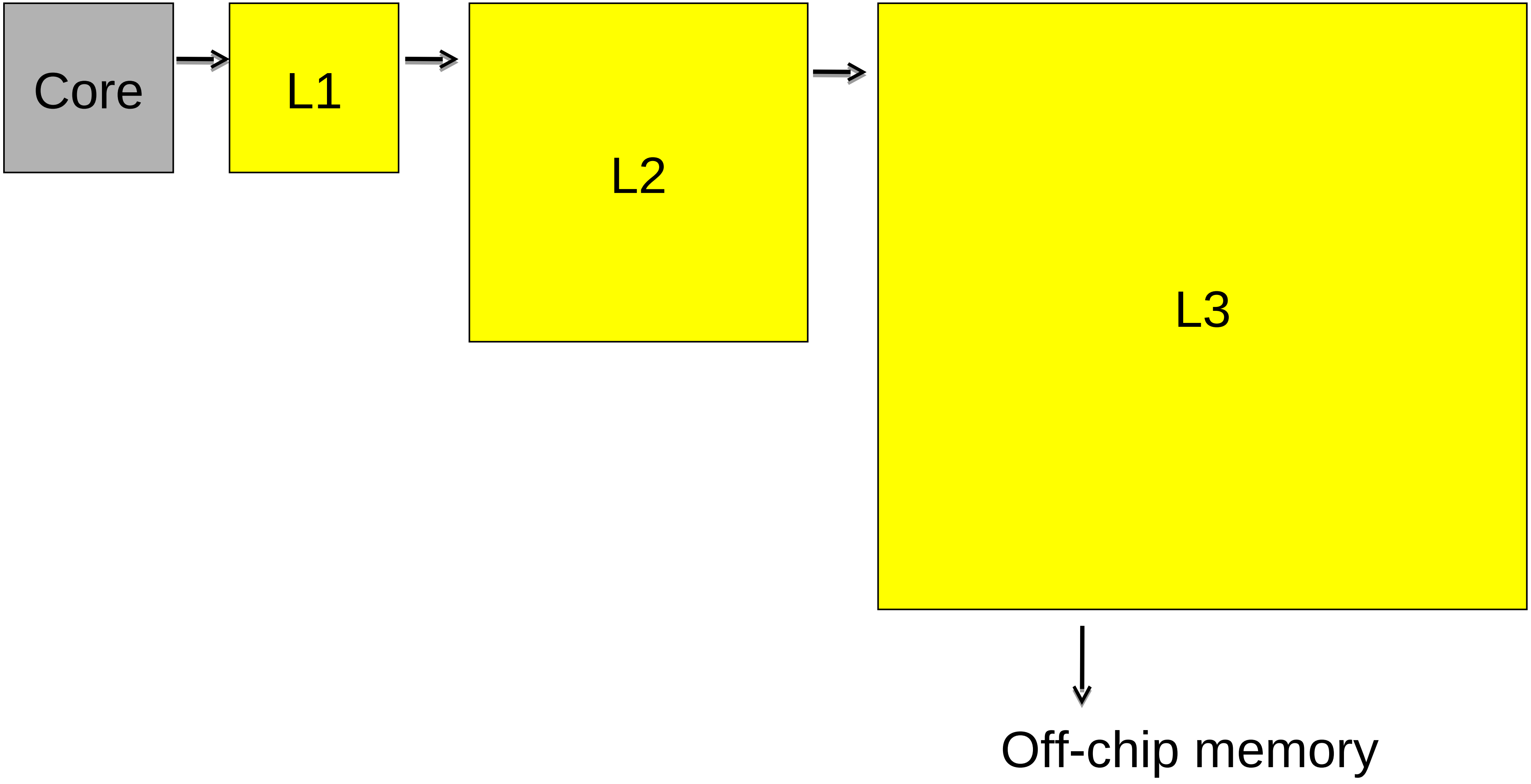
250P: Computer Systems Architecture

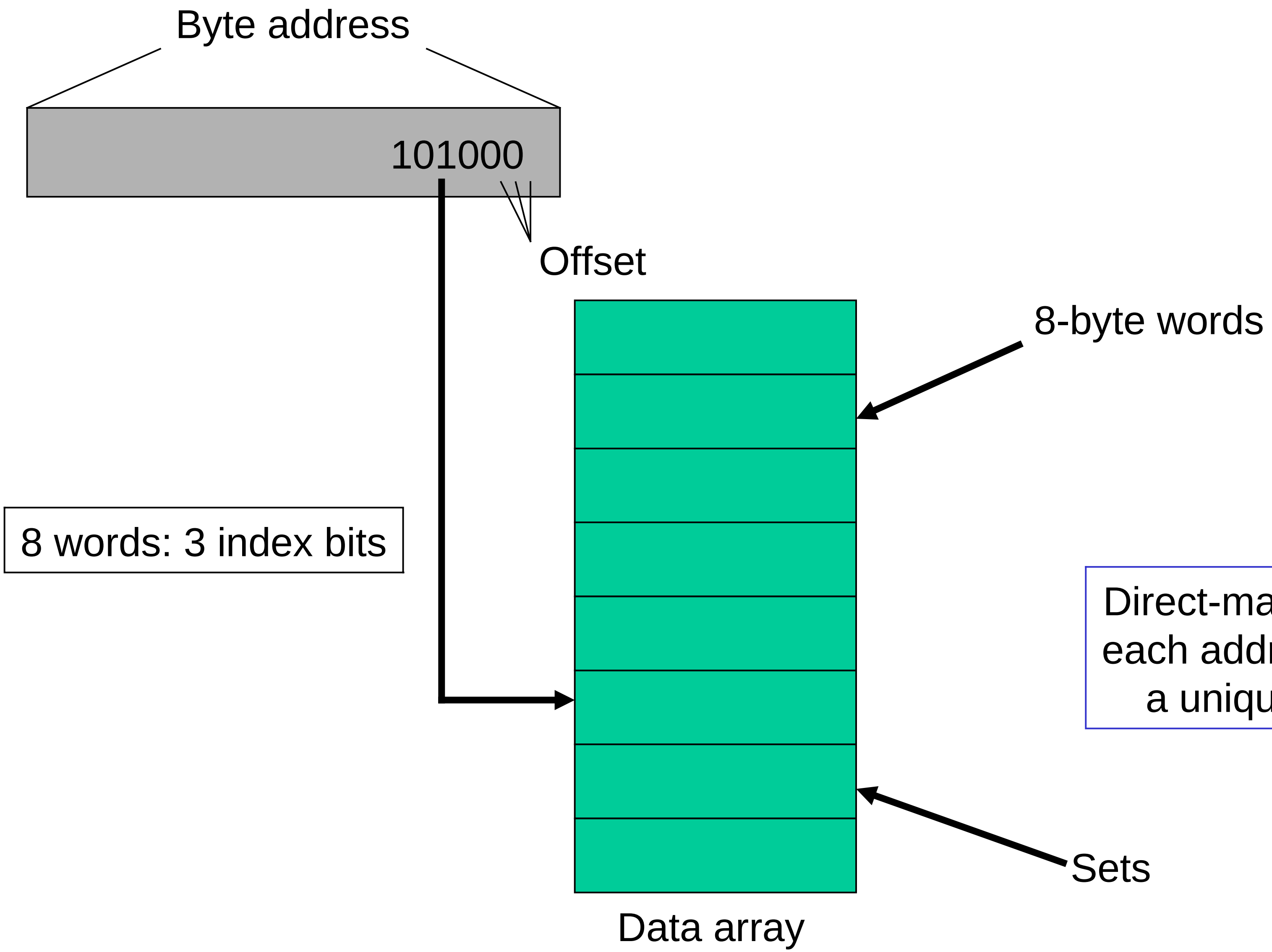
Lecture 10: Caches

Anton Burtsev
April, 2021

The Cache Hierarchy



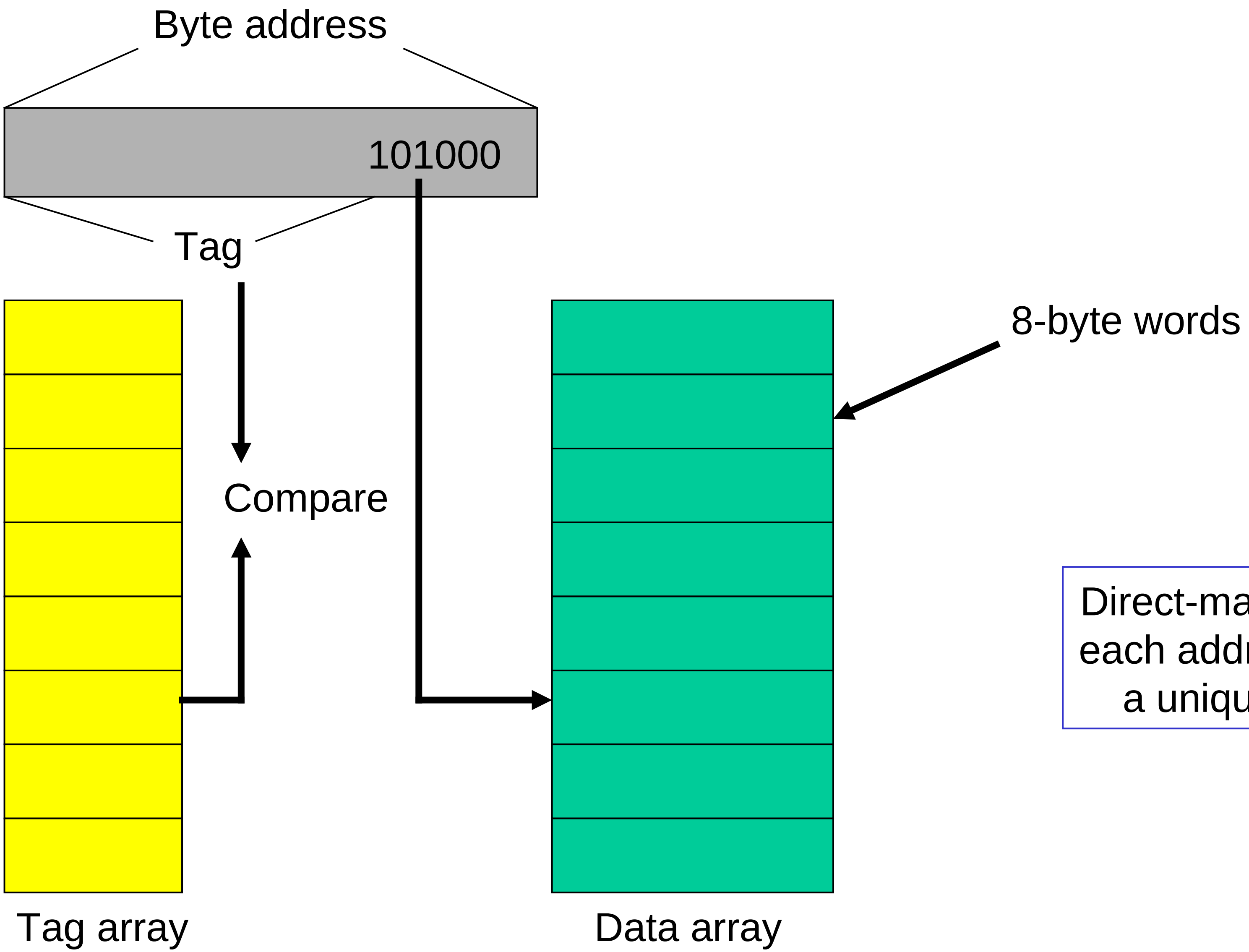
Accessing the Cache



8 words: 3 index bits

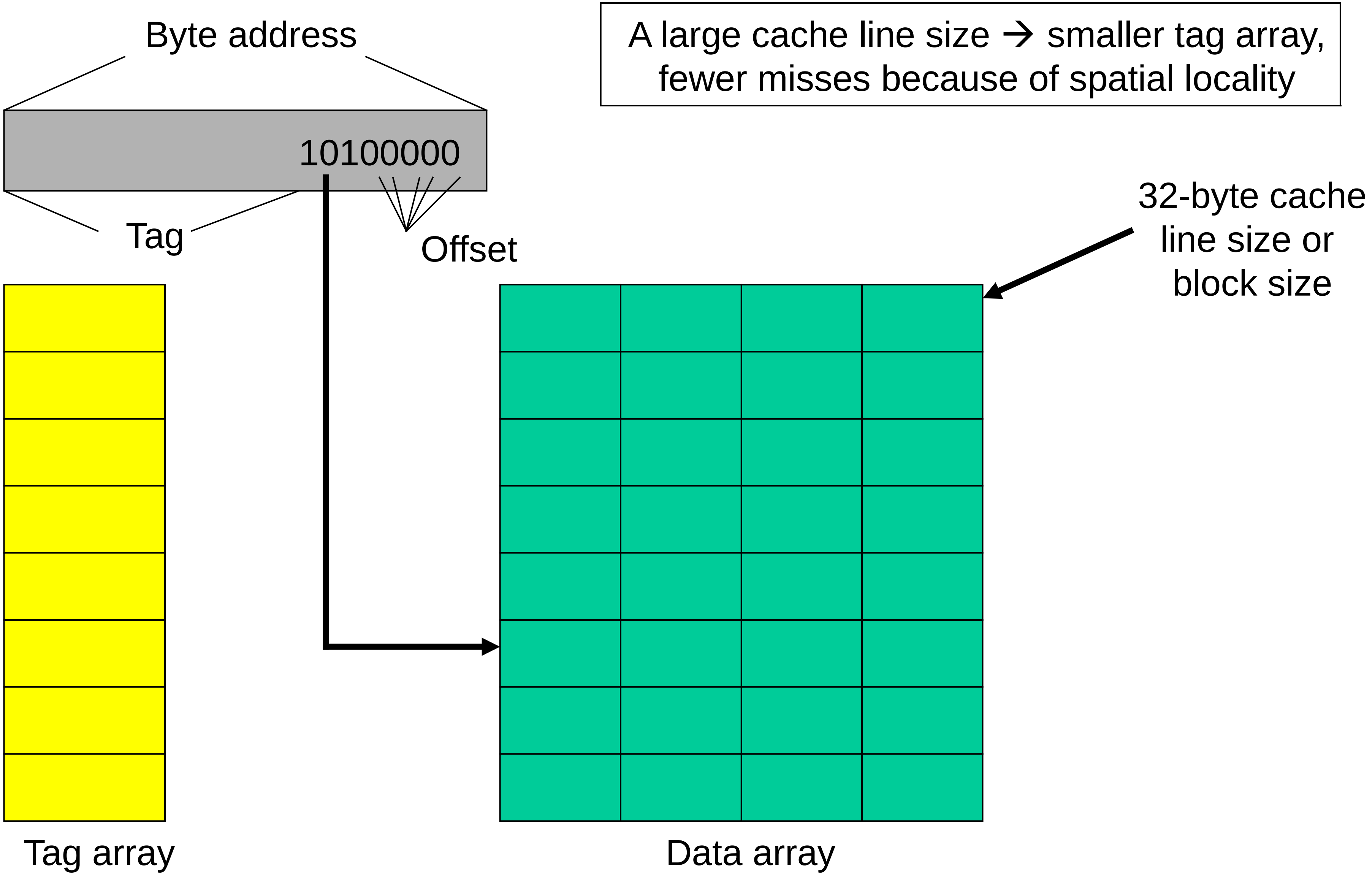
Direct-mapped cache: each address maps to a unique address

The Tag Array



Direct-mapped cache: each address maps to a unique address

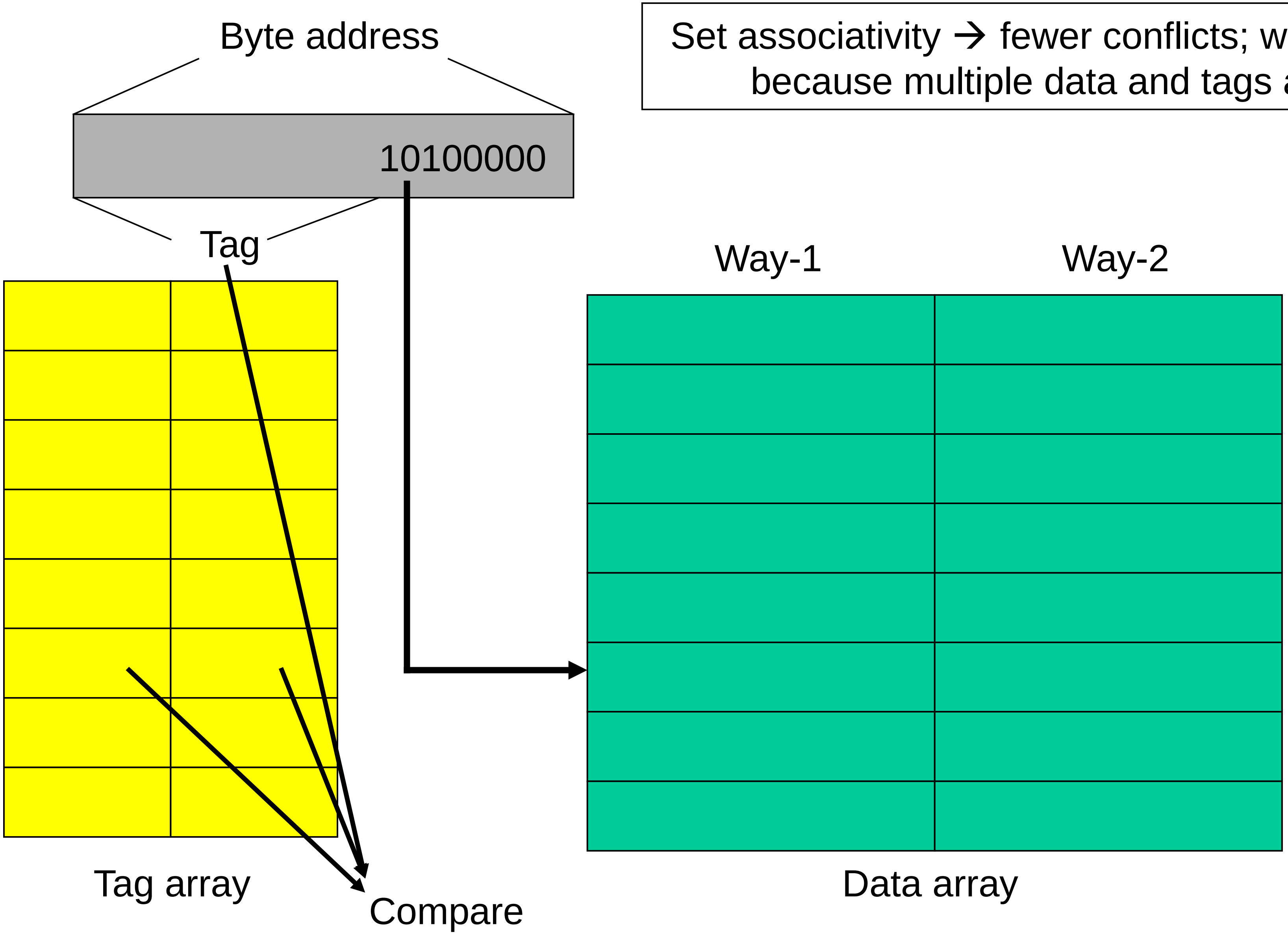
Increasing Line Size



A large cache line size → smaller tag array, fewer misses because of spatial locality

Associativity

Set associativity → fewer conflicts; wasted power because multiple data and tags are read



Example

- 32 KB 4-way set-associative data cache array with 32 byte line sizes

- How many sets?

- How many index bits, offset bits, tag bits?

- How large is the tag array?

Types of Cache Misses

- Compulsory misses: happens the first time a memory word is accessed – the misses for an infinite cache

- Capacity misses: happens because the program touched many other words before re-touching the same word – the misses for a fully-associative cache

- Conflict misses: happens because two words map to the same location in the cache – the misses generated while moving from a fully-associative to a direct-mapped cache

- Sidenote: can a fully-associative cache have more misses than a direct-mapped cache of the same size?

Reducing Miss Rate

- Large block size – reduces compulsory misses, reduces miss penalty in case of spatial locality – increases traffic between different levels, space waste, and conflict misses

- Large cache – reduces capacity/conflict misses – access time penalty

- High associativity – reduces conflict misses – rule of thumb: 2-way cache of capacity N/2 has the same miss rate as 1-way cache of capacity N – more energy

More Cache Basics

- L1 caches are split as instruction and data; L2 and L3 are unified
- The L1/L2 hierarchy can be inclusive, exclusive, or non-inclusive
- On a write, you can do write-allocate or write-no-allocate
- On a write, you can do writeback or write-through; write-back reduces traffic, write-through simplifies coherence
- Reads get higher priority; writes are usually buffered
- L1 does parallel tag/data access; L2/L3 does serial tag/data

Techniques to Reduce Cache Misses

- Victim caches
- Better replacement policies – pseudo-LRU, NRU, DRRIP
- Cache compression

Victim Caches

- A direct-mapped cache suffers from misses because multiple pieces of data map to the same location

- The processor often tries to access data that it recently discarded – all discards are placed in a small victim cache (4 or 8 entries) – the victim cache is checked before going to L2

- Can be viewed as additional associativity for a few sets that tend to have the most conflicts

Tolerating Miss Penalty

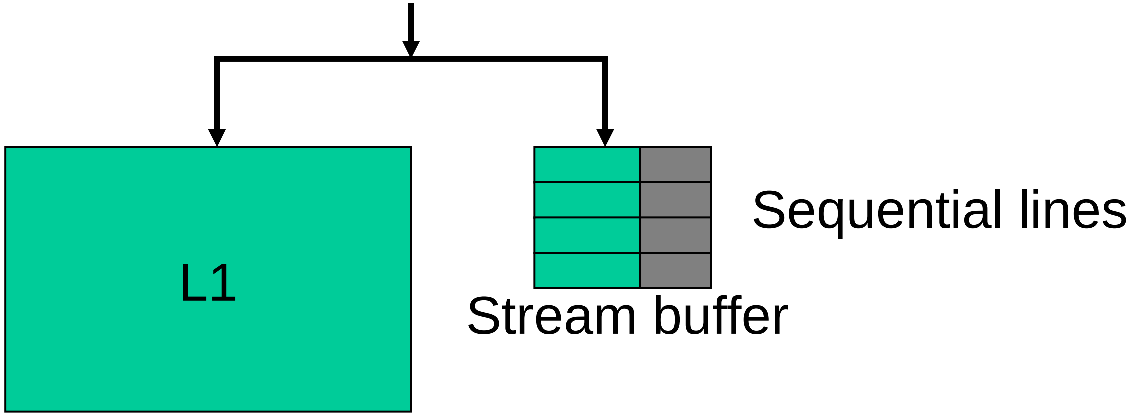
- Out of order execution: can do other useful work while waiting for the miss – can have multiple cache misses -- cache controller has to keep track of multiple outstanding misses (non-blocking cache)

- Hardware and software prefetching into prefetch buffers – aggressive prefetching can increase contention for buses

Stream Buffers

- Simplest form of prefetch: on every miss, bring in multiple cache lines

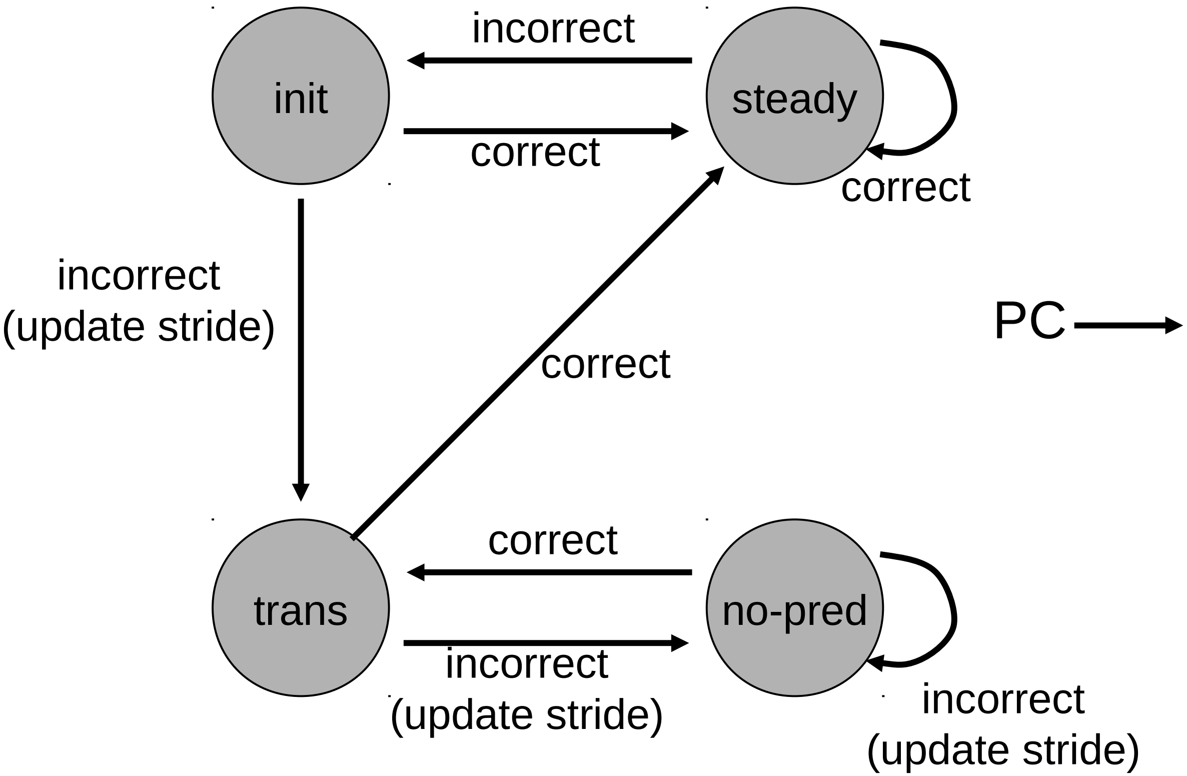
- When you read the top of the queue, bring in the next line



Stride-Based Prefetching

- For each load, keep track of the last address accessed by the load and a possibly consistent stride

- FSM detects consistent stride and issues prefetches



PC →

tag	prev_addr	stride	state

Prefetching

- Hardware prefetching can be employed for any of the cache levels

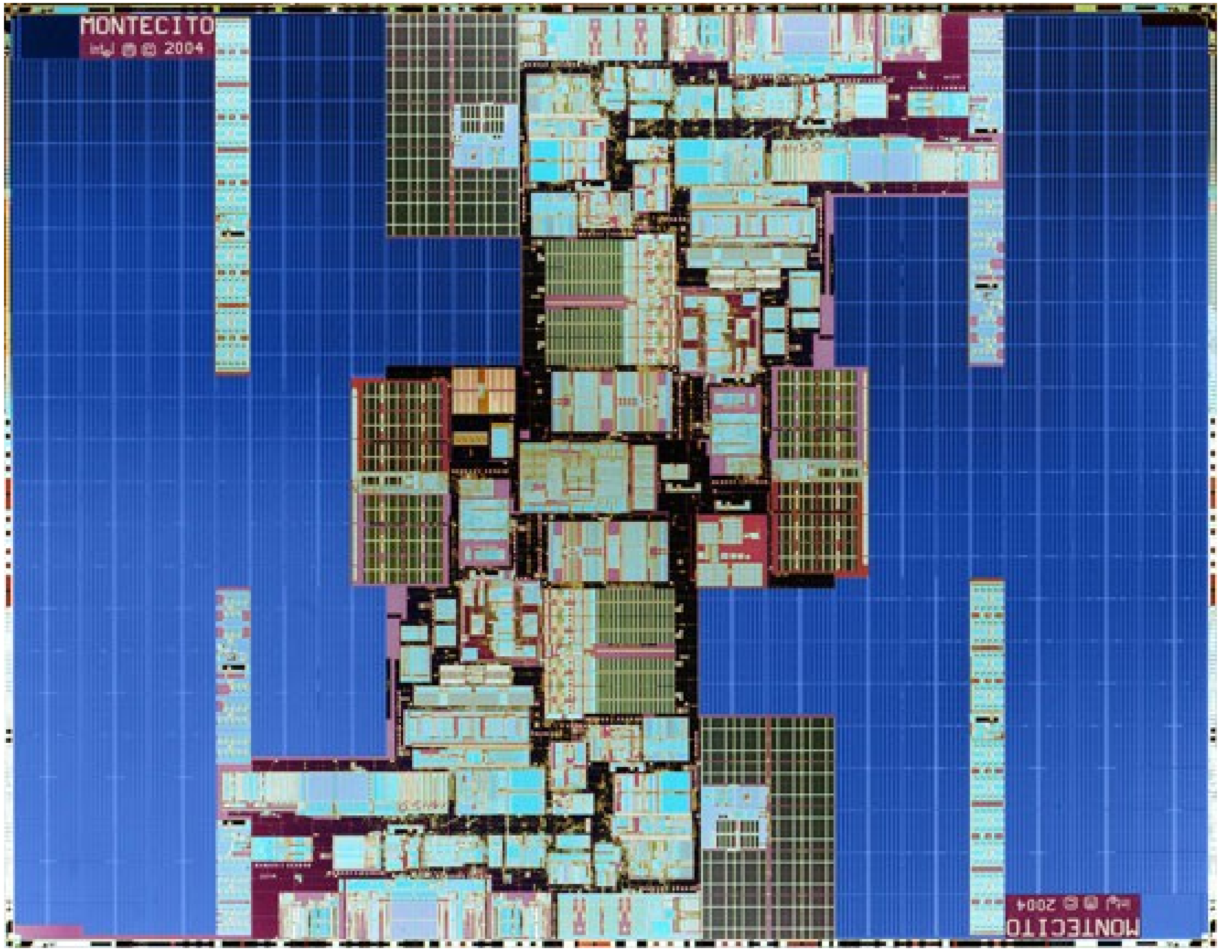
- It can introduce cache pollution – prefetched data is often placed in a separate prefetch buffer to avoid pollution – this buffer must be looked up in parallel with the cache access

- Aggressive prefetching increases “coverage”, but leads to a reduction in “accuracy” → wasted memory bandwidth

- Prefetches must be timely: they must be issued sufficiently in advance to hide the latency, but not too early (to avoid pollution and eviction before use)

Intel Montecito Cache

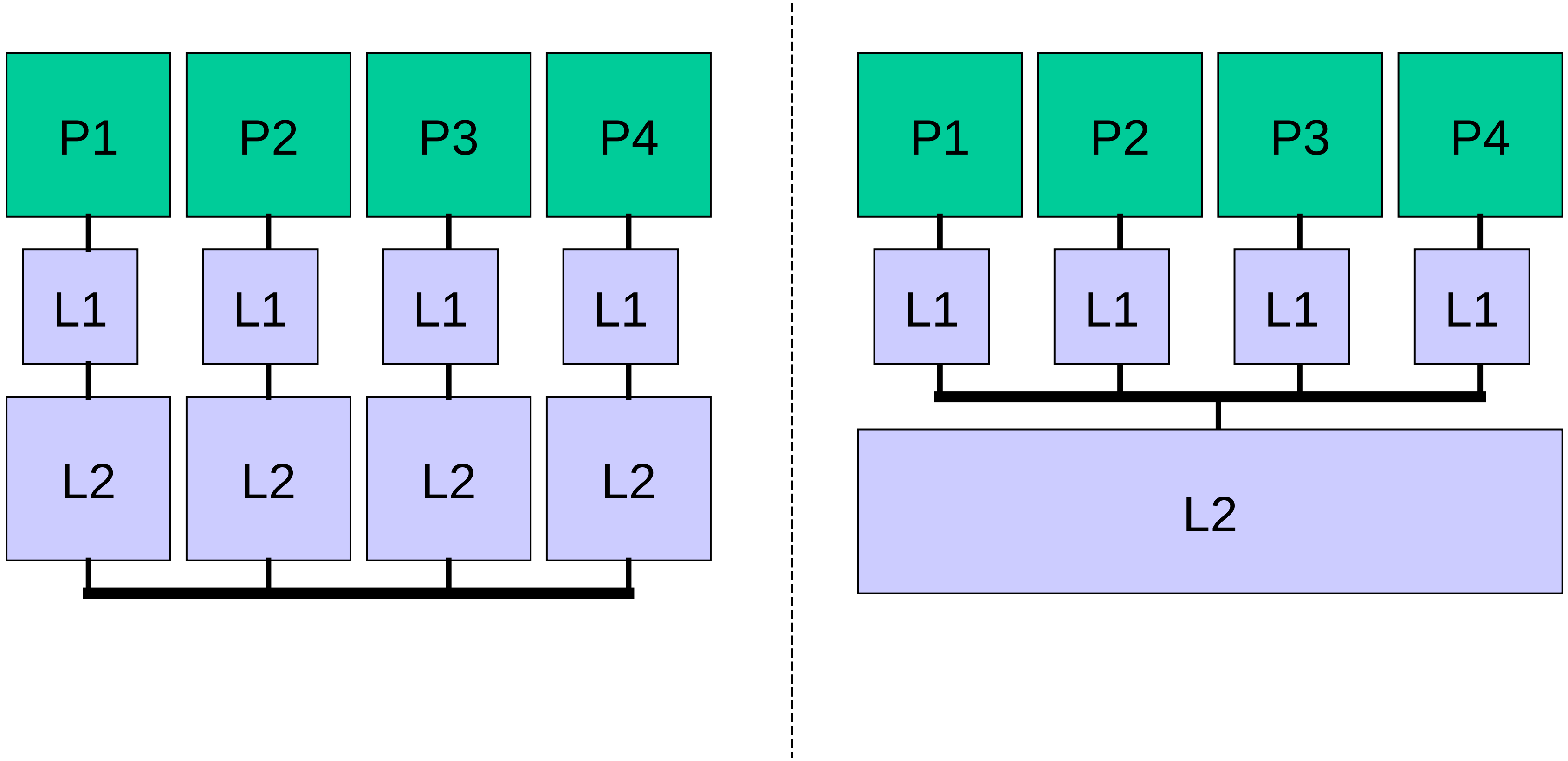
Two cores, each with a private 12 MB L3 cache and 1 MB L2



Naffziger et al., Journal of Solid-State Circuits, 2006

Shared Vs. Private Caches in Multi-Core

- What are the pros/cons to a shared L2 cache?



Shared Vs. Private Caches in Multi-Core

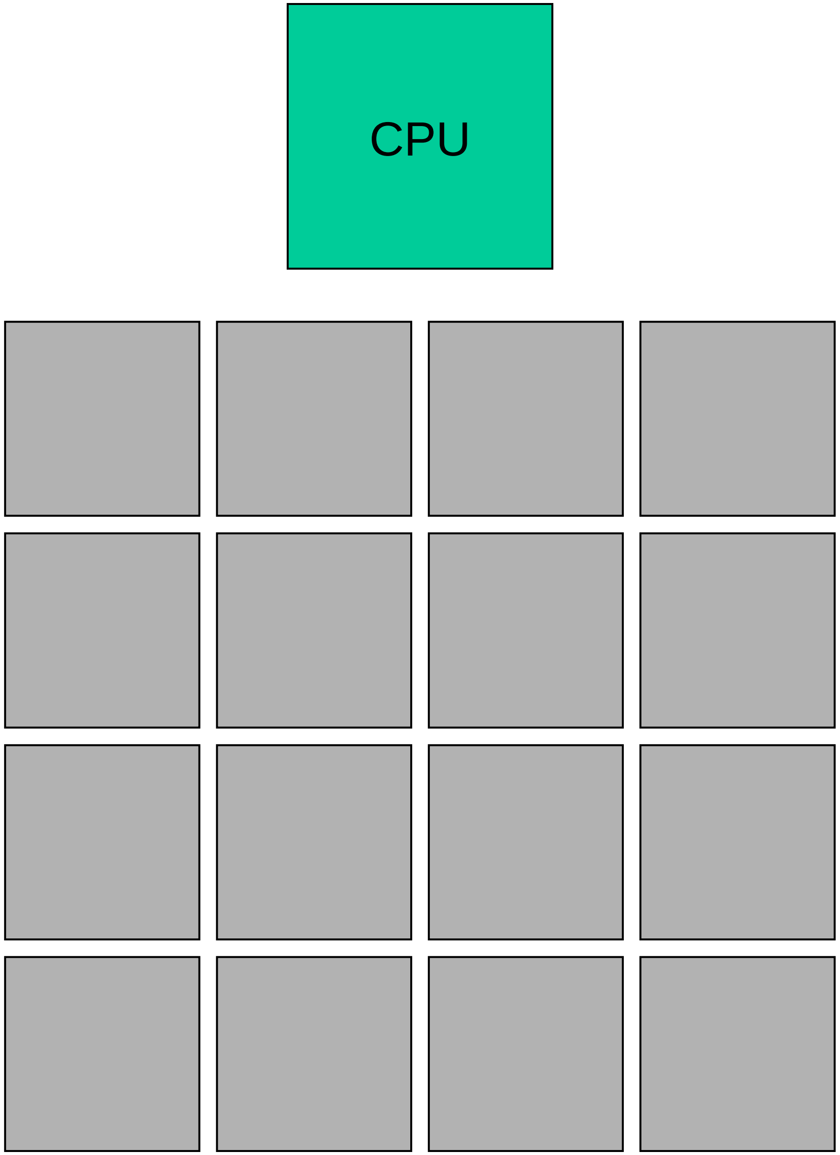
- Advantages of a shared cache:
  - Space is dynamically allocated among cores
  - No waste of space because of replication
  - Potentially faster cache coherence (and easier to locate data on a miss)

- Advantages of a private cache:
  - small L2 → faster access time
  - private bus to L2 → less contention

UCA and NUCA

- The small-sized caches so far have all been uniform cache access: the latency for any access is a constant, no matter where data is found
- For a large multi-megabyte cache, it is expensive to limit access time by the worst case delay: hence, non-uniform cache architecture

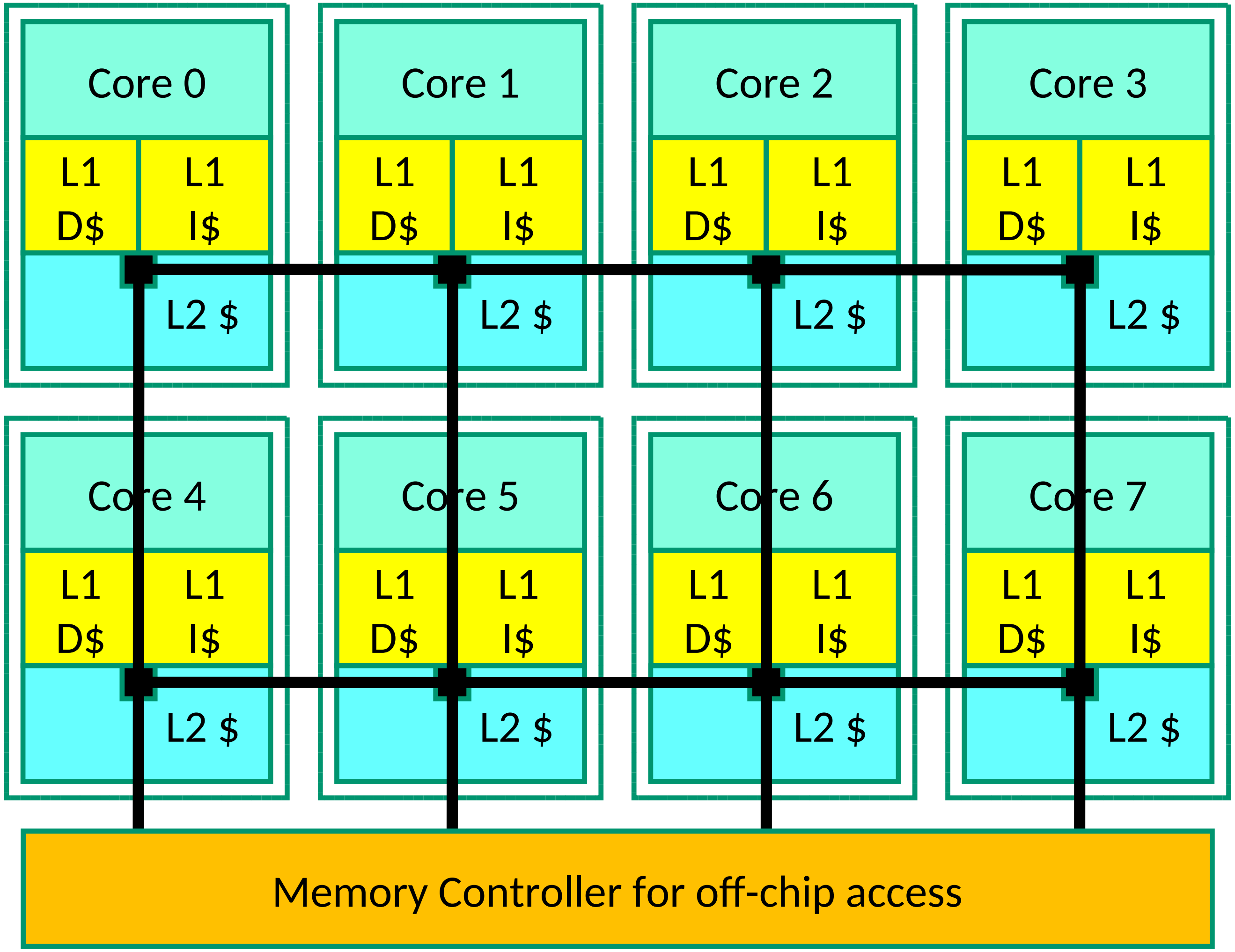
Large NUCA



Issues to be addressed for Non-Uniform Cache Access:

- Mapping
- Migration
- Search
- Replication

Shared NUCA Cache



A single tile composed of a core, L1 caches, and a bank (slice) of the shared L2 cache

The cache controller forwards address requests to the appropriate L2 bank and handles coherence operations

Virtual Memory

- Processes deal with virtual memory – they have the illusion that a very large address space is available to them

- There is only a limited amount of physical memory that is shared by all processes – a process places part of its virtual memory in this physical memory and the rest is stored on disk

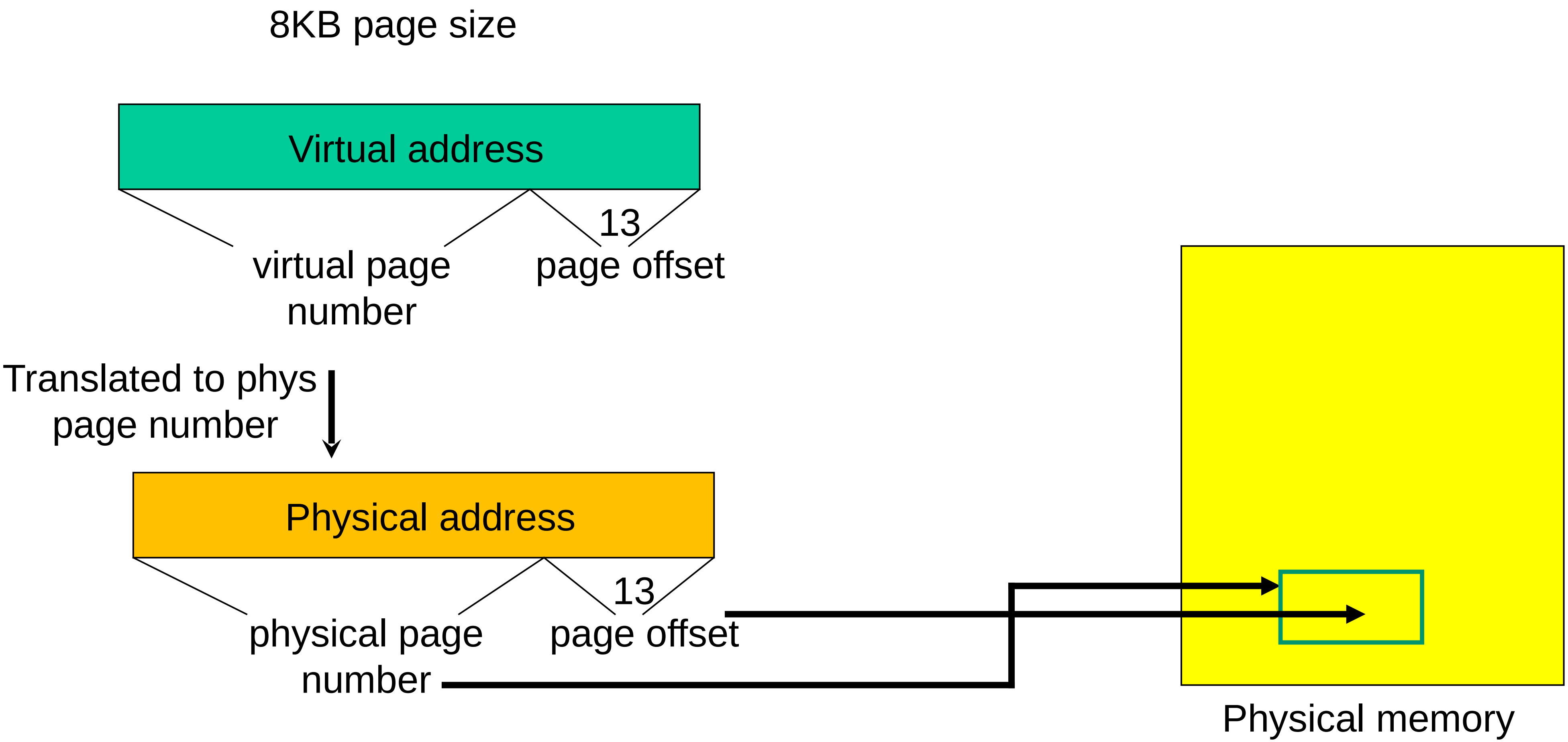
- Thanks to locality, disk access is likely to be uncommon

- The hardware ensures that one process cannot access the memory of a different process

Virtual Memory and Page Tables

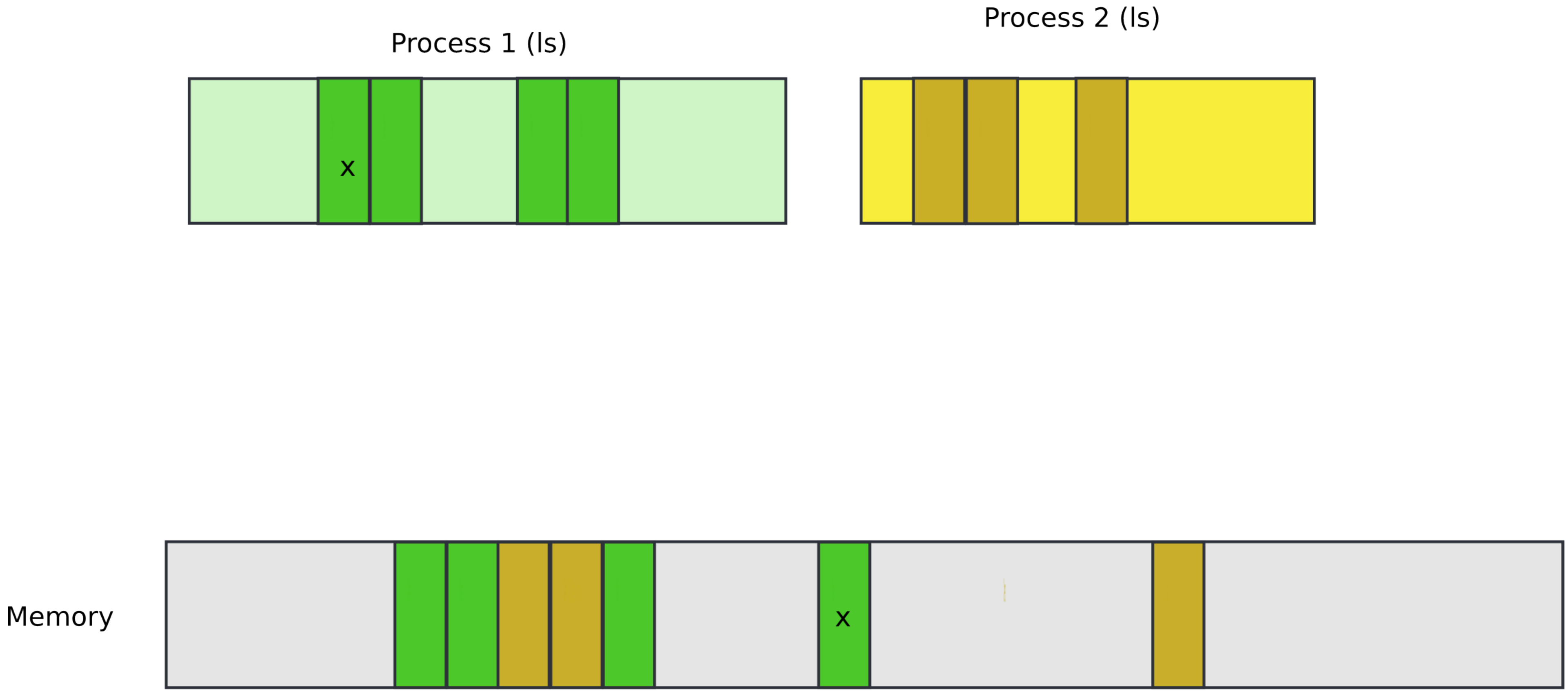
Address Translation

- The virtual and physical memory are broken up into pages

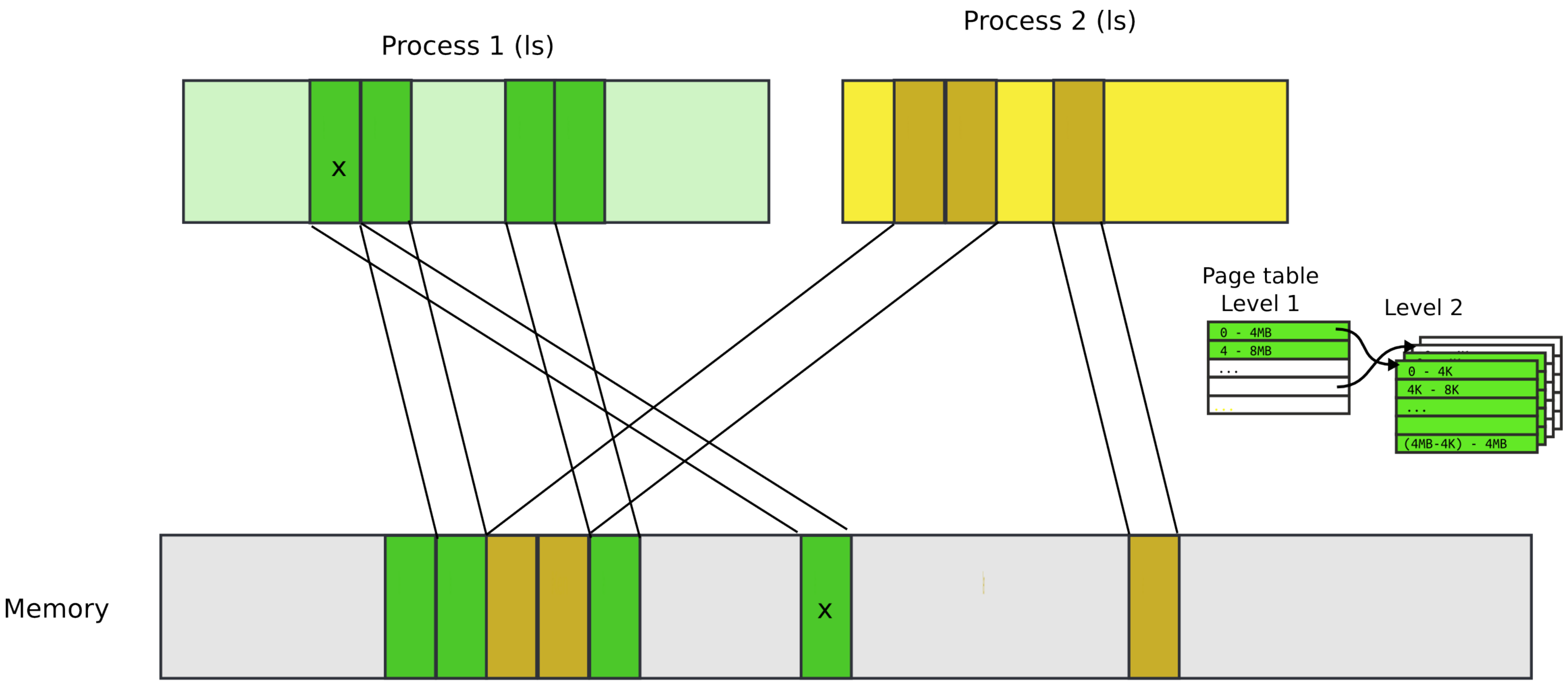


Paging

Pages



Pages

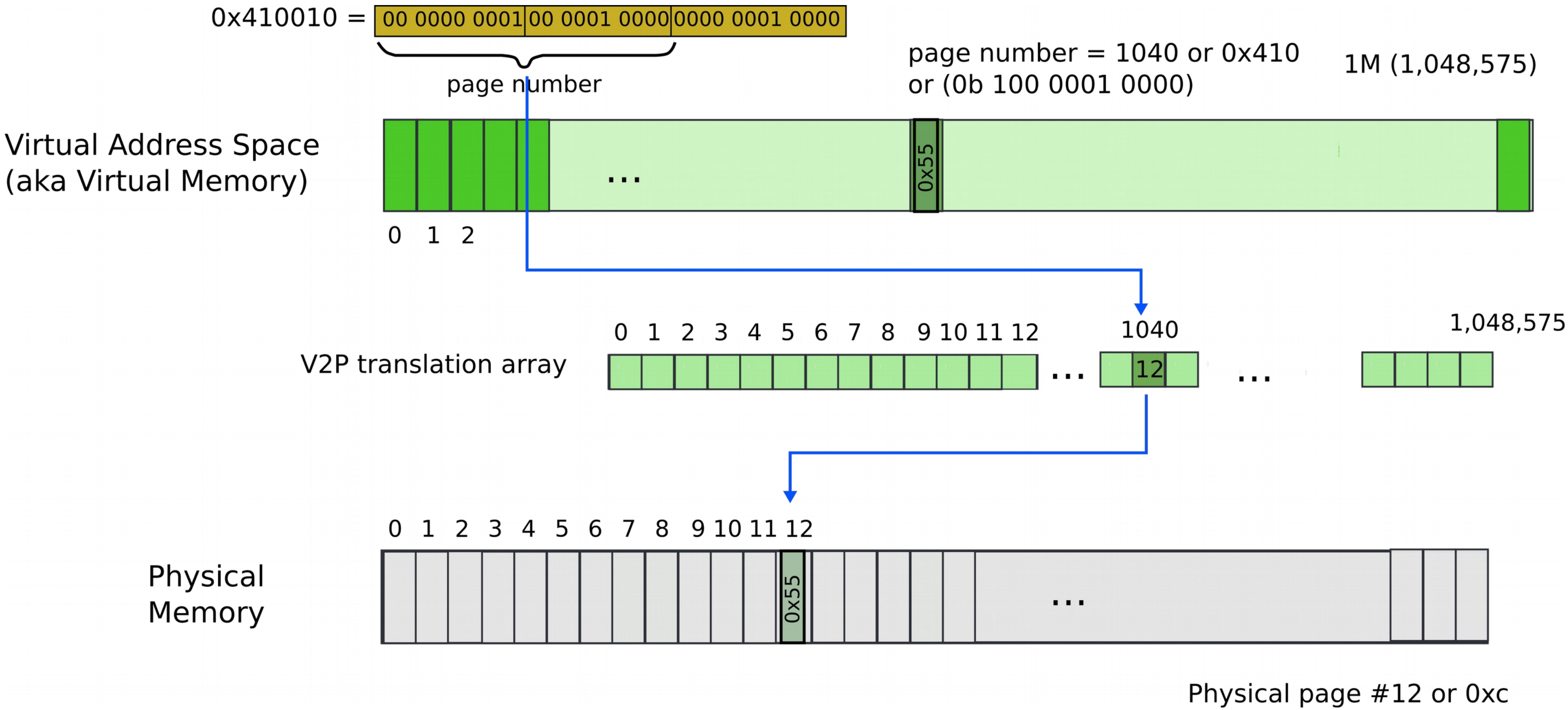

Process 1 (ls)
Process 2 (ls)
x
Page table
Level 1
0 - 4MB
4 - 8MB
...
...
Level 2
0 - 4K
4K - 8K
...
(4MB-4K) - 4MB
Memory
x

Paging idea

- Break up memory into 4096-byte chunks called pages
  - Modern hardware supports 2MB, 4MB, and 1GB pages
- Independently control mapping for each page of linear address space
- Compare with segmentation (single base + limit)
  - many more degrees of freedom

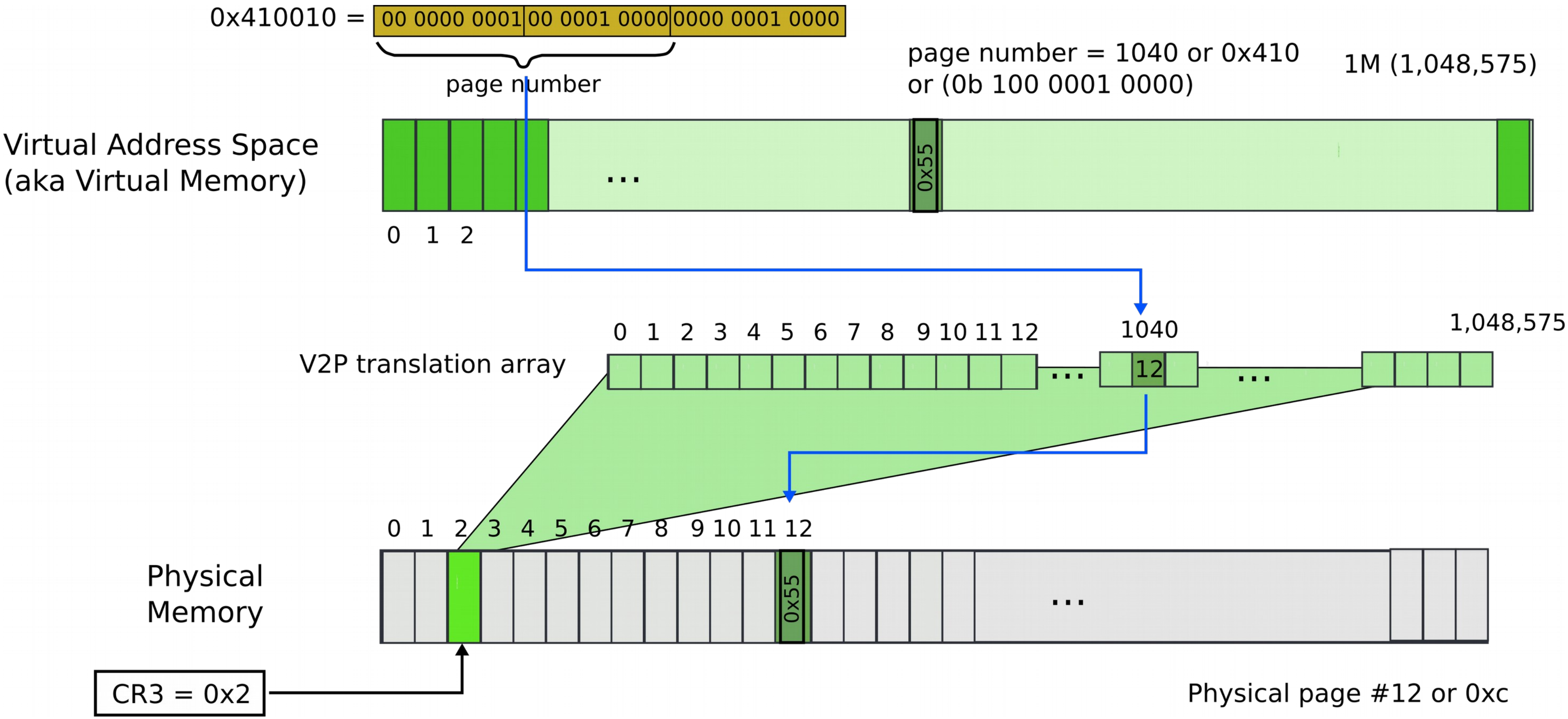
How can we build this translation mechanism?

Paging: naive approach: translation array



- Virtual address 0x410010

Paging: naive approach: translation array



- Virtual address 0x410010

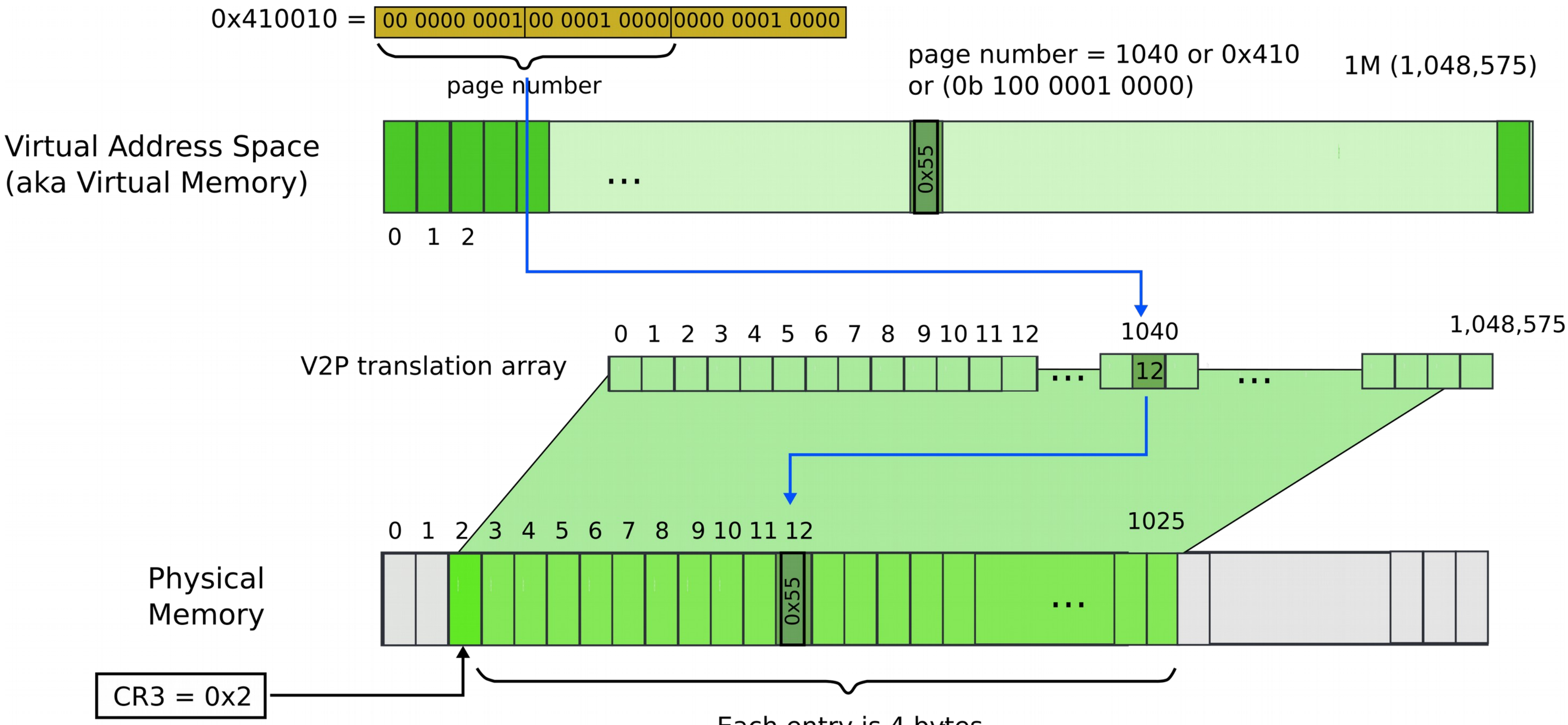
What is wrong?

What is wrong?

- We need 4 bytes to relocate each page
  - 20 bits for physical page number
  - 12 bits of access flags

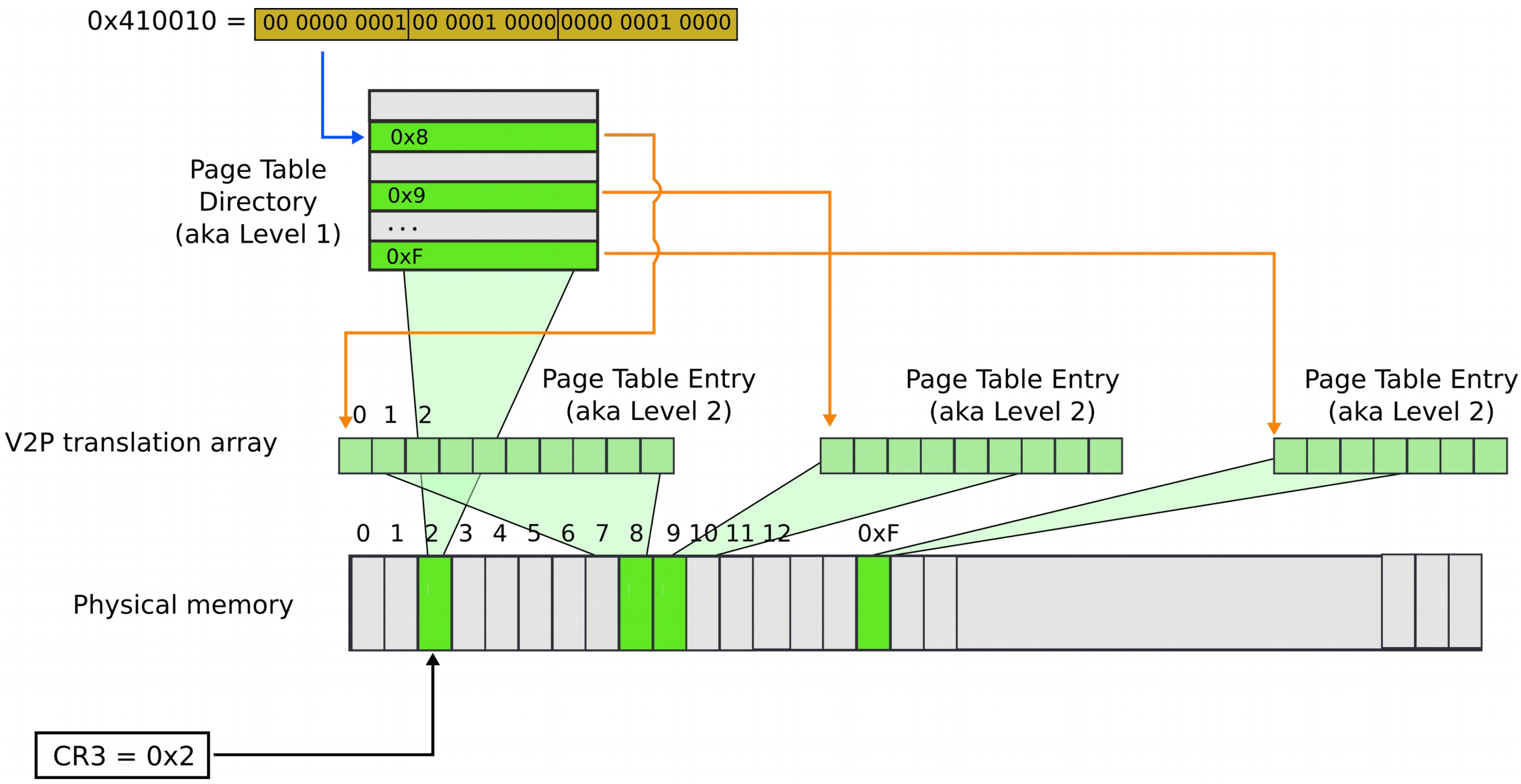
  - Therefore, we need array of 4 bytes x 1M entries
    - 4MBs

Paging: naive approach: translation array

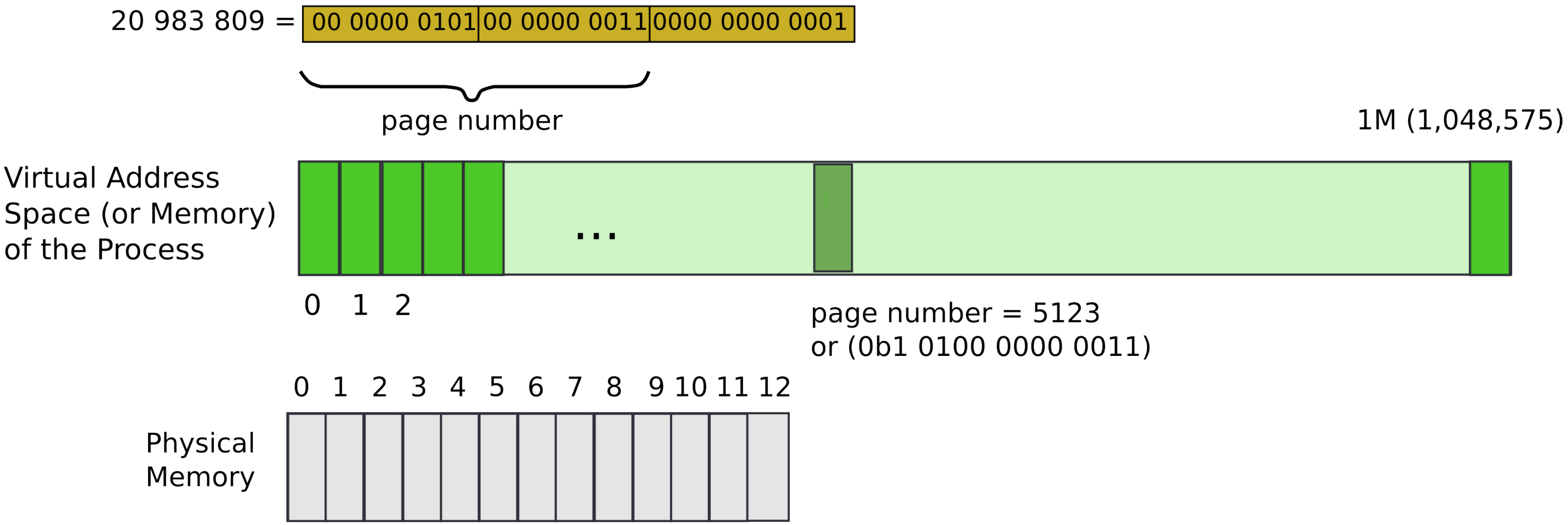


- Each entry is 4 bytes
  -- 20 bits to represent page number + access control bits
- 1 page can contain 1024 entries
- We need 1024 pages to represent all
  possible 1M translations

Paging: page table


0x410010 = 00 0000 0001 00 0001 0000 0000 0001 0000
Page Table Directory (aka Level 1)
0x8
0x9
. . .
0xF
Page Table Entry (aka Level 2)
Page Table Entry (aka Level 2)
Page Table Entry (aka Level 2)
V2P translation array
0 1 2
0 1 2 3 4 5 6 7 8 9 10 11 12 0xF
Physical memory
CR3 = 0x2

```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



20 983 809 = 00 0000 0101 | 00 0000 0011 | 0000 0000 0001

page number

1M (1,048,575)

Virtual Address Space (or Memory) of the Process

...

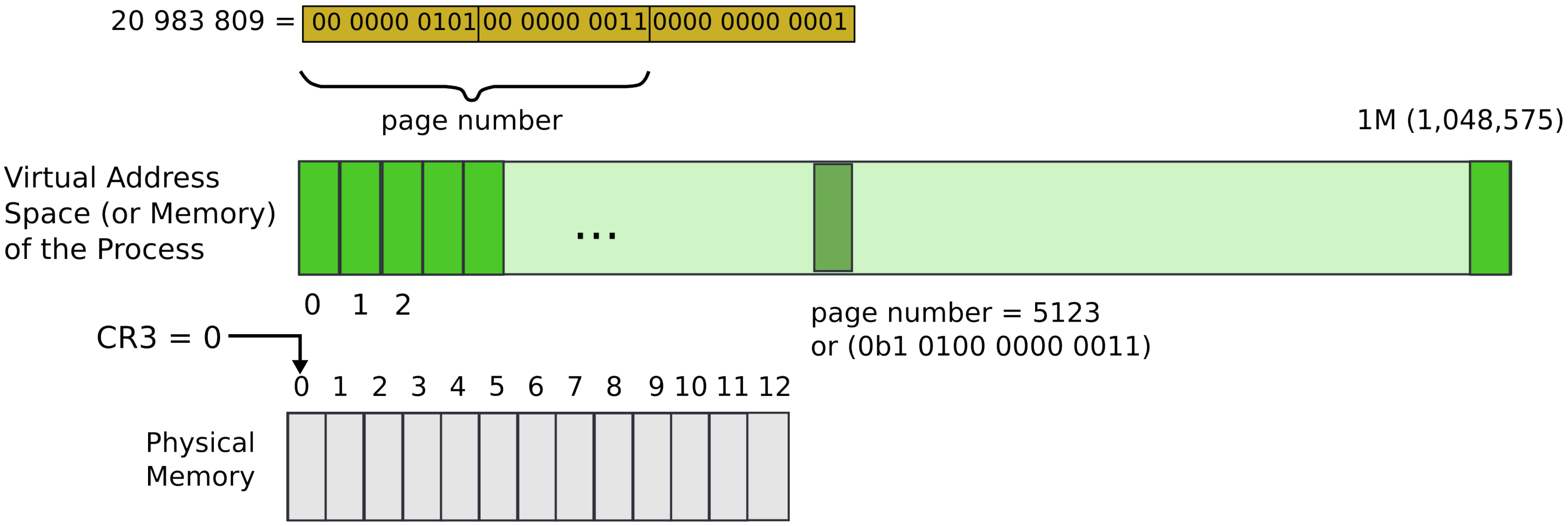
0 1 2

page number = 5123
or (0b1 0100 0000 0011)

0 1 2 3 4 5 6 7 8 9 10 11 12

Physical Memory

```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



20 983 809 = 00 0000 0101 | 00 0000 0011 | 0000 0000 0001

page number

1M (1,048,575)

Virtual Address Space (or Memory) of the Process

...

0 1 2

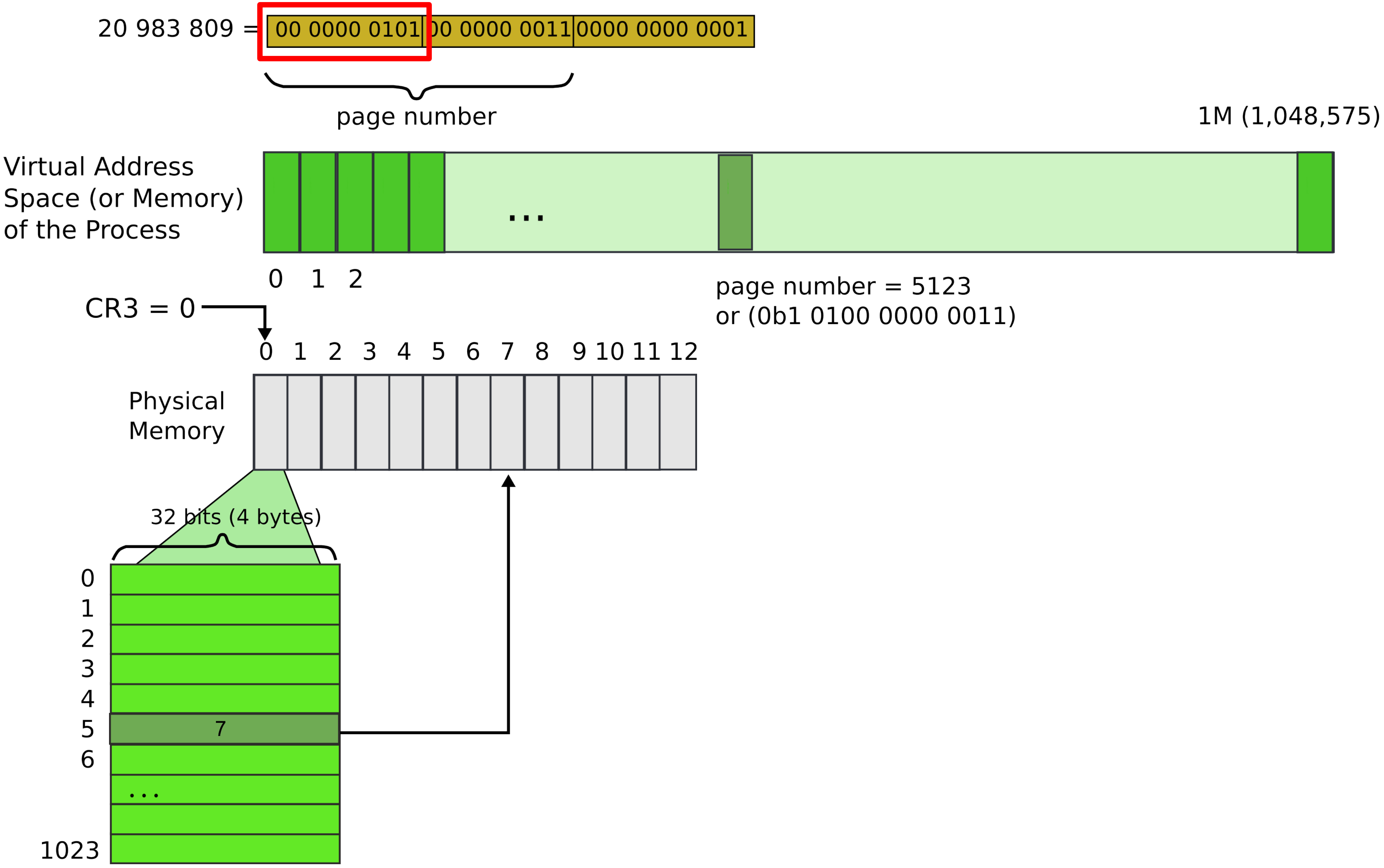
page number = 5123
or (0b1 0100 0000 0011)

CR3 = 0

0 1 2 3 4 5 6 7 8 9 10 11 12

Physical Memory

```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



20 983 809 = 00 0000 0101 | 00 0000 0011 | 0000 0000 0001

page number

1M (1,048,575)

Virtual Address Space (or Memory) of the Process

...

0 1 2

page number = 5123
or (0b1 0100 0000 0011)

CR3 = 0

0 1 2 3 4 5 6 7 8 9 10 11 12

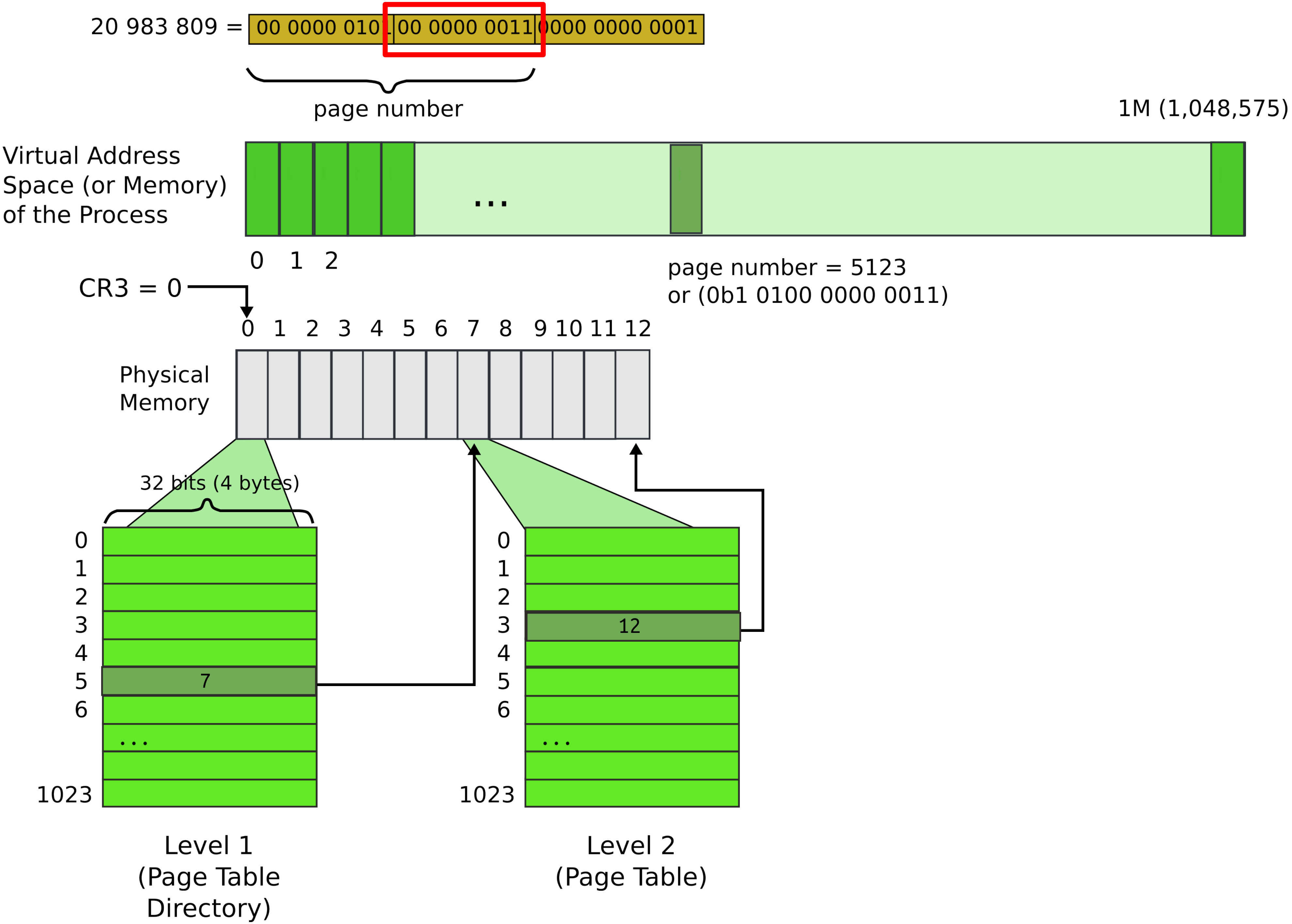
Physical Memory

32 bits (4 bytes)

0
1
2
3
4
5 7
6
...
1023

Level 1
(Page Table
Directory)

```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



20 983 809 = 00 0000 0101 | 00 0000 0011 | 0000 0000 0001

page number

1M (1,048,575)

Virtual Address Space (or Memory) of the Process

...

0 1 2

page number = 5123
or (0b1 0100 0000 0011)

CR3 = 0

0 1 2 3 4 5 6 7 8 9 10 11 12

Physical Memory

32 bits (4 bytes)

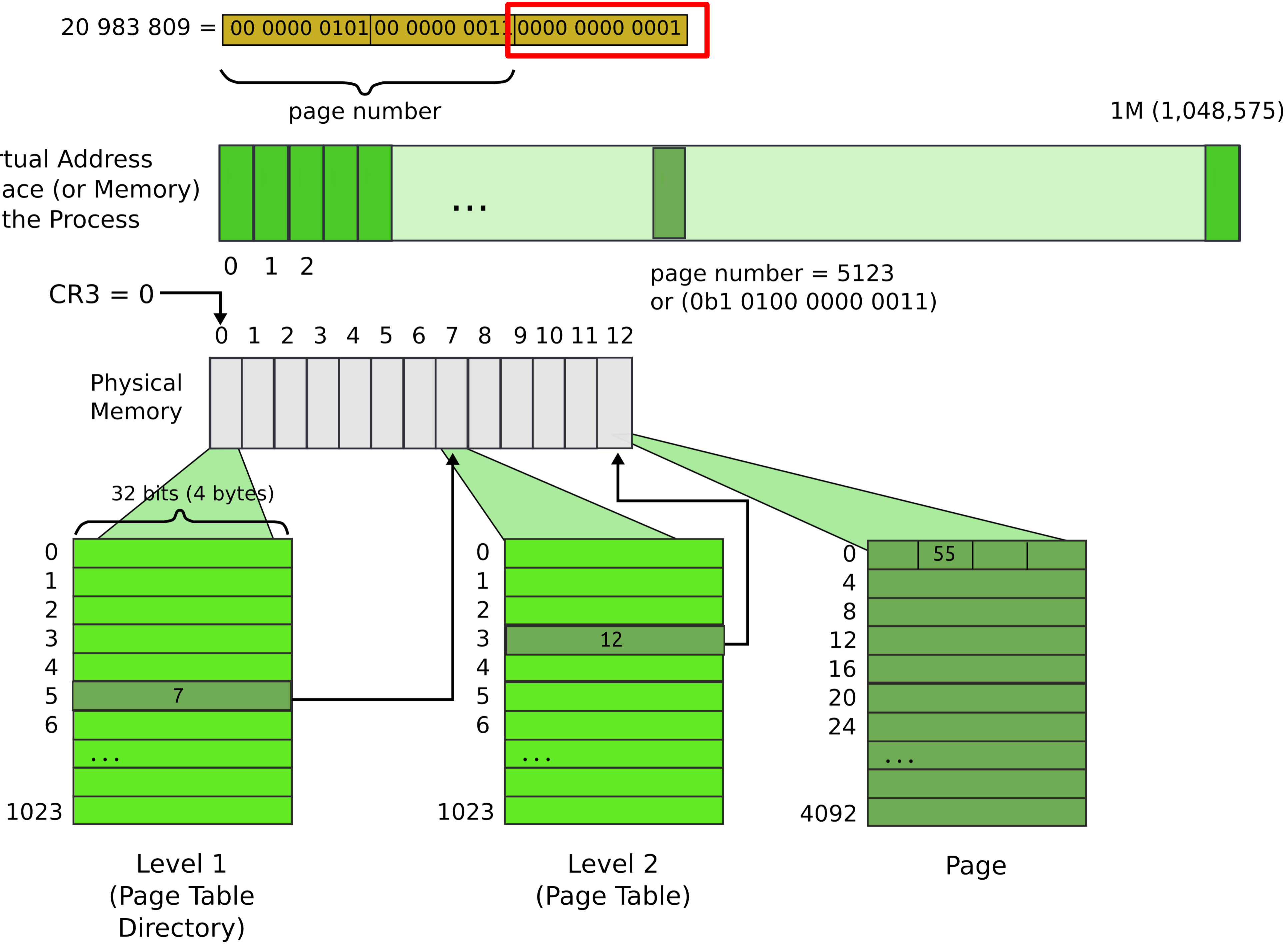
0
1
2
3
4
5 7
6
...
1023

Level 1
(Page Table Directory)

0
1
2
3 12
4
5
6
...
1023

Level 2
(Page Table)

```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```



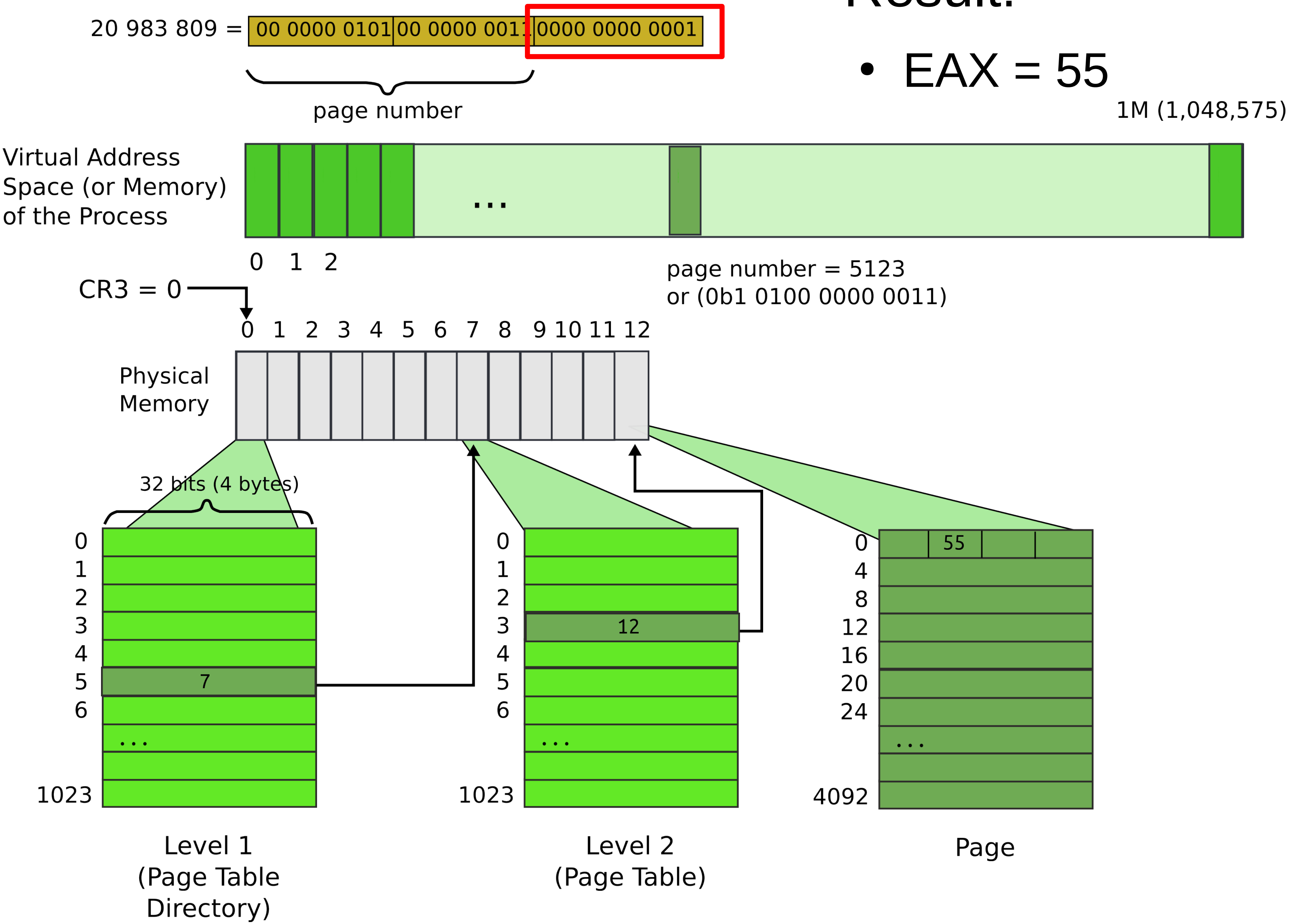
20 983 809 = 00 0000 0101|00 0000 0011|0000 0000 0001

page number = 5123
or (0b1 0100 0000 0011)

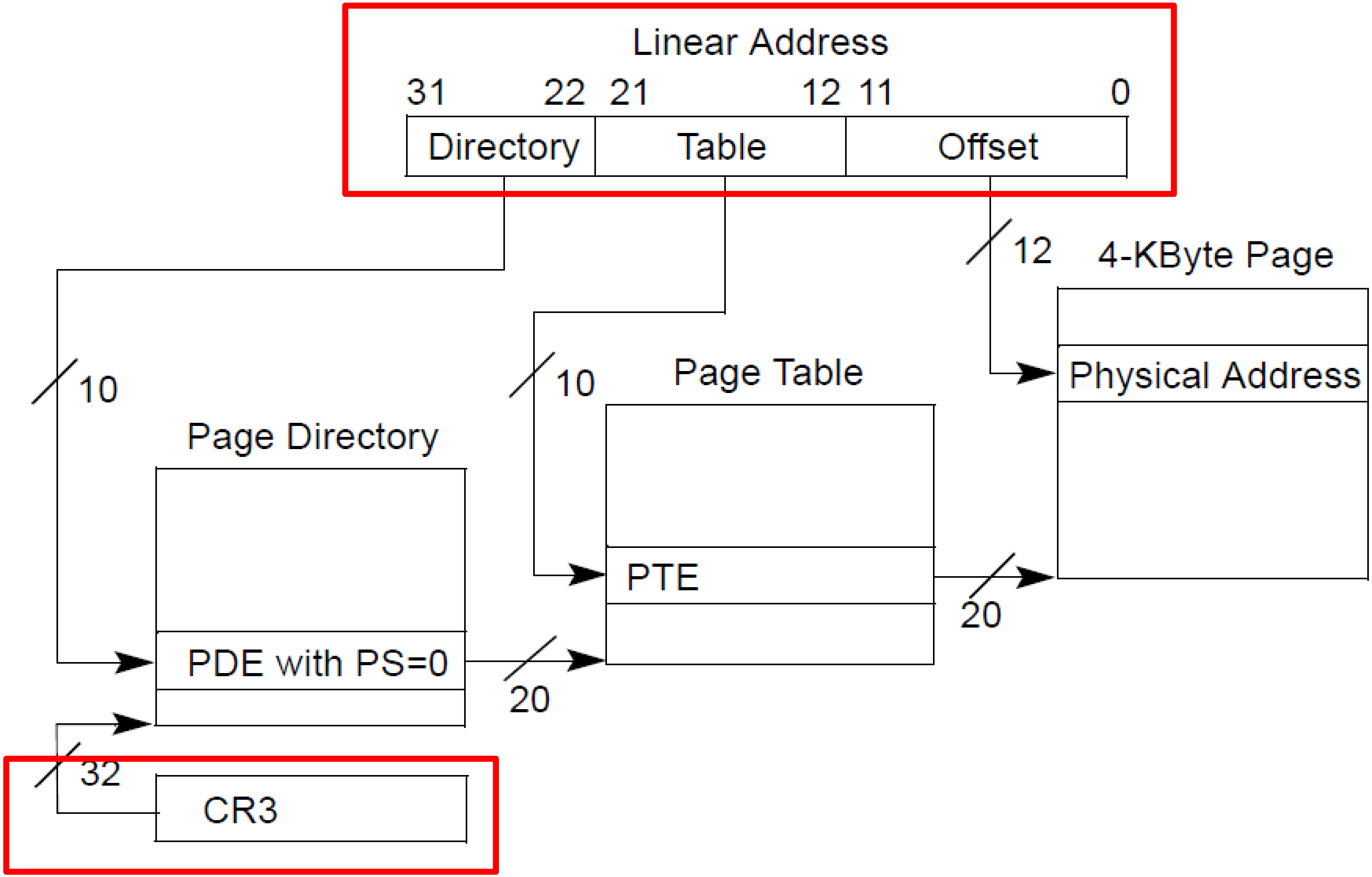
CR3 = 0

```
mov (%EBX), EAX   # mov value from the  location pointed by EBX into EAX
EAX = 0
EBX = 20 983 809
```

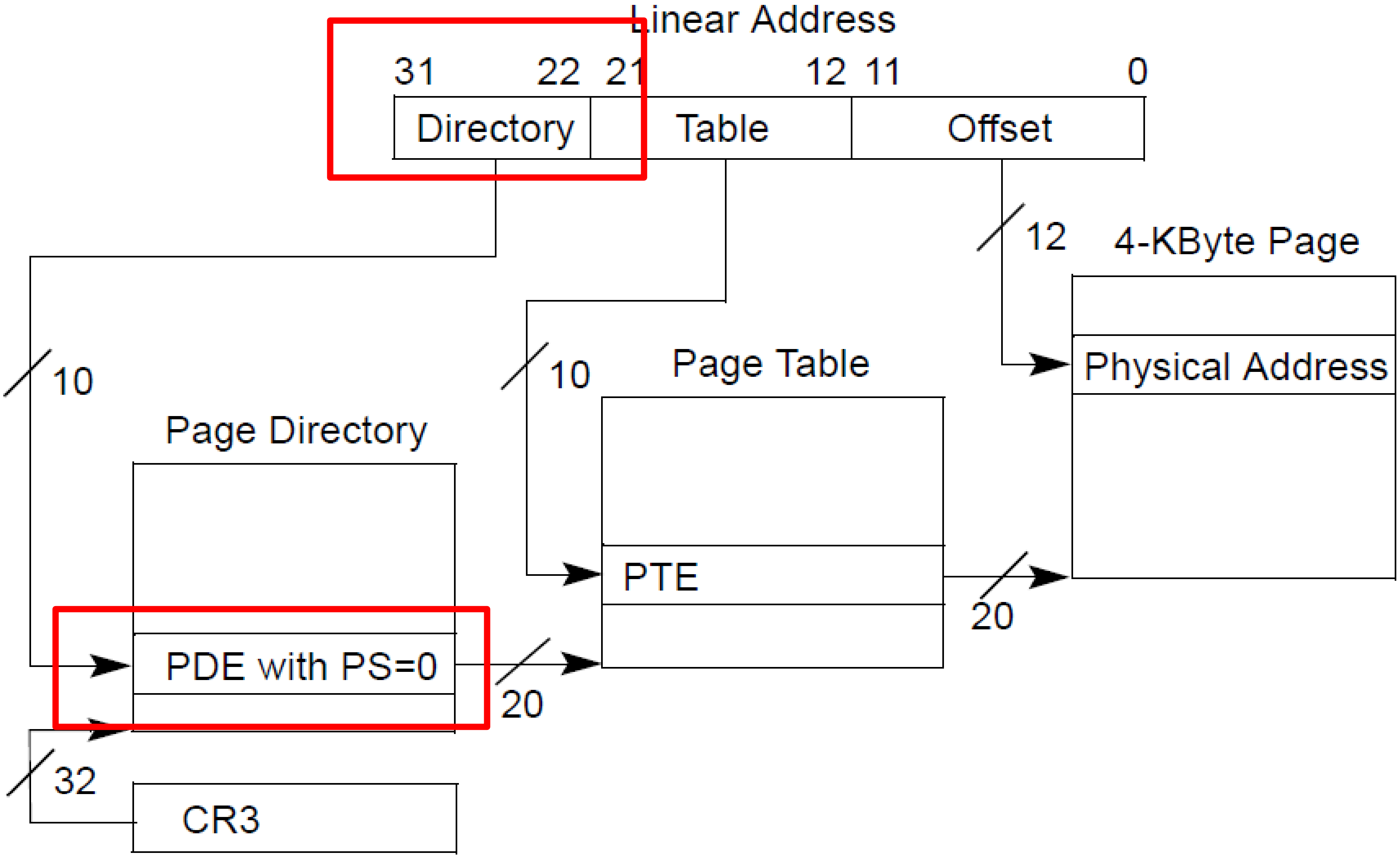
- Result:
  - EAX = 55



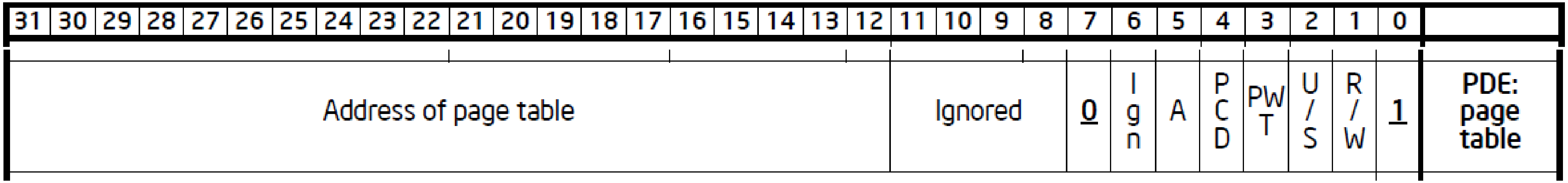
Page translation



Page translation



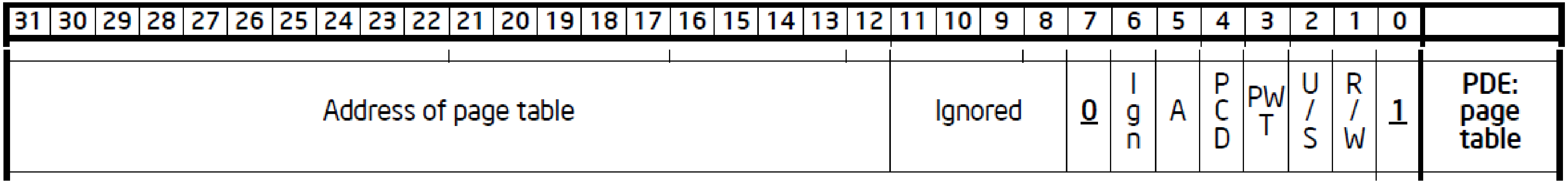
Page directory entry (PDE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**

- 20 bit address of the page table

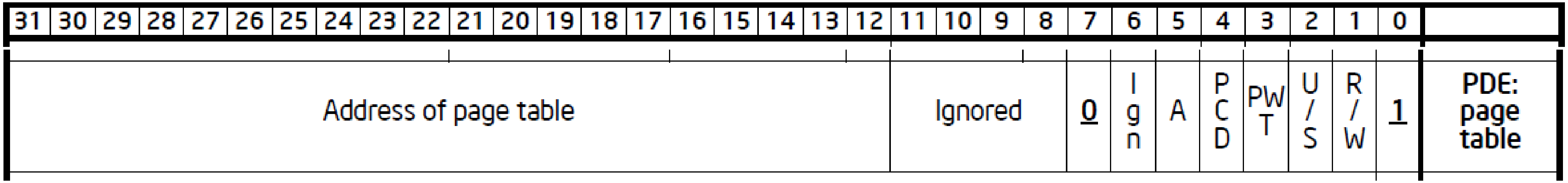
Page directory entry (PDE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				**0**	Ign	A	PCD	PWT	U/S	R/W	**1**	**PDE: page table**

- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits

Page directory entry (PDE)

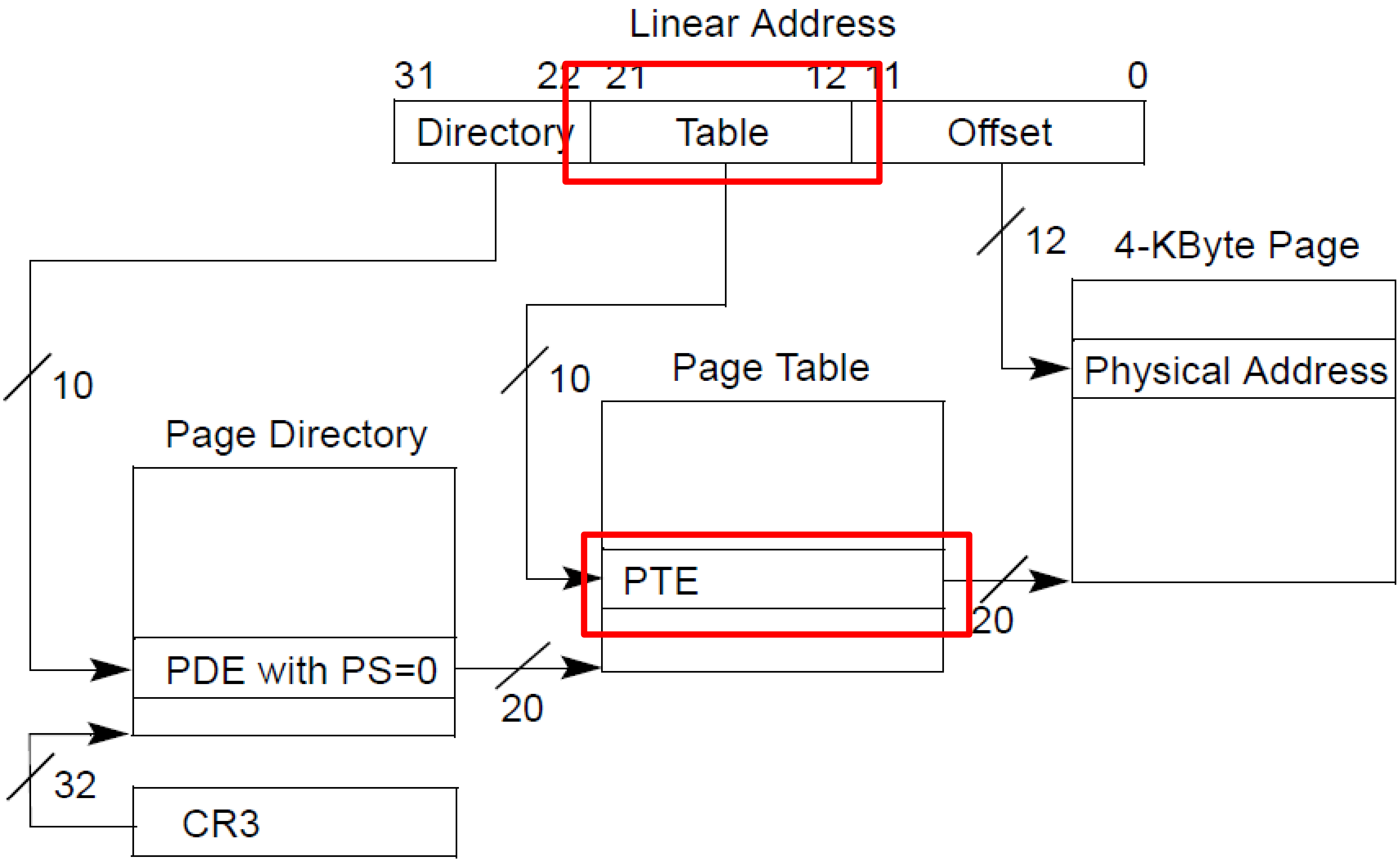


31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of page table																				Ignored				0	Ign	A	PCD	PWT	U/S	R/W	1	**PDE: page table**

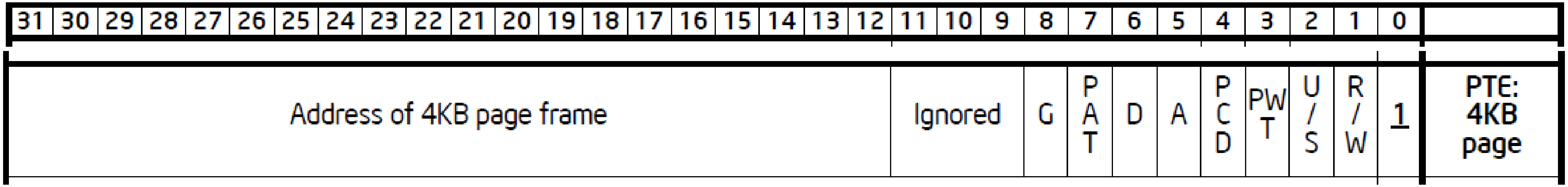
- 20 bit address of the page table
- Wait... 20 bit address, but we need 32 bits

  - Pages 4KB each, we need 1M to cover 4GB
  - Pages start at 4KB (page aligned boundary)

Page translation



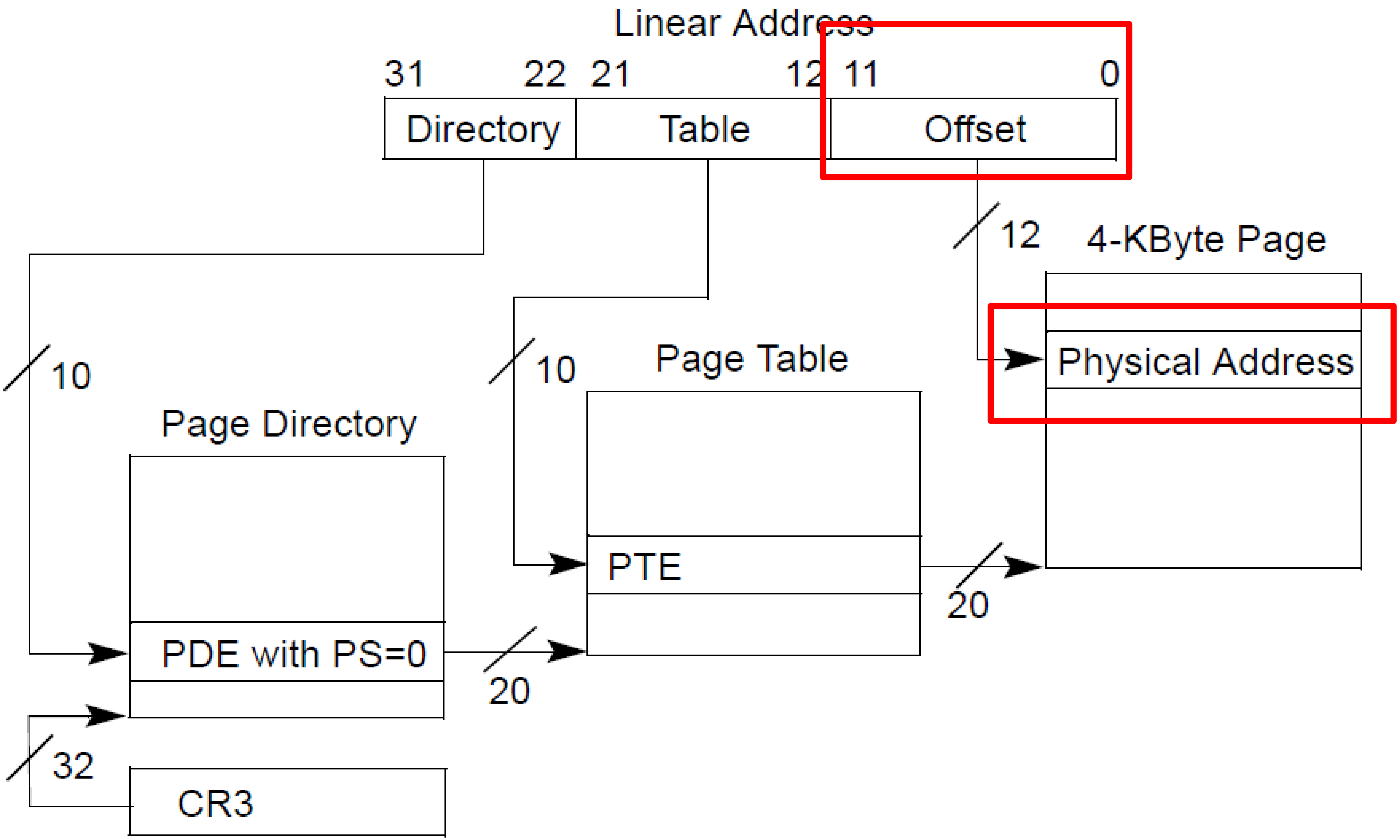
Page table entry (PTE)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Address of 4KB page frame																				Ignored			G	PAT	D	A	PCD	PWT	U/S	R/W	1	**PTE: 4KB page**

- 20 bit address of the 4KB page
  - Pages 4KB each, we need 1M to cover 4GB

Page translation

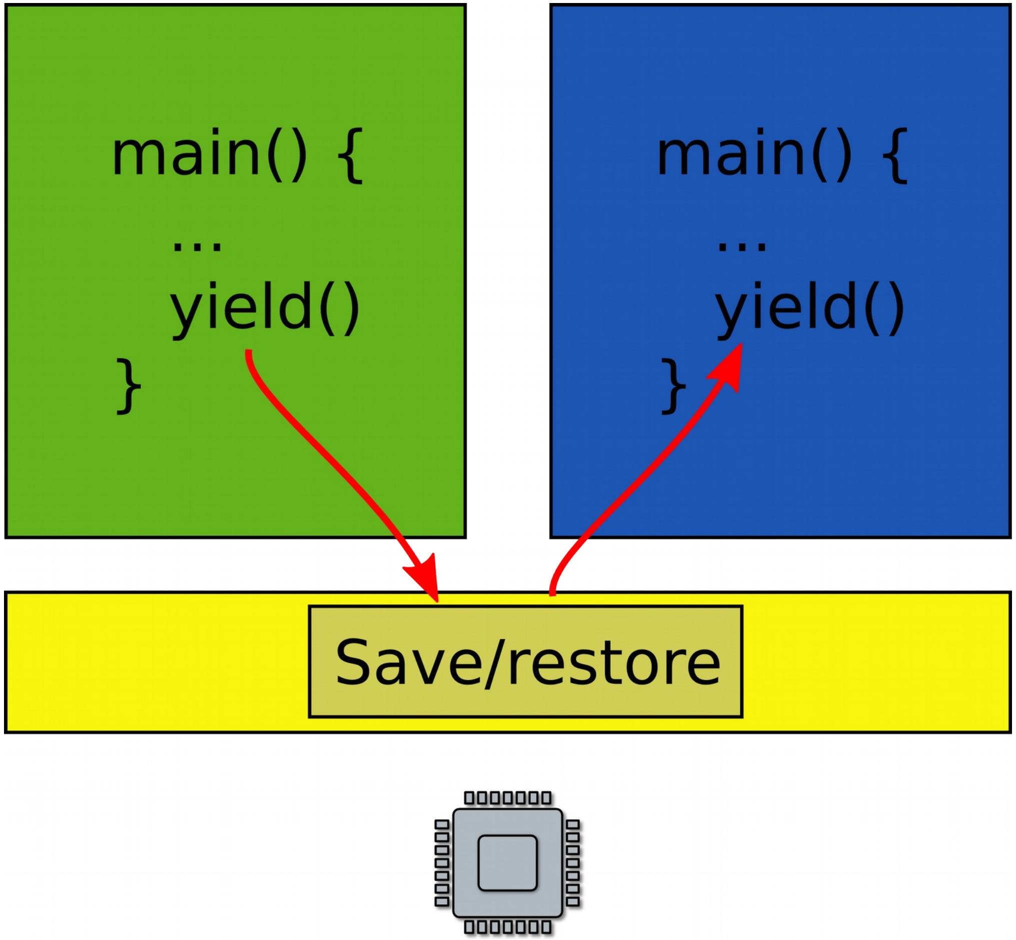


Benefit of page tables

… Compared to arrays?

- Page tables represent sparse address space more efficiently
  - An entire array has to be allocated upfront
  - But if the address space uses a handful of pages
  - Only page tables (Level 1 and 2 need to be allocated to describe translation)
- On a dense address space this benefit goes away
  - I'll assign a homework!

What about isolation?

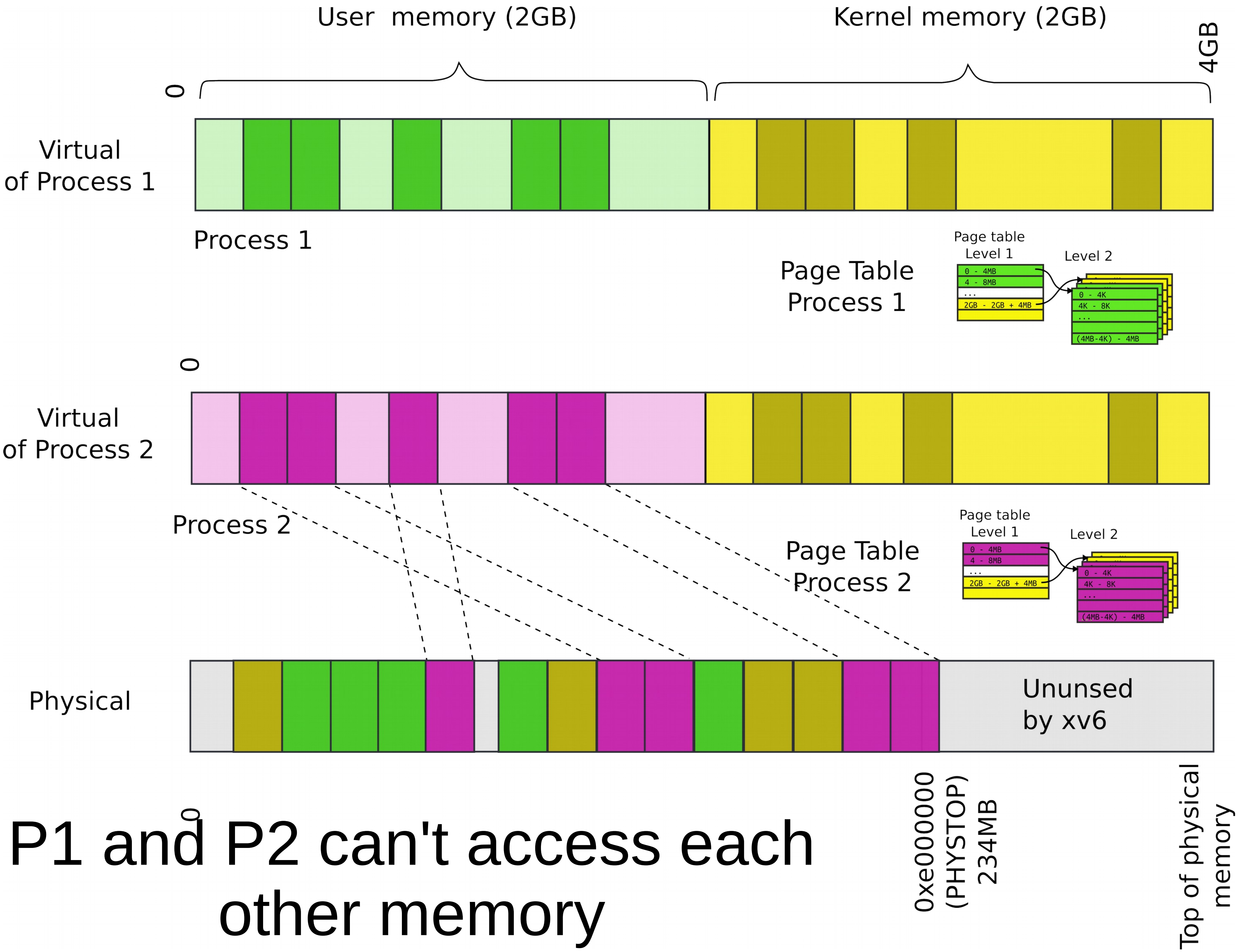


```
main() {
    ...
    yield()
}
```

```
main() {
    ...
    yield()
}
```

Save/restore

- Two programs, one memory?
- Each process has its own page table
  - OS switches between them



P1 and P2 can't access each other memory

TLB

- Since the number of pages is very high, the page table capacity is too large to fit on chip

- A translation lookaside buffer (TLB) caches the virtual to physical page number translation for recent accesses

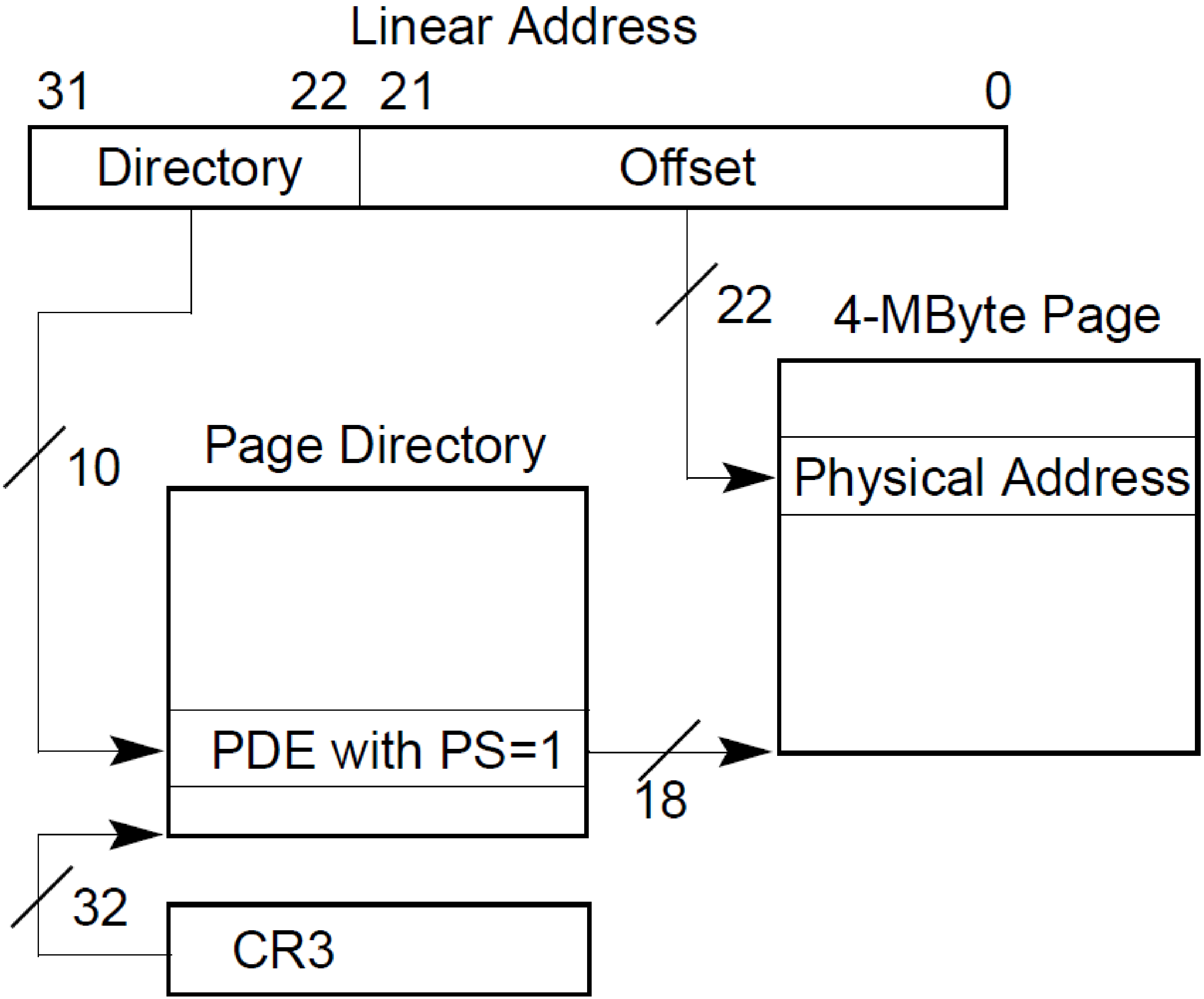
- A TLB miss requires us to access the page table, which may not even be found in the cache – two expensive memory look-ups to access one word of data!

- A large page size can increase the coverage of the TLB and reduce the capacity of the page table, but also increases memory waste

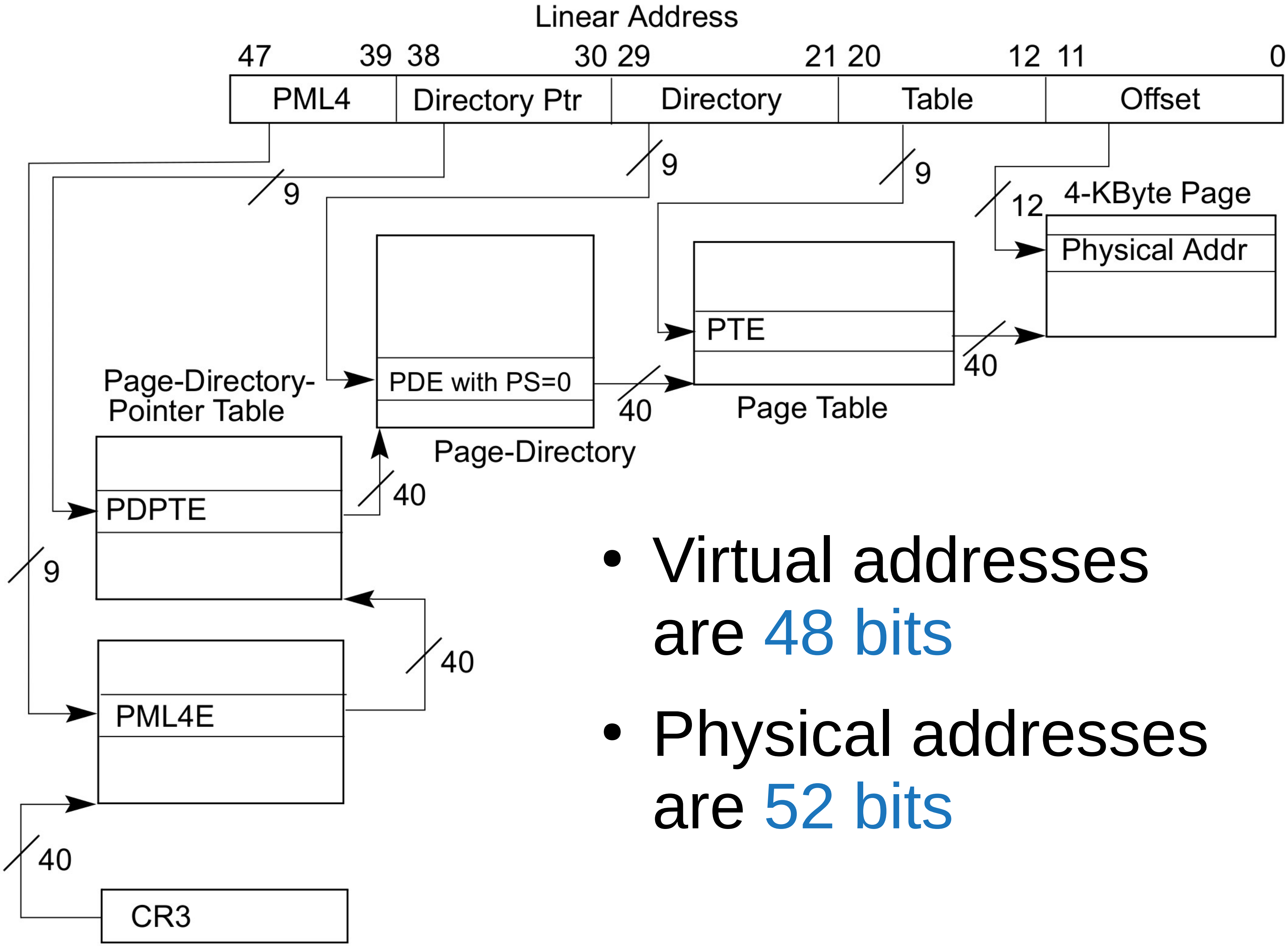
32bit x86 supports two page sizes

- 4KB pages
- 4MB pages

Page translation for 4MB pages



Page translation in 64bit mode



- Virtual addresses are 48 bits
- Physical addresses are 52 bits

Memory Hierarchy Properties

- A virtual memory page can be placed anywhere in physical memory (fully-associative)

- Replacement is usually LRU (since the miss penalty is huge, we can invest some effort to minimize misses)

- A page table (indexed by virtual page number) is used for translating virtual to physical page number

- The memory-disk hierarchy can be either inclusive or exclusive and the write policy is writeback

TLB and Cache

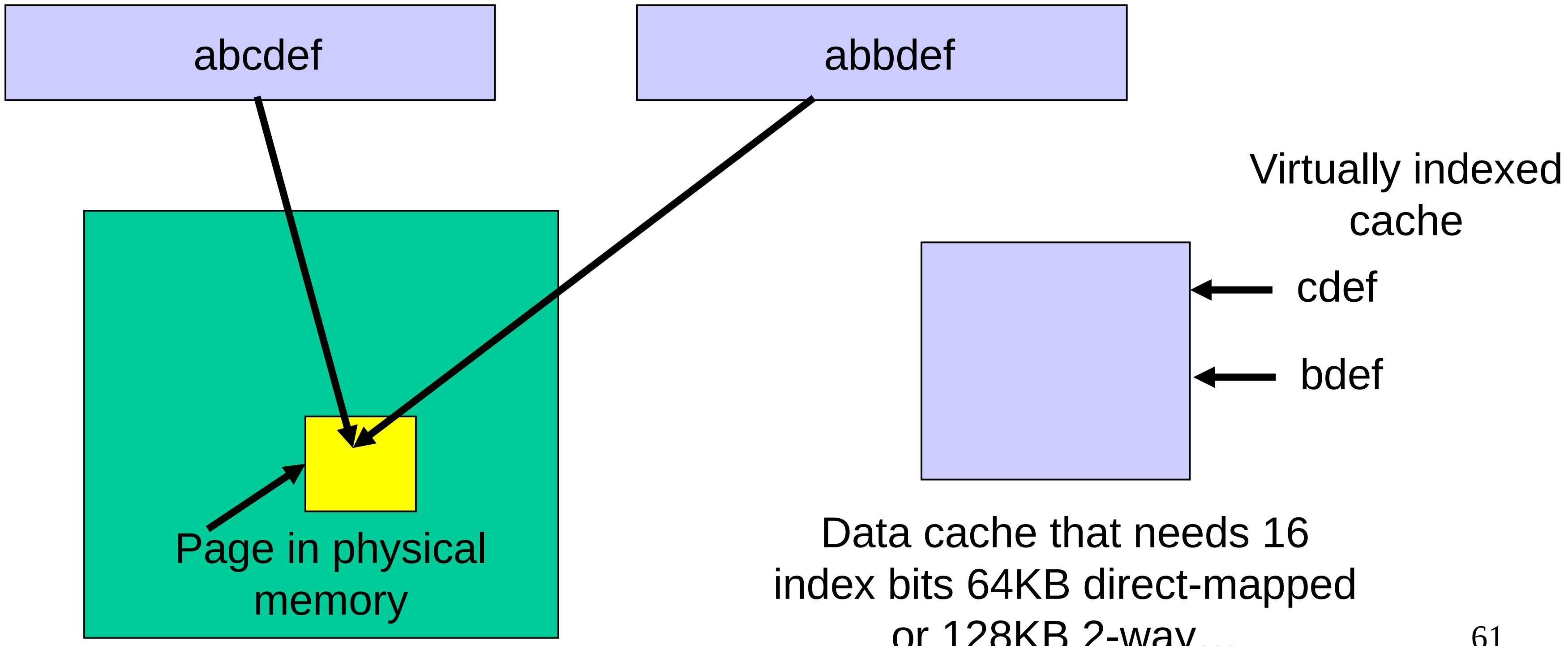
- Is the cache indexed with virtual or physical address?
  - To index with a physical address, we will have to first look up the TLB, then the cache → longer access time
  - Multiple virtual addresses can map to the same physical address – can we ensure that these different virtual addresses will map to the same location in cache? Else, there will be two different copies of the same physical memory word

- Does the tag array store virtual or physical addresses?
  - Since multiple virtual addresses can map to the same physical address, a virtual tag comparison can flag a miss even if the correct physical memory word is present

TLB and Cache

Virtually Indexed Caches

- 24-bit virtual address, 4KB page size → 12 bits offset and 12 bits virtual page number
- To handle the example below, the cache must be designed to use only 12 index bits – for example, make the 64KB cache 16-way
- Page coloring can ensure that some bits of virtual and physical address match



Data cache that needs 16 index bits 64KB direct-mapped or 128KB 2-way…

Thank you!

Problem 1

- Memory access time: Assume a program that has cache access times of 1-cyc (L1), 10-cyc (L2), 30-cyc (L3), and 300-cyc (memory), and MPKIs of 20 (L1), 10 (L2), and 5 (L3). Should you get rid of the L3?

Problem 1

- Memory access time: Assume a program that has cache access times of 1-cyc (L1), 10-cyc (L2), 30-cyc (L3), and 300-cyc (memory), and MPKIs of 20 (L1), 10 (L2), and 5 (L3). Should you get rid of the L3?

With L3: 1000 + 10x20 + 30x10 + 300x5 = 3000
Without L3: 1000 + 10x20 + 10x300 = 4200