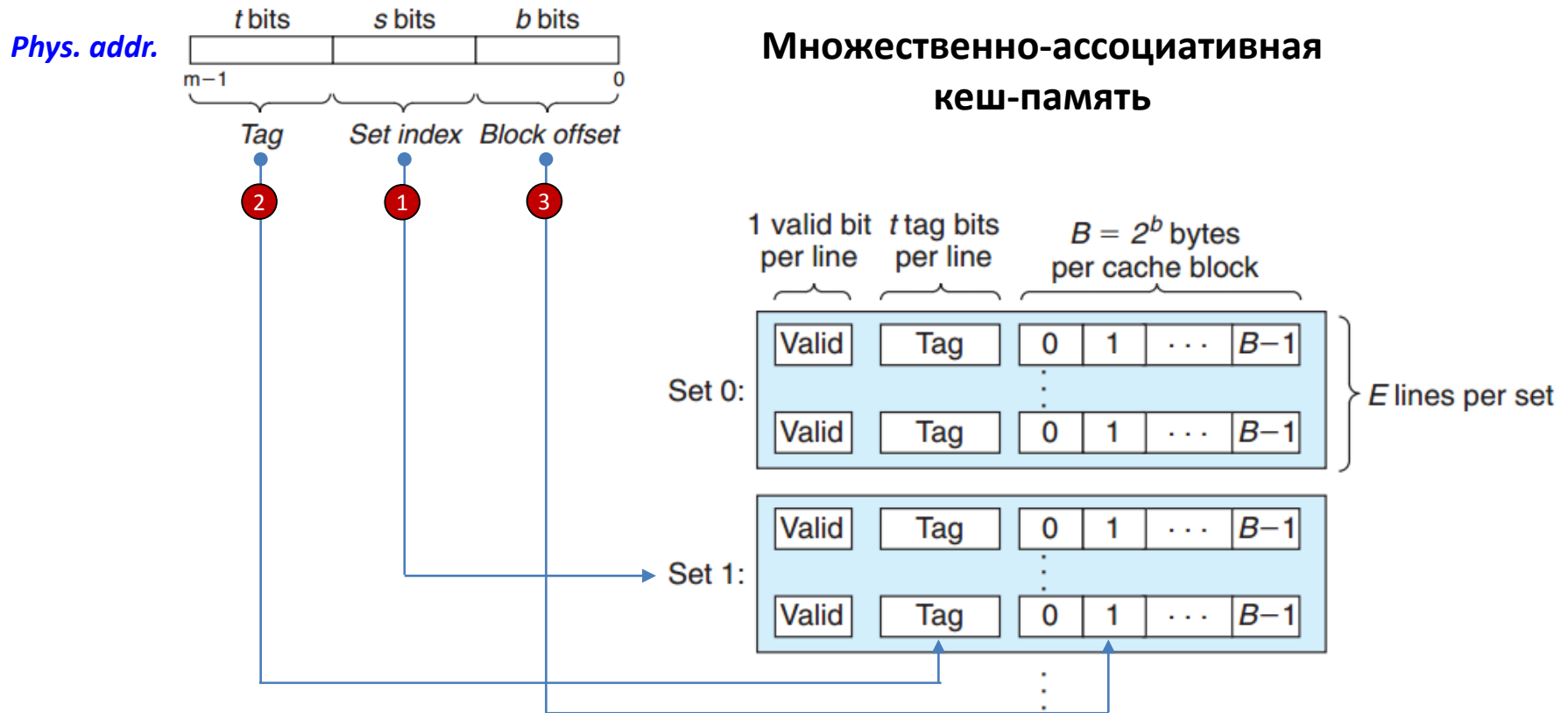
- **Метод отображения адресов (mapping method)** – метод сопоставления физическому адресу записи в кеш-памяти
- **Виды методов отображения (параметры кеш-памяти  $S$ ,  $E$ ,  $B$ ):**
  - ☐ **Множественно-ассоциативное отображение** (set-associative mapping)
  - ☐ **Прямое отображение** (direct mapping) – в каждом множестве (set) по одной записи ( $E = 1$ , поле Tag не требуется, только Set index)
  - ☐ **Полностью ассоциативное отображение** (full associative mapping) – одно множество ( $S = 1$ , поле Set index не требуется, только Tag)

# Загрузка данных из памяти (Load/read)



```
if <Блок с адресом ADDR в кеш-памяти> then
    /* Cache hit */
    <Вернуть значение из кеш-памяти>
else
    /* Cache miss */
    <Загрузить блок данных из кеш-памяти следующего уровня (либо DRAM)>
    <Разместить загруженный блок в одной из строк кеш-памяти>
    <Вернуть значение из кеш-памяти (загруженное)>
end if
```

# Загрузка данных из памяти (Load/read)



1. Выбирается одно из  $S$  множеств (по полю *Set index*)
2. Среди  $E$  записей множества отыскивается строка с требуемым полем *Tag* и установленным битом *Valid* (нашли – *cache hit*, не нашли – *cache miss*)
3. Данные из блока считываются с заданным смещением *Block offset*

# Замещение записей кеш-памяти

## 2-way set associative cache:

Set0	V	Tag	Word0	Word1	Word2	Word3
Set1						
Set2	1	0001	15	20	35	40
	1	0011	1234	1222	3434	896
Set3						

## Memory:

0-3	Word0	Word1	Word2	Word3
4-7				
8-11				
12-15				
16-19				
20-23				
24-27	15	20	35	40
28-31				
32-35				
36-39				
40-43	12	2312	342	7717
44-47				
48-51				
52-55				
56-59	1234	1222	3434	896
60-63				
64-67				
68-71				
...				

## Промах при загрузке данных

```
// Load from memory to register  
movl (40), %eax
```

- В какую строку (way) множества 2 загрузить блок с адресом 40?
- Какую запись вытеснить (evict) из кеш-памяти в DRAM?

Address 40:

Tag: 000010 <sub>2</sub>	Index: 10 <sub>2</sub>	Offset: 00 <sub>2</sub>
--------------------------	------------------------	-------------------------

Address 24:

Tag: 000001 <sub>2</sub>	Index: 10 <sub>2</sub>	Offset: 00 <sub>2</sub>
--------------------------	------------------------	-------------------------

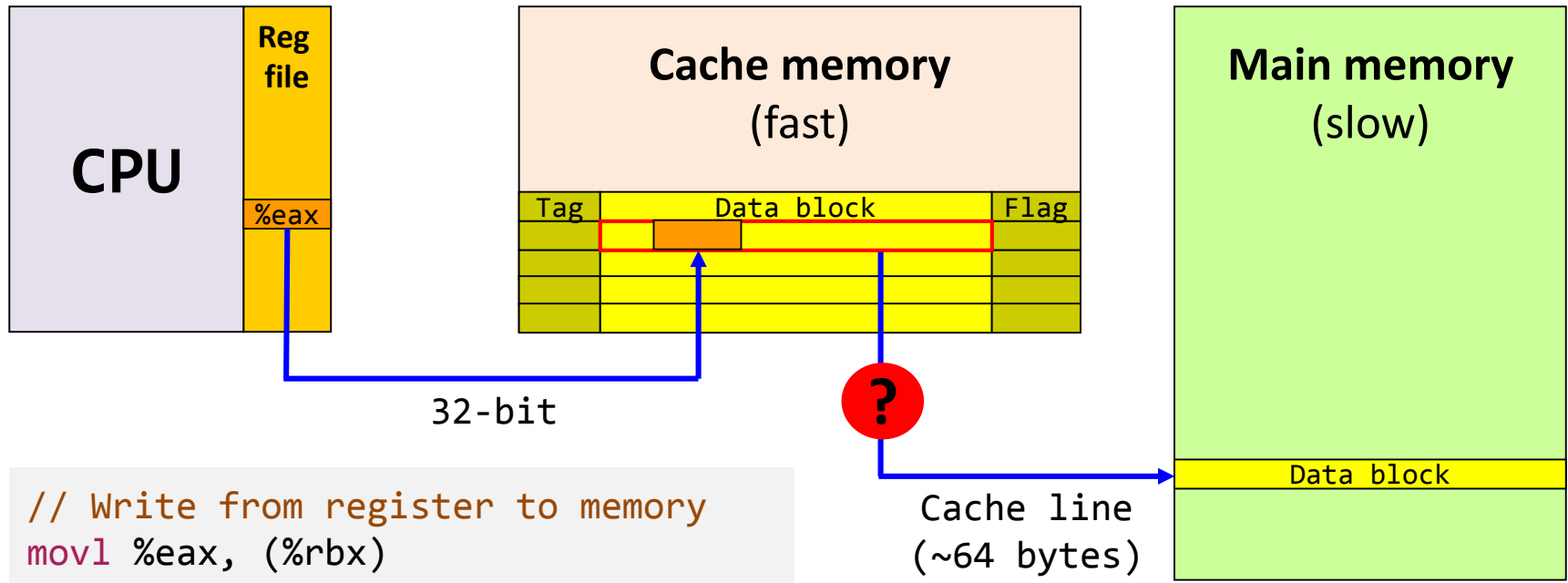


# Алгоритмы замещения записей кеш-памяти

---

- **Алгоритмы замещения** (Replacement policy) требуют хранения вместе с каждой строкой кеш-памяти специализированного поля флагов/истории
- **Алгоритм L. Belady** – вытесняет запись, которая с большой вероятностью не понадобится в будущем
- **LRU** (Least Recently Used) – вытесняется строка неиспользованную дольше всех
- **MRU** (Most Recently Used) – вытесняет последнюю использованную строку
- **RR** (Random Replacement) – вытесняет случайную строку
- ...

# Запись данных в память (store/write)



- **Политики записи (Write policy) определяют:**

- ☐ Когда данные должны быть переданы из кеш-памяти в оперативную память
- ☐ Как должна вести себя кеш-память при событии “write miss” – запись отсутствует в кеш-памяти

# Алгоритмы записи в кеш (write policy)

---

## Политика поведения кеш-памяти в ситуации “write hit” (запись имеется в кеш-памяти)

- **Политика Write-through** (сквозная запись) – каждая запись в кеш-память влечет за собой обновление данных в кеш-памяти и оперативной памяти (кеш “отключается” для записи)
- **Политика Write-back** (отложенная запись, copy-back) – первоначально данные записываются только в кеш-память
- Все измененные строки кеш-памяти помечаются как “грязные” (Dirty)
- Запись в память “грязных” строк осуществляется при их замещении или специальному событию (lazy write)
- **Внимание:** чтение может повлечь за собой запись в память
  - ❑ При чтении возник cache miss, данные загружаются из кеш-памяти верхнего уровня (либо DRAM)
  - ❑ Нашли строку для замещения, если флаг dirty = 1, записываем её данные в память
  - ❑ Записываем в строку новые данные

# Алгоритмы записи в кэш (write policy)

---

Политика поведения кеш-памяти в ситуации  
“write miss” (записи не оказалось в кеш-памяти)

- **Write-Allocate**

1. В кеш-памяти выделяется запись (это может привести к выгрузке в память старой записи)
2. Данные загружаются в выделенную строку (при необходимости запись помечается как “dirty”)

- **No-Write-Allocate (Write around)**

- Данные не записываются в кеш-память, сразу передаются в оперативную память (кеш работает только для операция чтения)

- **Часто используются следующие комбинации:**

- ☐ Write-back + Write-allocate
- ☐ Write-through + No-write-allocate

# Показатели эффективности кеш-памяти

---

- **Cache latency** – время доступа к данным в кеш-памяти (clocks)
- **Cache bandwidth** – количество байт передаваемых за такт между кеш-памятью и вычислительным ядром процессора (byte/clock)
- **Cache hit rate** – отношения числа успешных чтений данных из кеш-памяти (без промахов) к общему количеству обращений к кеш-памяти

$$\text{HitRate} = \frac{N_{\text{CacheHit}}}{N_{\text{Access}}}$$

$$0 \leq \text{HitRate} \leq 1$$

# Пример: 4-way set associative cache

---

- Рассмотрим 4-х каналный (4-way) множественно-ассоциативный L1-кеш размером 32 KiB с длиной строки 64 байт (cache line)
- В какой записи кеш-памяти будут размещены данные для физического адреса длиной 52 бит: **0x0000FFFFAB64**?
- Количество записей в кеш-памяти (Cache lines):  $32 \text{ KiB} / 64 = 512$
- Количество множеств (Sets):  $512 / 4 = 128$
- Каждое множество содержит 4 канала (4-ways, 4 lines per set)
- Поле смещения (Offset):  $\log_2(64) = 6$  бит
- Поле номер множества (Index):  $\log_2(128) = 7$  бит
- Поле Tag:  $52 - 7 - 6 = 39$  бит

Tag: 39 бит	Set Index: 7 бит	Offset: 6 бит
-------------	------------------	---------------

# Пример: 4-way set associative cache

---

Physical Address (52 bits):

Tag (39 bit)	Set Index (7 bit)	Offset (6 bit)
--------------	-------------------	----------------

Cache (4-way set associative):

Set 0		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		Tag2	Cache line (64 bytes)
		Tag3	Cache line (64 bytes)
Set 1		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		Tag2	Cache line (64 bytes)
		Tag3	Cache line (64 bytes)
...			
Set 127		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		Tag2	Cache line (64 bytes)
		Tag3	Cache line (64 bytes)

# Пример: 4-way set associative cache

Address: **0x0000FFFFAB64** = **1111111111111111010101101100100**<sub>2</sub>

Tag: <b>111111111111111101</b>	Set Index: <b>0101101</b> = <b>45</b> <sub>10</sub>	Offset: <b>100100</b> = <b>36</b> <sub>10</sub>
--------------------------------	---	---

Cache (4-way set associative):

...		...	...
<b>Set 45</b>		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		<b>111111111111111101</b>	<b>[Start from byte 36, ...</b>
		Tag3	Cache line (64 bytes)
...		...	...

- Тег **111111111111111101** может находиться в любом из 4 каналов множества 45



# Пример: чтение из памяти

В программе имеется массив `int v[100]`

Обратились к элементу `v[17]`, физический адрес которого:

`0x0000FFFFAB64` = `11111111111111111010101101100100`<sub>2</sub>

Tag: <code>1111111111111111101</code>	Set Index: <code>0101101</code> = <code>45</code> <sub>10</sub>	Offset: <code>100100</code> = <code>36</code> <sub>10</sub>
---------------------------------------	---	---

- В кеш-память будет загружен блок из 64 байт с начальным адресом:  
`0x0000FFFFAB40` = `11111111111111111010101101000000`<sub>2</sub>
- В строке кеш-памяти будут размещены 16 элементов по 4 байта (`int`):  
`v[8]`, `v[9]`, `v[10]`, `v[11]`, ..., `v[17]`, ..., `v[23]`

Cache (4-way set associative):

Set <b>45</b>		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		<code>1111111111111111101</code>	<code>v[8]</code> , <code>v[9]</code> , ..., <code>v[17]</code> , ..., <code>v[23]</code>
		Tag3	Cache line (64 bytes)

# Cache line split access

Что будет если прочитать 4 байта начиная с адреса

Address: **0x0000FFFFAB64** = **111111111111111101010110111110**<sub>2</sub>

Tag: <b>111111111111111101</b>	Set Index: <b>0101101</b> = <b>45</b> <sub>10</sub>	Offset: <b>111110</b> = <b>62</b> <sub>10</sub>
--------------------------------	---	---

Cache (4-way set associative):

...		...	...
<b>Set 45</b>		Tag0	Cache line (64 bytes)
		Tag1	Cache line (64 bytes)
		<b>111111111111111101</b>	<b>[0, 1, ..., 62, 63]</b>
		Tag3	Cache line (64 bytes)
<b>Set 46</b>		...	...
		<b>111111111111111101</b>	<b>[0, 1, ..., 63]</b>

**2 байта находятся в другой строке кеш-памяти  
(set = 0101110<sub>2</sub> = 46, offset = 0)  
Cache line split access (split load)**

# Многоуровневая кеш-память

---

- **Inclusive caches** – данные, присутствующие в кеш-памяти L1, обязательно должны присутствовать в кеш-памяти L2
- **Exclusive caches** – те же самые данные в один момент времени могут располагаться только в одной из кеш-памяти – L1 или L2 (например, AMD Athlon)
- Некоторые процессоры допускают одновременное нахождение данных и в L1 и в L2 (например, Pentium 4)
- Intel Nehalem:  
L1 not inclusive with L2; L1 & L2 inclusive with L3

# Intel Nehalem

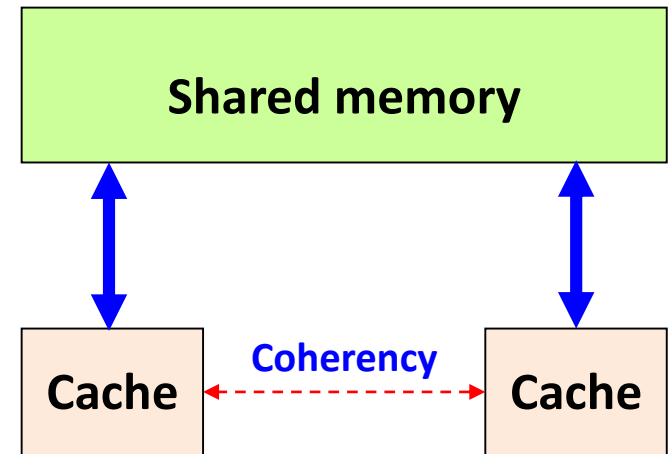
---

- **L1 Instruction Cache:** 32 KiB, 4-way set associative, cache line 64 байта, один порт доступа, разделяется двумя аппаратными потоками (при включенном Hyper-Threading)
- **L1 Data Cache:** 32 KiB, 8-way set associative, cache line 64 байта, один порт доступа, разделяется двумя аппаратными потоками (при включенном Hyper-Threading)
- **L2 Cache:** 256 KiB, 8-way set associative, cache line 64 байта, unified cache (для инструкций и данных), write policy: write-back, non-inclusive
- **L3 Cache:** 8 MiB, 16-way set associative, cache line 64 байта, unified cache (для инструкций и данных), write policy: write-back, inclusive (если данные/инструкции есть в L1 или L2, то они будут находится и в L3, 4-bit valid vector)

# Cache-Coherence Protocols

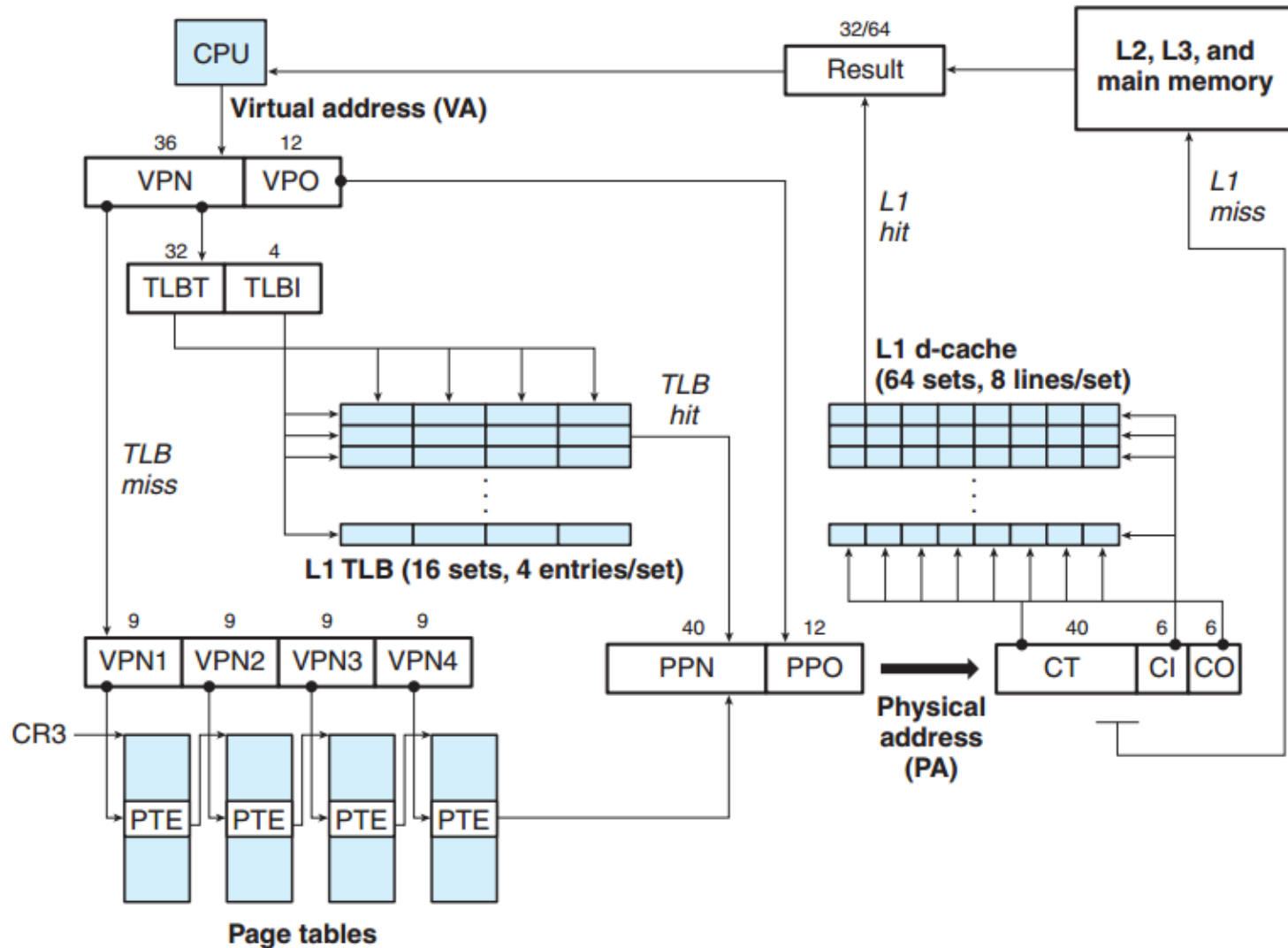
---

- **Когерентность кеш-памяти**  
(cache coherence) – свойство кеш-памяти, означающее согласованность данных, хранящихся в локальных кешах, с данными в оперативной памяти



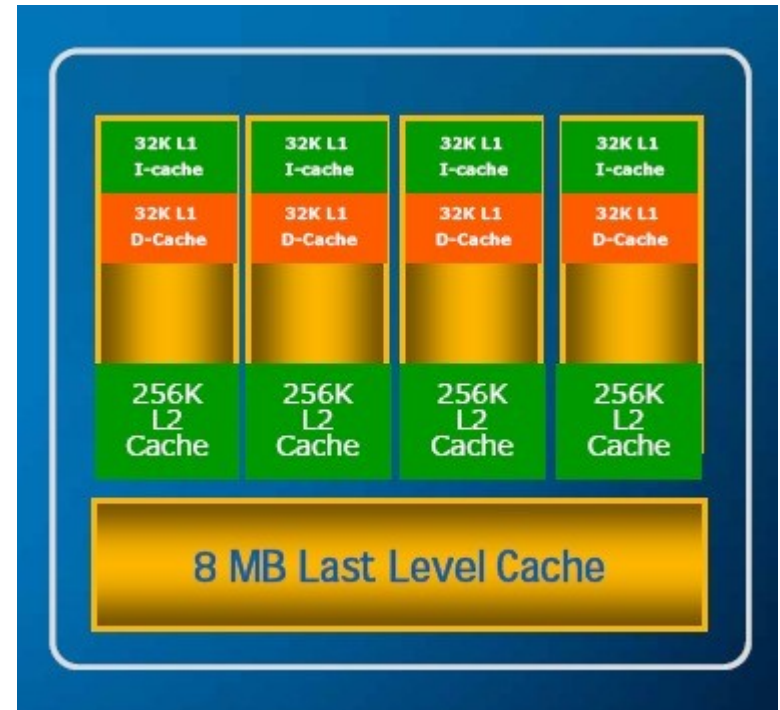
- **Протоколы поддержки когерентности кешей**
  - ☐ MSI
  - ☐ MESI
  - ☐ MOSI
  - ☐ MOESI (AMD Opteron)
  - ☐ MESI+F (Modified, Exclusive, Shared, Invalid and Forwarding): Intel Nehalem (via QPI)

# Intel Core i7 address translation



# Intel Nehalem

- **L1 Cache:**  
32 KiB кеш инструкций,  
32 KiB кеш данных
- **L2 Cache:** 256 KiB кеш
- **L3 Cache:** 8 MiB,  
общий для всех ядер процессора

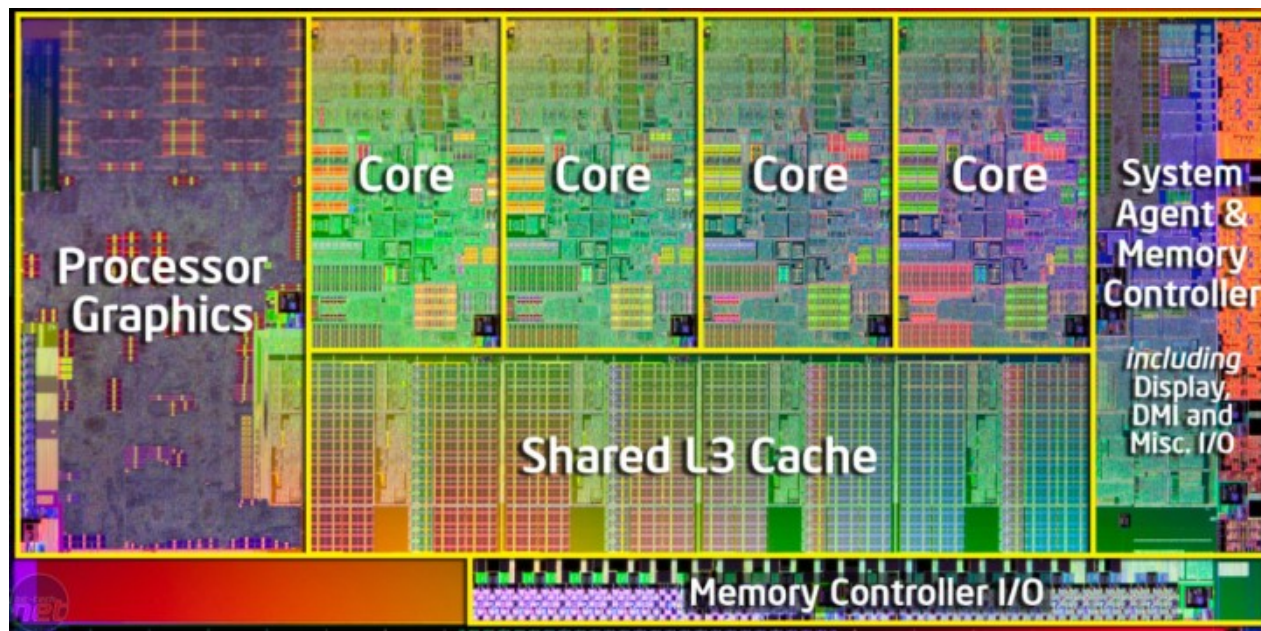


Nehalem Caches (Latency & Bandwidth)	
L1 Cache Latency	4 cycles (16 b/cycle)
L2 Cache Latency	10 cycles (32 b/cycle to L1)
L3 Cache Latency	52 cycles

# Intel Sandy Bridge

---

- **L0 Cache:** кеш для 1500 декодированных микроопераций (uops)
- **L1 Cache:** 32 KiB кеш инструкций, 32 KiB кеш данных (4 cycles, load 32 b/cycle, store 16 b/cycle)
- **L2 Cache:** 256 KiB (11 cycles, 32 b/cycle to L1)
- **LLC Cache (Last Level Cache, L3):** 8 MiB, общий для всех ядер процессора и ядер интегрированного GPU! (25 cycles, 256 b/cycle)

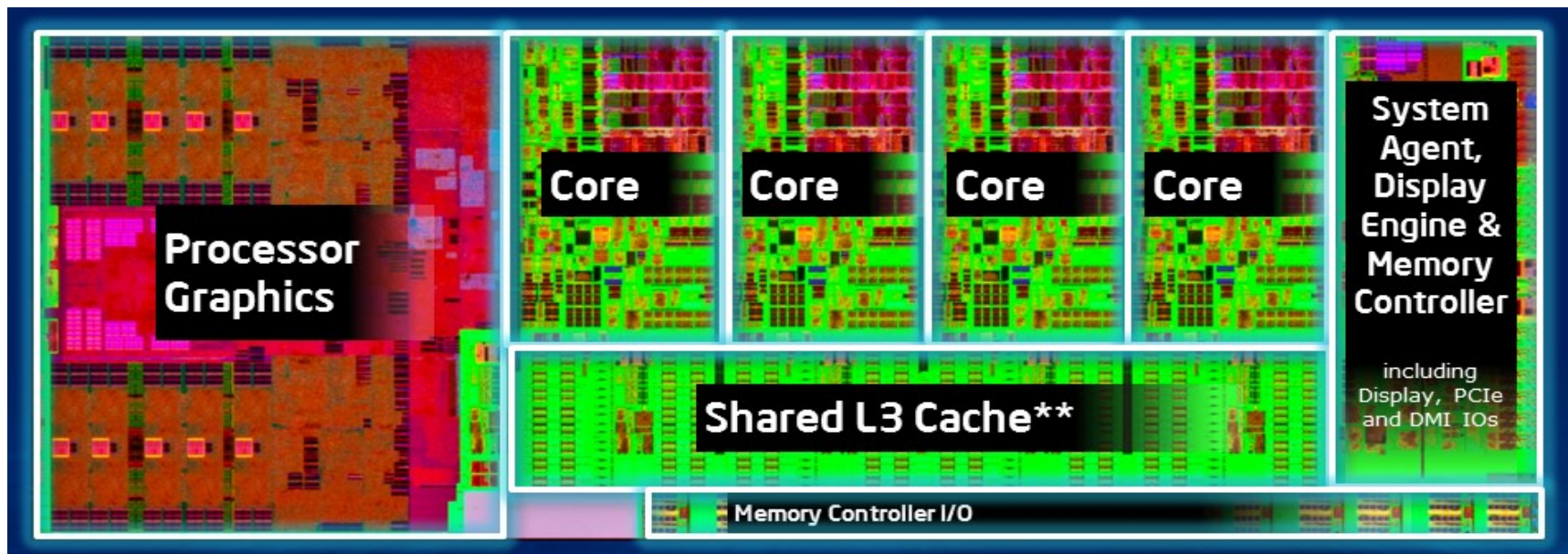




# Intel Haswell

---

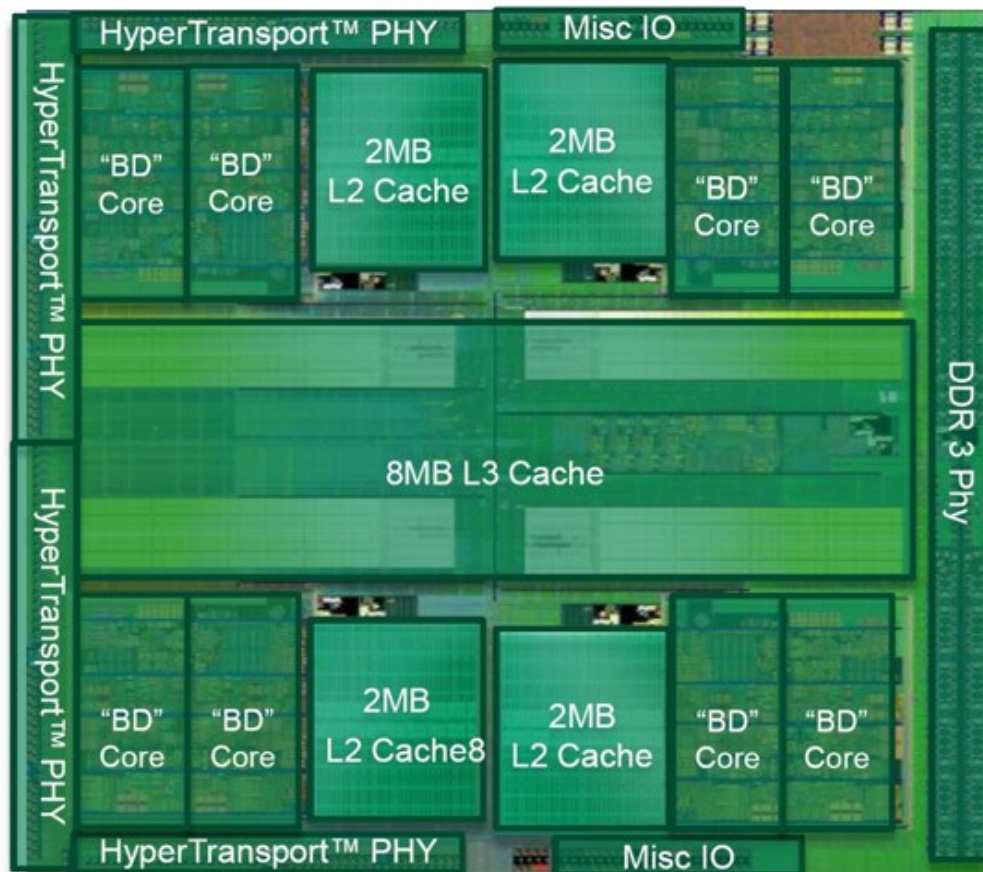
- **L1 Cache:** 32 KiB кеш инструкций, 32 KiB кеш данных (4 cycles, load 64 b/cycle, store 32 b/cycle)
- **L2 Cache:** 256 KiB кеш (11 cycles, 64 b/cycle to L1)
- **LLC Cache (Last Level Cache, L3):** 8 MiB, общий для всех ядер процессора и ядер интегрированного GPU



# AMD Bulldozer

---

- **L1d**: 16 KiB per cluster; **L1i**: 64 KiB per core
- **L2**: 2 MiB per module
- **L3**: 8 MiB



# AMD Bulldozer Cache/Memory Latency

---

	L1 Cache (clocks)	L2 Cache (clocks)	L3 Cache (clocks)	Main Memory (clocks)
<b>AMD FX-8150 (3.6 GHz)</b>	4	21	65	195
<b>AMD Phenom II X4 975 BE (3.6 GHz)</b>	3	15	59	182
<b>AMD Phenom II X6 1100T (3.3 GHz)</b>	3	14	55	157
<b>Intel Core i5 2500K (3.3 GHz)</b>	4	11	25	148

<http://www.anandtech.com/show/4955/the-bulldozer-review-amd-fx8150-tested>

# Windows CPU Cache Information (CPU-Z)

The screenshot shows the CPU-Z application window with the 'CPU' tab selected. The 'Processor' section displays 'Intel Sandy Bridge' and 'Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz'. The 'Cache' section, highlighted with a red dashed border, shows 'L1 Data: 32 KBytes 8-way' and 'Level 2: 256 KBytes 8-way'. The 'Clocks (Core #0)' section shows 'Core Speed: 2495.2 MHz'. The 'Selection' dropdown is set to 'Processor #1', and 'Cores' and 'Threads' are both set to 1.

Cache		
L1 Data	32 KBytes	8-way
L1 Inst.		
Level 2	256 KBytes	8-way
Level 3		

- **L1 Data:**  
32 KB, 8-way
- **L2**  
256 KB, 8-way

Информация о структуре кеш-памяти может быть получена инструкцией  
**CPUID**

# GNU/Linux CPU Cache Information (/proc)

---

```
$ cat /proc/cpuinfo
processor      : 0
...
model name    : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
stepping      : 7
microcode     : 0x29
cpu MHz       : 2975.000
cache size    : 3072 KB
physical id   : 0
siblings      : 4
core id       : 0
cpu cores     : 2
apicid        : 0
initial apicid : 0
...
bogomips      : 4983.45
clflush size  : 64
cache_alignment : 64
address sizes  : 36 bits physical, 48 bits virtual
```

# GNU/Linux CPU Cache Information (/sys)

---

```
/sys/devices/system/cpu/cpu0/cache
```

```
index0/
```

```
    coherency_line_size
```

```
    number_of_sets
```

```
    shared_cpu_list  size
```

```
    ways_of_associativity
```

```
    level
```

```
    physical_line_partition
```

```
    shared_cpu_map
```

```
    type
```

```
index1/
```

```
index2/
```

```
...
```

# GNU/Linux CPU Cache Information (SMBIOS)

```
# dmidecode -t cache
SMBIOS 2.6 present.
Handle 0x0002, DMI type 7, 19 bytes
Cache Information
    Socket Designation: L1-Cache
    Configuration: Enabled, Not Socketed, Level 1
    Operational Mode: Write Through
    Location: Internal
    Installed Size: 64 kB
    Maximum Size: 64 kB
    Supported SRAM Types:
        Synchronous
    Installed SRAM Type: Synchronous
    Speed: Unknown
    Error Correction Type: Single-bit ECC
    System Type: Data
    Associativity: 8-way Set-associative

Handle 0x0003, DMI type 7, 19 bytes
Cache Information
    Socket Designation: L2-Cache
    Configuration: Enabled, Not Socketed, Level 2
    Operational Mode: Write Through
```

# CPU Cache Information

---

```
// Microsoft Windows
BOOL WINAPI GetLogicalProcessorInformation(
    _Out_ PSYSTEM_LOGICAL_PROCESSOR_INFORMATION Buffer,
    _Inout_ PDWORD ReturnLength
);
```

```
// GNU/Linux
#include <unistd.h>
long sysconf(int name); /* name = _SC_CACHE_LINE */
```

```
// CPUID
```



# Программные симуляторы кеш-памяти

---

- **SimpleScalar** (<http://www.simplescalar.com>)
  - ❑ Симулятор суперскалярного процессора с внеочередным выполнением команд
  - ❑ `cache.c` – реализации логики работы кеш-памяти (функция `int cache_access(...)`)
- **MARSSx86** (Micro-ARchitectural and System Simulator for x86-based Systems, <http://marss86.org>)
- **PTLsim** – cycle accurate microprocessor simulator and virtual machine for the x86 and x86-64 instruction sets ([www.ptlsim.org](http://www.ptlsim.org))
- **MARS** (MIPS Assembler and Runtime Simulator) <http://courses.missouristate.edu/kenvollmar/mars/>
- **CACTI** – an integrated cache access time, cycle time, area, leakage, and dynamic power model for cache architectures <http://www.cs.utah.edu/~rajeev/cacti6/>

# Суммирование элементов массива

---

```
int sumarray3d_def(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                sum += a[k][i][j];
            }
        }
    }
    return sum;
}
```

Массив `a[3][3][3]` хранится в памяти строка за строкой (row-major order)

*Reference pattern: stride-(N\*N)*

Address 0	a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[0][2][0]	a[0][2][1]	a[0][2][2]
36	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]	a[1][2][0]	a[1][2][1]	a[1][2][2]
72	a[2][0][0]	a[2][0][1]	a[2][0][2]	a[2][1][0]	a[2][1][1]	a[2][1][2]	a[2][2][0]	a[2][2][1]	a[2][2][2]

# Суммирование элементов массива

---

```
int sumarray3d_def(int a[N][N][N])
{
    int i, j, k, sum = 0;

    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            for (k = 0; k < N; k++) {
                sum += a[i][j][k];
            }
        }
    }
    return sum;
}
```

Массив `a[3][3][3]` хранится в памяти строка за строкой (row-major order)

*Reference pattern: stride-1*

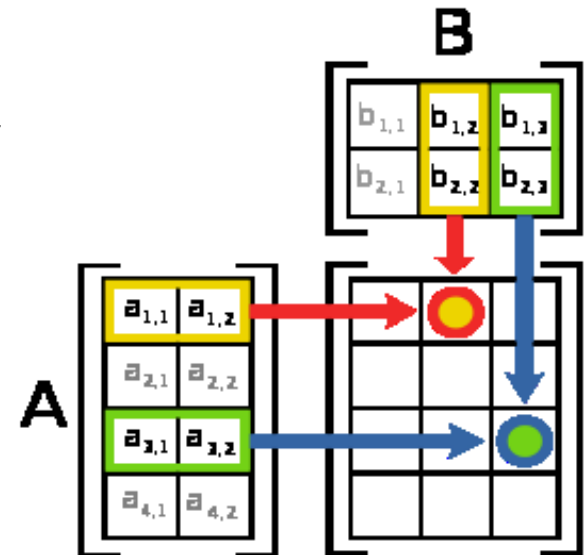
Address 0	a[0][0][0]	a[0][0][1]	a[0][0][2]	a[0][1][0]	a[0][1][1]	a[0][1][2]	a[0][2][0]	a[0][2][1]	a[0][2][2]
36	a[1][0][0]	a[1][0][1]	a[1][0][2]	a[1][1][0]	a[1][1][1]	a[1][1][2]	a[1][2][0]	a[1][2][1]	a[1][2][2]
72	a[2][0][0]	a[2][0][1]	a[2][0][2]	a[2][1][0]	a[2][1][1]	a[2][1][2]	a[2][2][0]	a[2][2][1]	a[2][2][2]

# Умножение матриц (DGEMM) v1.0

```
for (i = 0; i < N; i++) {  
    for (j = 0; j < N; j++) {  
        for (k = 0; k < N; k++) {  
            c[i][j] += a[i][k] * b[k][j];  
        }  
    }  
}
```

**DGEMM –**

Double precision GEneral Matrix Multiply



# Умножение матриц (DGEMM) v1.0

**a**

0, 0	0, 1	0, 2	...	0, N-1
1, 0	1, 1	1, 2	...	1, N-1
...				
<b><i>i</i>, 0</b>	<b><i>i</i>, 1</b>	<b><i>i</i>, 2</b>	...	<b><i>i</i>, N-1</b>
...				

**b**

0, 0	0, 1	...	<b>0, <i>j</i></b>	...
1, 0	1, 1		<b>1, <i>j</i></b>	
2, 0	2, 1		<b>1, <i>j</i></b>	
...	...		...	
N-1, 0	N-1, 1		<b>N-1, <i>j</i></b>	

- Read  $a[i, 0]$  - cache miss
- Read  $b[0, j]$  - cache miss
- Read  $a[i, 1]$  - **cache hit**
- Read  $b[1, j]$  - cache miss
- Read  $a[i, 2]$  - **cache hit**
- Read  $b[2, j]$  - cache miss
- ...

```
c[i][j] += a[i][k] * b[k][j];
```

# Умножение матриц (DGEMM) v2.0

```
for (i = 0; i < N; i++) {  
    for (k = 0; k < N; k++) {  
        for (j = 0; j < N; j++) {  
            c[i][j] += a[i][k] * b[k][j];  
        }  
    }  
}
```

**a**


**b**


**Доступ по адресам, последовательно  
расположенным в памяти!**

# Умножение матриц (DGEMM)

---

- Процессор Intel Core i5 2520M (2.50 GHz)
- GCC 4.8.1
- CFLAGS = -O0 -Wall -g
- N = 512 (double)

	DGEMM v1.0	DGEMM v2.0
Time (sec.)	1.0342	0.7625
Speedup	-	<b>1.36</b>

# Умножение матриц (DGEMM) v3.0

```
for (i = 0; i < n; i += BS) {  
    for (j = 0; j < n; j += BS) {  
        for (k = 0; k < n; k += BS) {  
            for (i0 = 0, c0 = (c + i * n + j),  
                 a0 = (a + i * n + k); i0 < BS;  
                 ++i0, c0 += n, a0 += n)  
            {  
                for (k0 = 0, b0 = (b + k * n + j);  
                     k0 < BS; ++k0, b0 += n)  
                {  
                    for (j0 = 0; j0 < BS; ++j0) {  
                        c0[j0] += a0[k0] * b0[j0];  
                    }  
                }  
            }  
        }  
    }  
}
```

**Блочный алгоритм умножения матриц**

Подматрицы могут поместит в кеш-память

(Cache-oblivious algorithms)



# Умножение матриц

---

- Процессор Intel Core i5 2520M (2.50 GHz)
- GCC 4.8.1
- CFLAGS = -O0 -Wall -g
- N = 512 (double)

	DGEMM v1.0	DGEMM v2.0	DGEMM v3.0
Time (sec.)	1.0342	0.7625	0.5627
Speedup	-	1.36	<b>1.84</b>



- **DGEMM v4.0:** Loop unrolling, SIMD (SSE, AVX), OpenMP, ... 57

# Профилирование DGEMM v1.0

---

```
$ perf stat -e cache-misses ./dgemm-v1  
Elapsed time: 1.039159 sec.
```

```
Performance counter stats for './dgemm-v1':
```

```
4,027,708 cache-misses
```

```
3.739953330 seconds time elapsed
```

# Профилирование DGEMM v3.0

---

```
$ perf stat -e cache-misses ./dgemm-v3  
Elapsed time: 0.563926 sec.
```

```
Performance counter stats for './dgemm-v3':
```

```
368,594 cache-misses
```

```
2.317988237 seconds time elapsed
```

# Оптимизация структур I

```
struct data {  
    int a;      /* 4 байта */  
    int b;      /* 4 байта */  
    int c;      /* 4 байта */  
    int d;      /* 4 байта */  
};  
  
struct data *p;  
  
for (i = 0; i < N; i++) {  
    p[i].a = p[i].b;      // Используются только a и b  
}
```

**Cache line (64 байта) после чтения p[0].b:**

b	c	d	a	b	c	d	a	b	c	d	a	b	c	d	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Оптимизация структур I

```
// Split into 2 structures
struct data {
    int a;    /* 4 байта */
    int b;    /* 4 байта */
};

struct data *p;

for (i = 0; i < N; i++) {
    p[i].a = p[i].b;
}
```

Cache line (64 байта) после чтения p[0].b:

b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Оптимизация структур I

```
// Split into 2 structures
struct data {
    int a;    /* 4 байта */
    int b;    /* 4 байта */
};

struct data *p;

for (i = 0; i < N; i++) {
    p[i].a = p[i].b;
}
```

**Speedup 1.37**

- $N = 10 * 1024 * 1024$
- GCC 4.8.1 (Fedora 19 x86\_64)
- CFLAGS = -O0 -Wall -g
- Intel Core i5 2520M

**Cache line (64 байта) после чтения p[0].b:**

b	a	b	a	b	a	b	a	b	a	b	a	b	a	b	a
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

# Оптимизация структур II

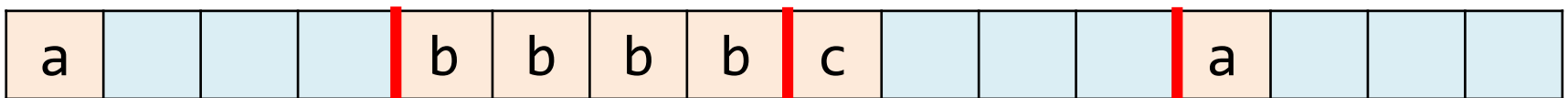
```
struct data {  
    char a;  
    /* padding 3 bytes */  
    int b;  
    char c;  
    /* padding: 3 bytes */  
};
```

```
struct data *p;
```

```
for (i = 0; i < N; i++) {  
    p[i].a++;  
}
```

Компилятор  
выравнивает структуру:  
`sizeof(struct data) = 12`

Cache line (64 байта):



# Выравнивание структур на x86

---

## x86

- char (one byte) will be 1-byte aligned.
- short (two bytes) will be 2-byte aligned.
- int (four bytes) will be 4-byte aligned.
- long (four bytes) will be 4-byte aligned.
- float (four bytes) will be 4-byte aligned.
- double (eight bytes) will be 8-byte aligned on Windows and 4-byte aligned on Linux.
- pointer (four bytes) will be 4-byte aligned.

## x86\_64

- long (eight bytes) will be 8-byte aligned.
- double (eight bytes) will be 8-byte aligned.
- pointer (eight bytes) will be 8-byte aligned.

**Размер структуры должен быть кратен размеру самого большого поля**

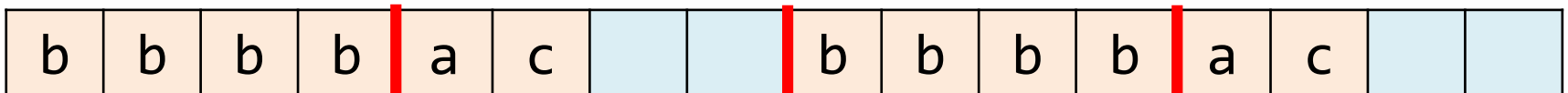


# Оптимизация структур II

```
struct data {  
    int b;  
    char a;  
    char c;  
    /* padding 2 bytes */  
};  
  
struct data *p;  
  
for (i = 0; i < N; i++) {  
    p[i].a++;  
}
```

Компилятор  
выравнивает структуру:  
`sizeof(struct data) = 8`

Cache line (64 байта):



# Оптимизация структур II

```
struct data {  
    int b;  
    char a;  
    char c;  
    /* padding 2 bytes */  
};
```

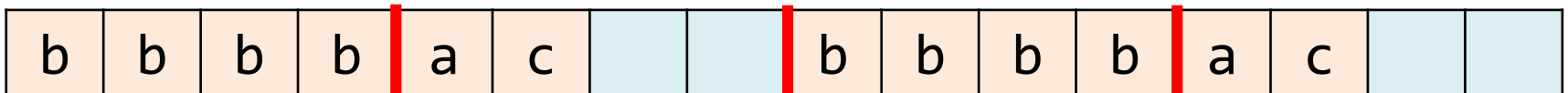
```
struct data *p;  
  
for (i = 0; i < N; i++) {  
    p[i].a++;  
}
```

Компилятор  
выравнивает структуры:  
`sizeof(struct data) = 8`

**Speedup 1.21**

- $N = 10 * 1024 * 1024$
- GCC 4.8.1 (Fedora 19 x86\_64)
- CFLAGS = -O0 -Wall -g
- Intel Core i5 2520M

**Cache line (64 байта):**



# Оптимизация структур III

```
#define SIZE 65
```

```
struct point {  
    double x;          /* 8-byte aligned */  
    double y;          /* 8-byte aligned */  
    double z;          /* 8-byte aligned */  
    int data[SIZE];    /* 8-byte aligned */  
};
```

```
struct point *points;
```

```
for (i = 0; i < N; i++) {  
    d[i] = sqrt(points[i].x * points[i].x +  
                points[i].y * points[i].y);  
}
```

**sizeof(struct point) = 288**  
(4 байта выравнивания)

# Оптимизация структур III

## Cache line (64 байта)

x	y	z	data[0], data[1], ..., data[9]
---	---	---	--------------------------------

```
double y;           /* 8-byte aligned */
double z;           /* 8-byte aligned */
int data[SIZE];     /* 8-byte aligned */
};

struct point *points;

for (i = 0; i < N; i++) {
    d[i] = sqrt(points[i].x * points[i].x +
                points[i].y * points[i].y);
}
```

# Оптимизация структур III

```
struct point1 {  
    double x;  
    double y;  
    double z;  
};  
  
struct point2 {  
    int data[SIZE];  
}  
  
struct point1 *points1;  
struct point2 *points2;  
  
for (i = 0; i < N; i++) {  
    d[i] = sqrt(points1[i].x * points1[i].x +  
                points1[i].y * points1[i].y);  
}
```

# Оптимизация структур III

Cache line (64 байта)

x	y	z	x	y	z	x	y
---	---	---	---	---	---	---	---

```
};
```

```
struct point2 {  
    int data[SIZE];  
}
```

```
struct point1 *points1;  
struct point2 *points2;
```

```
for (i = 0; i < N; i++) {  
    d[i] = sqrt(points1[i].x * points1[i].x +  
                points1[i].y * points1[i].y);  
}
```

# Записная книжка v1.0

```
#define NAME_MAX 16
```

```
struct phonebook {  
    char lastname[NAME_MAX];  
    char firstname[NAME_MAX];  
    char email[16];  
    char phone[10];  
    char cell[10];  
    char addr1[16];  
    char addr2[16];  
    char city[16];  
    char state[2];  
    char zip[5];  
    struct phonebook *next;  
};
```

- Записи хранятся в односвязном списке
- Размер структуры 136 байт

## Записная книжка v1.0

---

```
struct phonebook *phonebook_lookup(  
    struct phonebook *head, char *lastname)  
{  
    struct phonebook *p;  
  
    for (p = head; p != NULL; p = p->next) {  
        if (strcmp(p->lastname, lastname) == 0) {  
            return p;  
        }  
    }  
    return NULL;  
}
```



# Записная книжка v1.0

---

```
struct phonebook *phonebook_lookup(  
    struct phonebook *head, char *lastname)  
{  
    struct phonebook *p;  
  
    for (p = head; p != NULL; p = p->next) {  
        if (strcmp(p->lastname, lastname) == 0) {  
            return p;  
        }  
    }  
}
```

- При обращении к полю **p->lastname** в кеш-память загружаются поля: **firstname**, **email**, **phone**, ...
- На каждой итерации происходит “промах” (cache miss) при чтении поля **p->lastname**

# Записная книжка v2.0

```
struct phonebook {  
    char firstname[NAME_MAX];  
    char email[16];  
    char phone[10];  
    char cell[10];  
    char addr1[16];  
    char addr2[16];  
    char city[16];  
    char state[2];  
    char zip[5];  
    struct phonebook *next;  
};
```

- Последовательное размещение в памяти полей **lastname**
- Массив можно сделать динамическим

```
char lastnames[SIZE_MAX][NAME_MAX];  
struct phonebook phonebook[SIZE_MAX];  
int nrecords;
```

## Записная книжка v2.0

---

```
struct phonebook *phonebook_lookup(char *lastname)
{
    int i;

    for (i = 0; i < nrecords; i++) {
        if (strcmp(lastnames[i], lastname) == 0) {
            return &phonebook[i];
        }
    }
    return NULL;
}
```

# Записная книжка (lookup performance)

---

- Intel Core i5 2520M (2.50 GHz)
- GCC 4.8.1
- CFLAGS = -O0 -Wall
- Количество записей: 10 000 000 (random lastname[16])

	PhoneBook Lookup Performance	
	Linked list (v1.0)	1D array (v2.0)
Cache misses (Linux perf)	1 689 046	622 152
Time (sec)	0.017512	0.005457
Speedup		<b>3.21</b>

# Литература

---

- Randal E. Bryant, David R. O'Hallaron. **Computer Systems: A Programmer's Perspective**. - Addison-Wesley, 2010
- Drepper Ulrich. **What Every Programmer Should Know About Memory** // <http://www.akkadia.org/drepper/cpumemory.pdf>
- Ричард Гербер, Арт Бик, Кевин Смит, Ксинмин Тиан. **Оптимизация ПО. Сборник рецептов**. - СПб.: Питер, 2010
- Agner Fog. **Optimizing subroutines in assembly language: An optimization guide for x86 platforms** // [http://www.agner.org/optimize/optimizing\\_assembly.pdf](http://www.agner.org/optimize/optimizing_assembly.pdf)
- **Intel 64 and IA-32 Architectures Optimization Reference Manual**
- Herb Sutter. **Machine Architecture: Things Your Programming Language Never Told You** // [http://www.nwcpp.org/Downloads/2007/Machine\\_Architecture\\_-\\_NWCPP.pdf](http://www.nwcpp.org/Downloads/2007/Machine_Architecture_-_NWCPP.pdf)
- David Levinthal. **Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors** // [http://software.intel.com/sites/products/collateral/hpc/vtune/performance\\_analysis\\_guide.pdf](http://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf)
- **System V Application Binary Interface (AMD64 Architecture Processor Supplement)** // <http://www.x86-64.org/documentation/abi.pdf>