# CSE251: System Programming

## 15. Virtual Memory (1)

Seongil Wi

# HW4: Cache

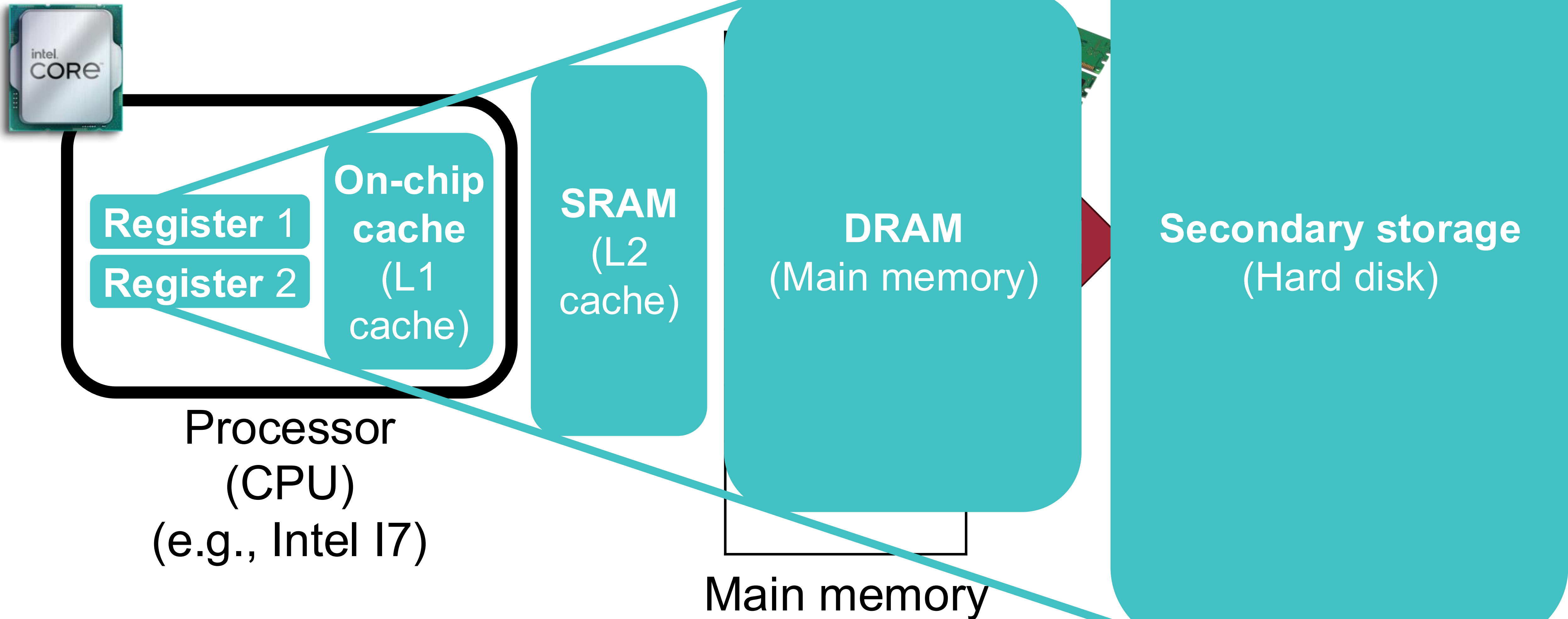
1. Write a small C program (about 200–300 lines) that simulates the behavior of a cache memory
2. Optimize a matrix transpose function with the goal of minimizing the number of cache misses
3. Answer the given questions

- Due: May. 18, 11:59PM
- Responsible TA: Jongyoung Kang
- If you have any requests or questions, please ask via Blackboard Q&A session

# Information Session: Information Security Micro Minor

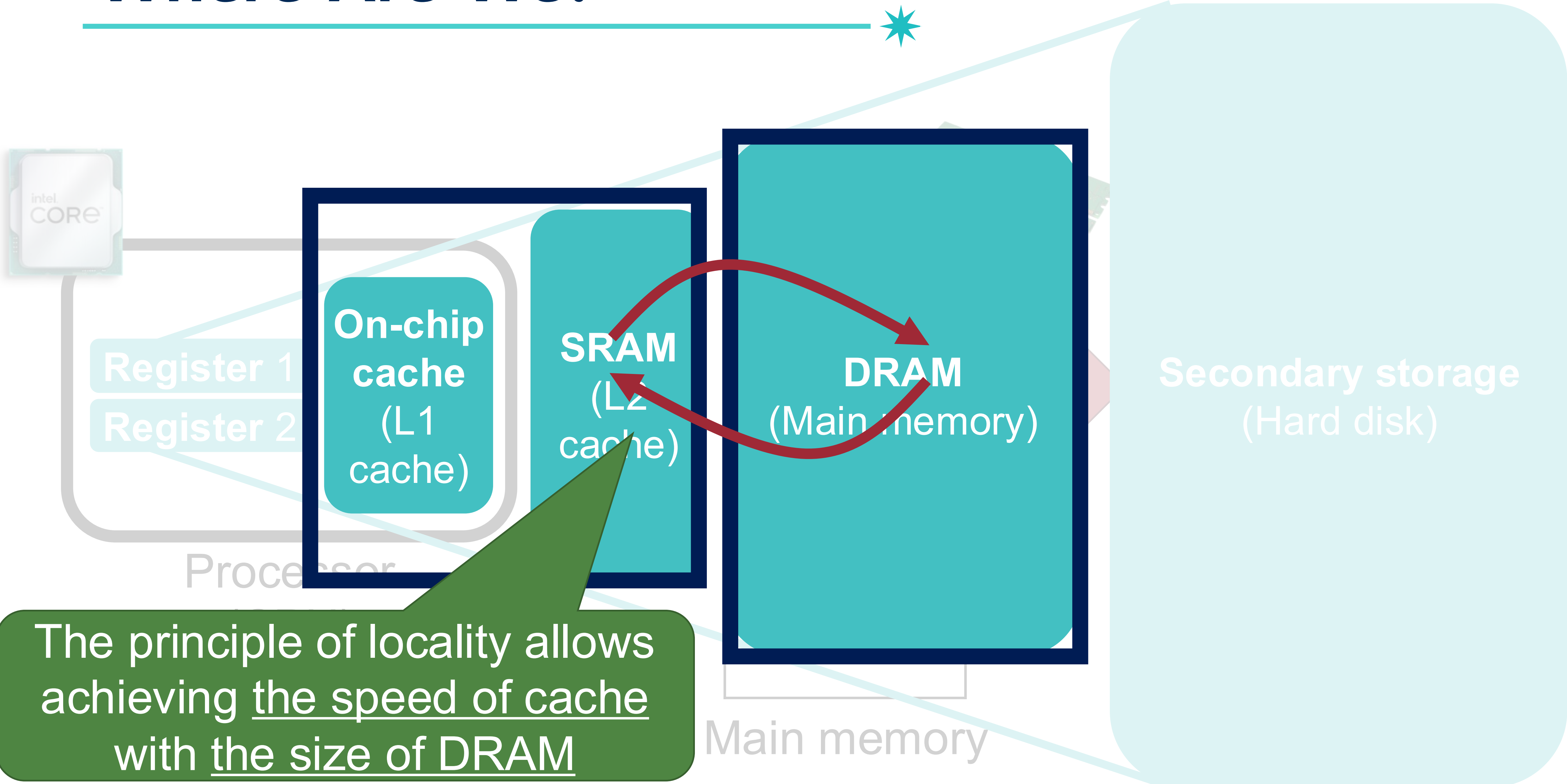
- Date and Time: May 22 (Fri.), 2PM~
- Venue: Room E103, Building 104
- Target: Interested Undergraduate Students
- Details: Information on the curriculum, completion requirements, KISA support programs, etc.
- Please register in advance!: https://forms.gle/KQMnCVmZXezAsgmj8

# Where Are We?



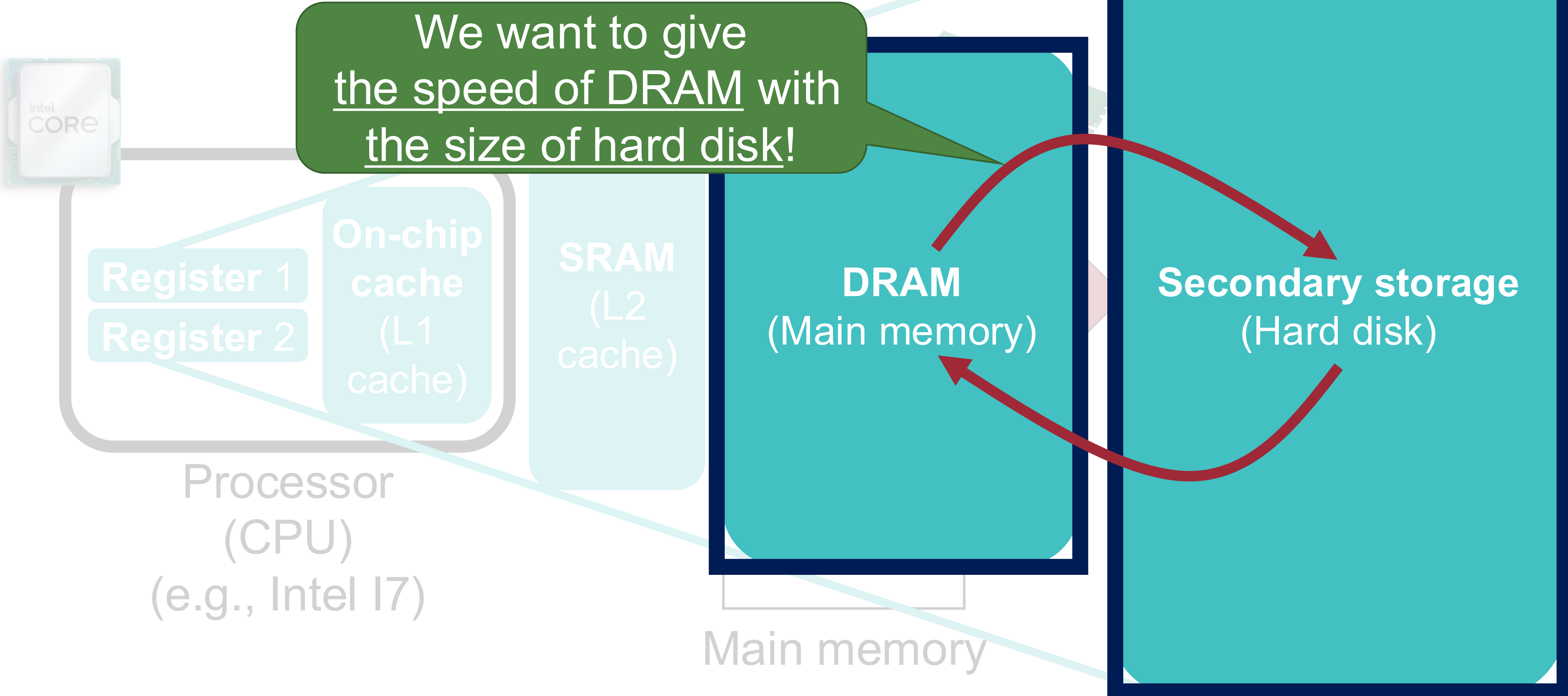
# Where Are We?





# Today's Topic: DRAM/HDD





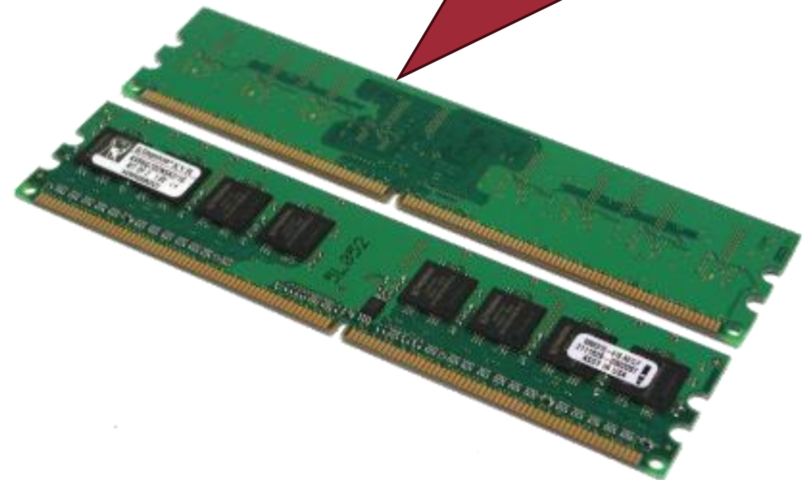
# Introduction to Virtual Memory

# In the Physical World, We Have Limited-sized Memory

```
mov rax, [rbx]
```

Processor

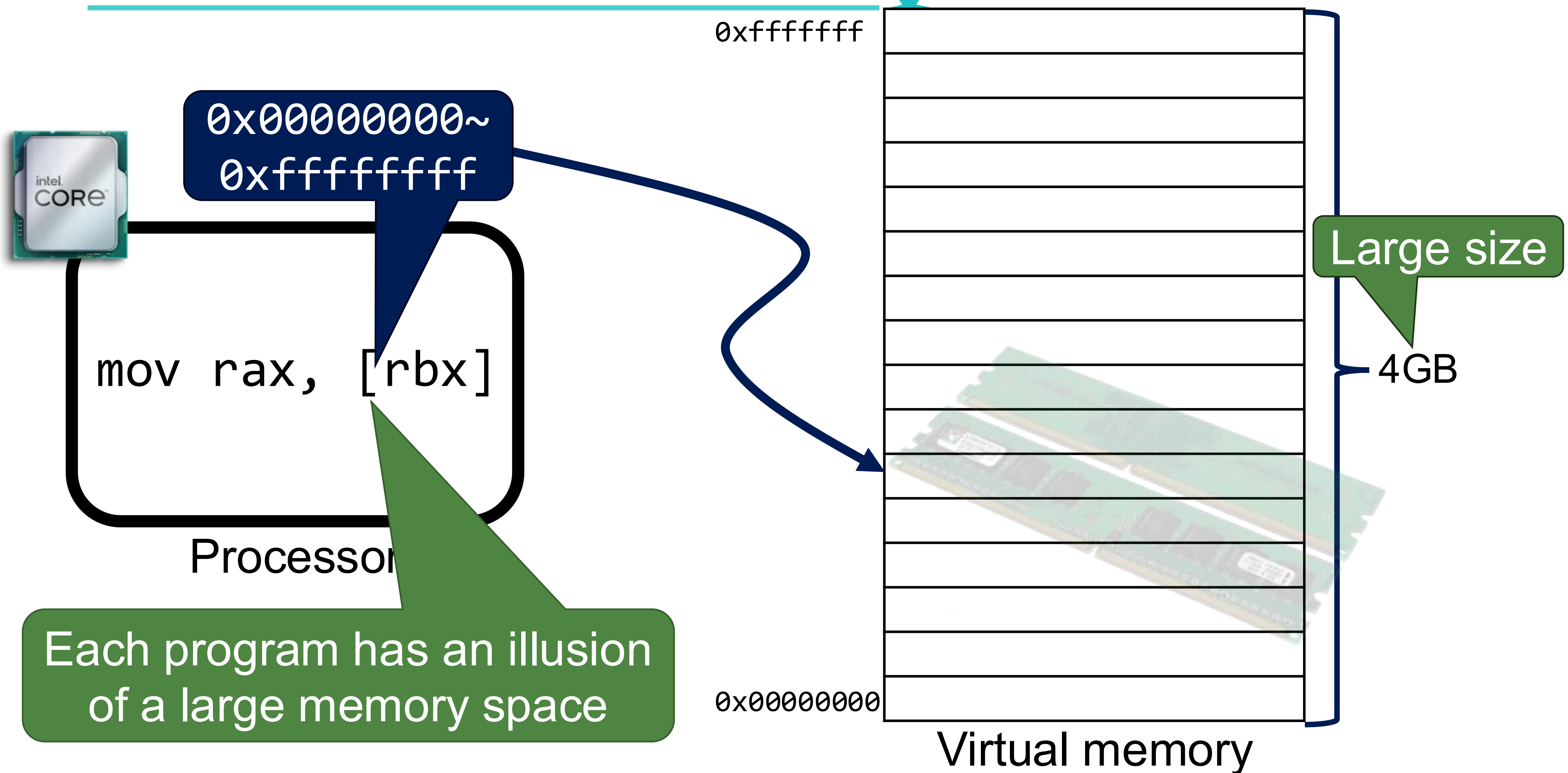
Very small size (e.g., 256MB)



Physical memory

# But, In the Virtual (Program) World, ...





# Virtual Memory

- Give programmers **an illusion of a large memory space** irrespective of actual capacity

# Virtual Memory

- Give programmers **an illusion of a large memory space** irrespective of actual capacity

***How do we give an illusion despite having small-sized physical memory?***

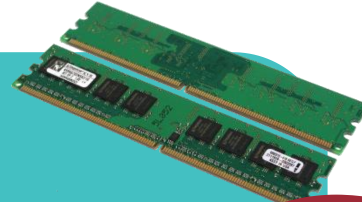
**Memory hierarchy between RAM and HDD**

# Memory Hierarchy: DRAM/HDD

0x00000000~
0xffffffff

```
mov rax, [rbx]
```

Processor

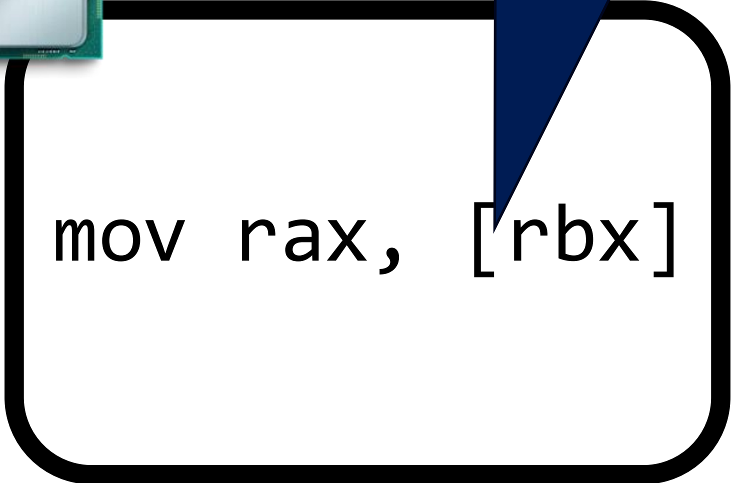


**DRAM**
(256MB)

**Secondary storage**
(100GB)

Give the
speed of the DRAM with
the size of hard disk!

# Memory Hierarchy: DRAM/HDD




Effective address
(virtual address)

# Memory Hierarchy: DRAM/HDD

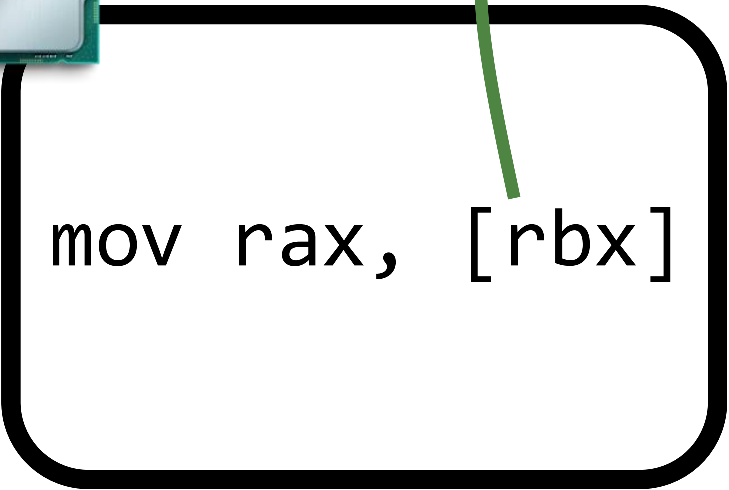
Virtual addresses

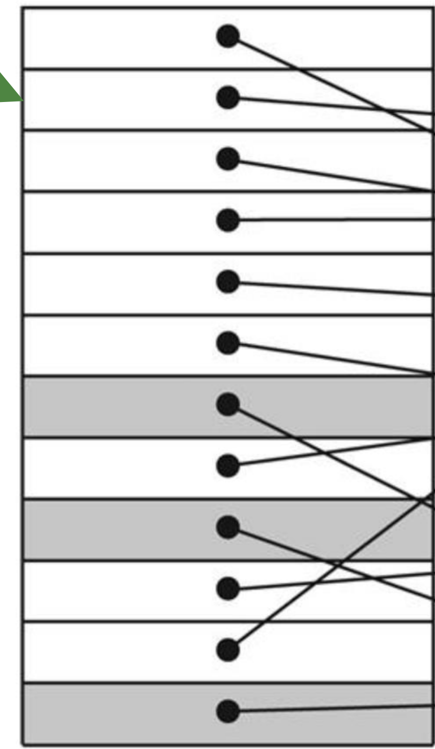
```
mov rax, [rbx]
```

Processor

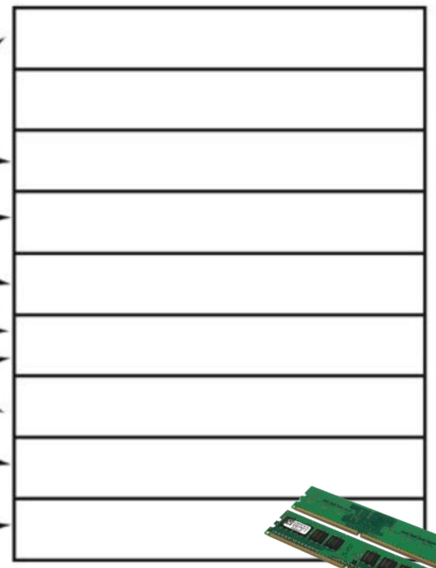
# Memory Hi

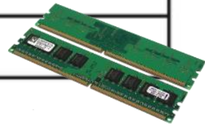
Address translation table (OS-managed table)
Virtual address → Physical address

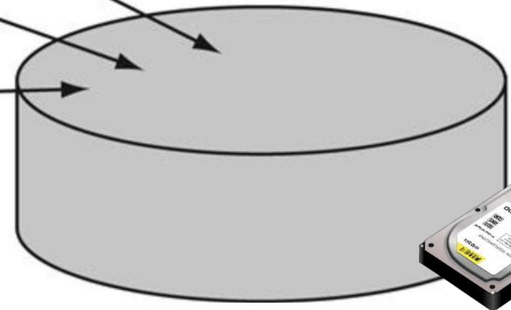



Address translation

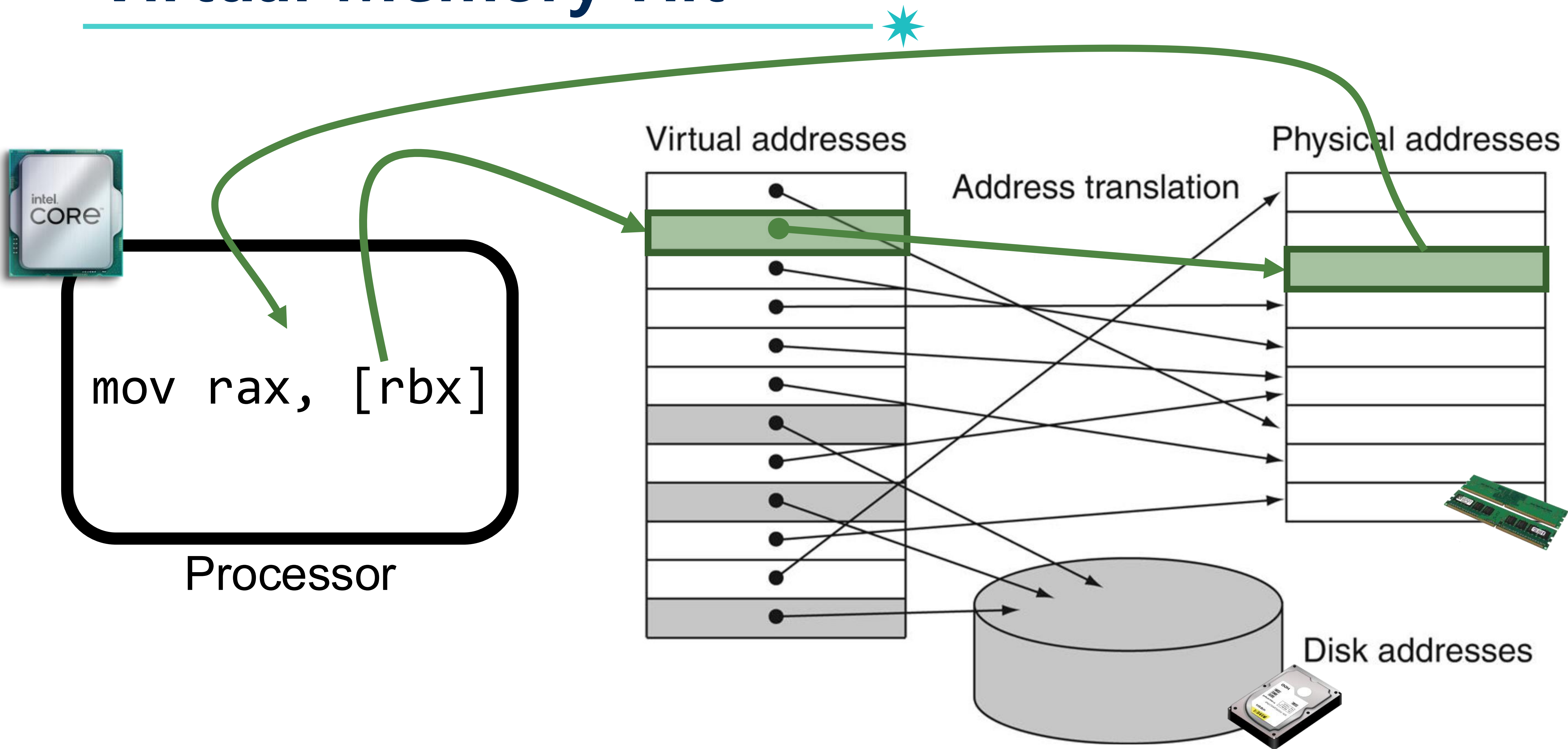




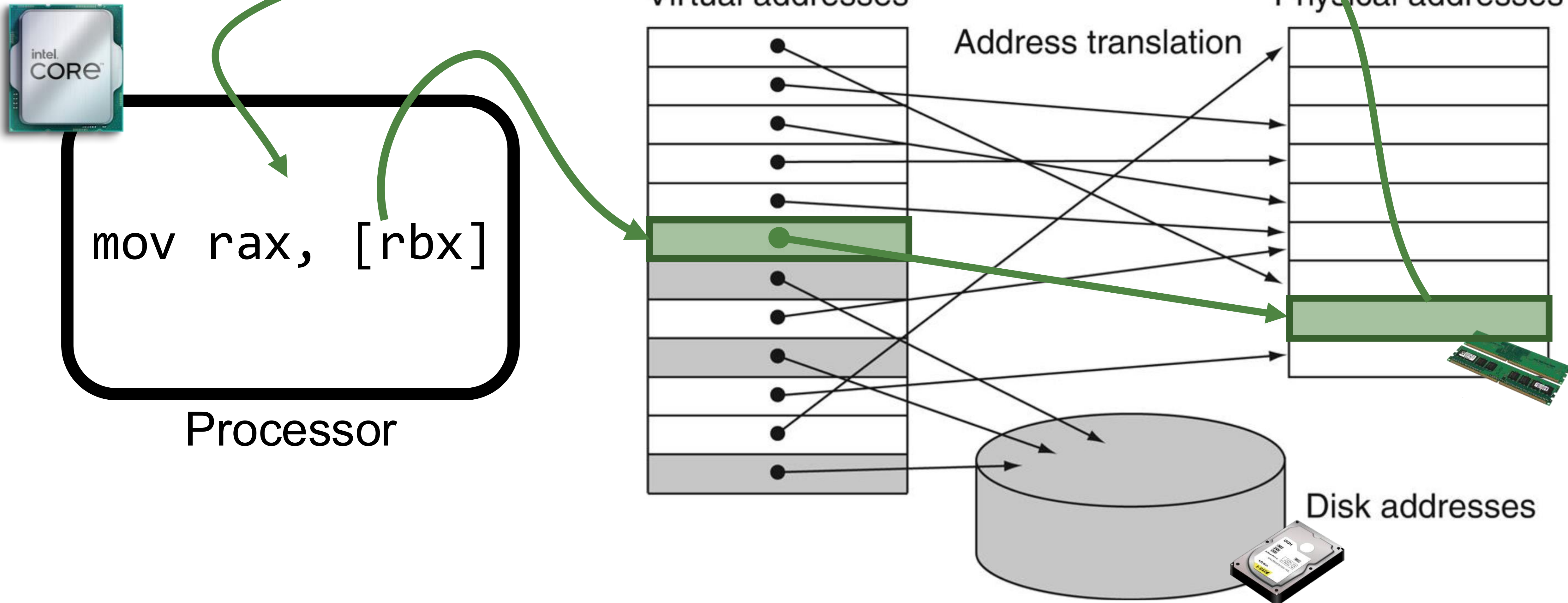
# Virtual Memory Hit





# Virtual Memory Hit





# Virtual Memory Miss (a.k.a., Page Fault)



```
mov rax, [rbx]
```

Processor

Virtual addresses

Address translation

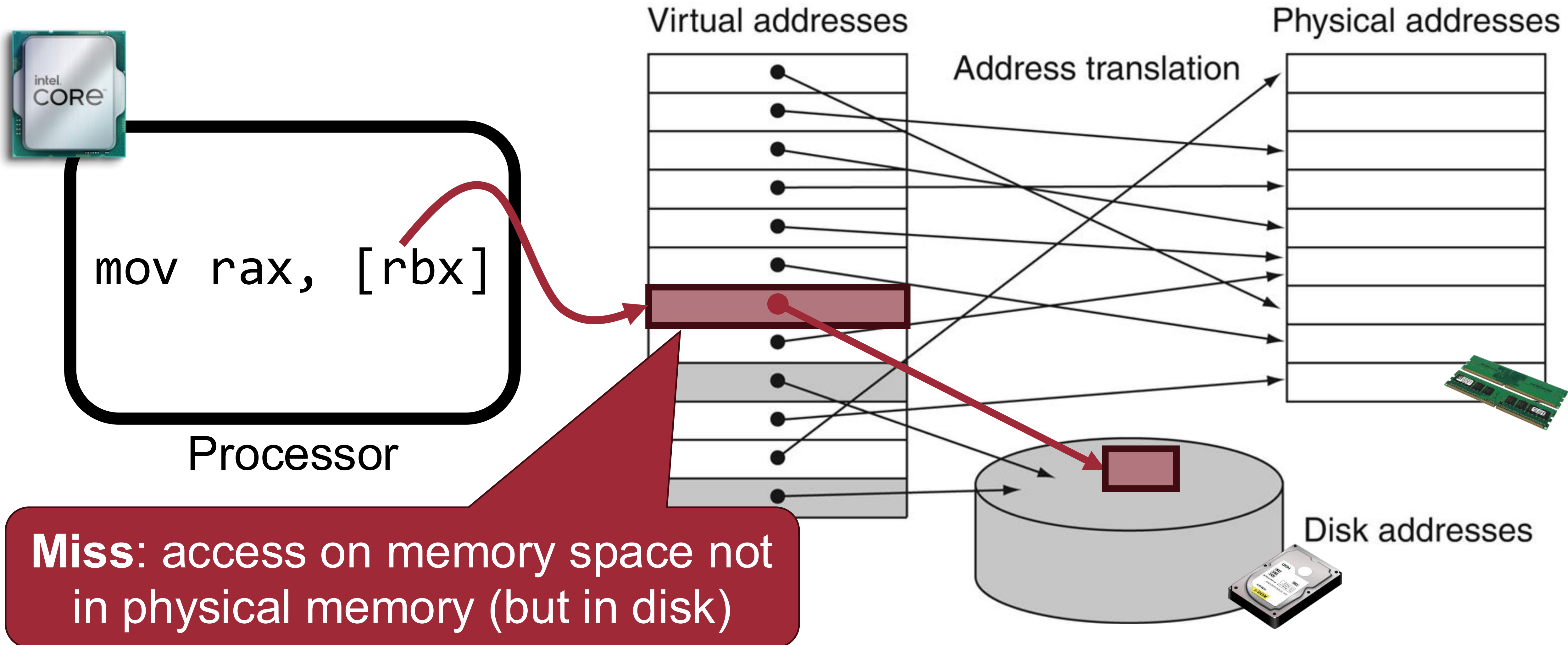
Physical addresses

Disk addresses

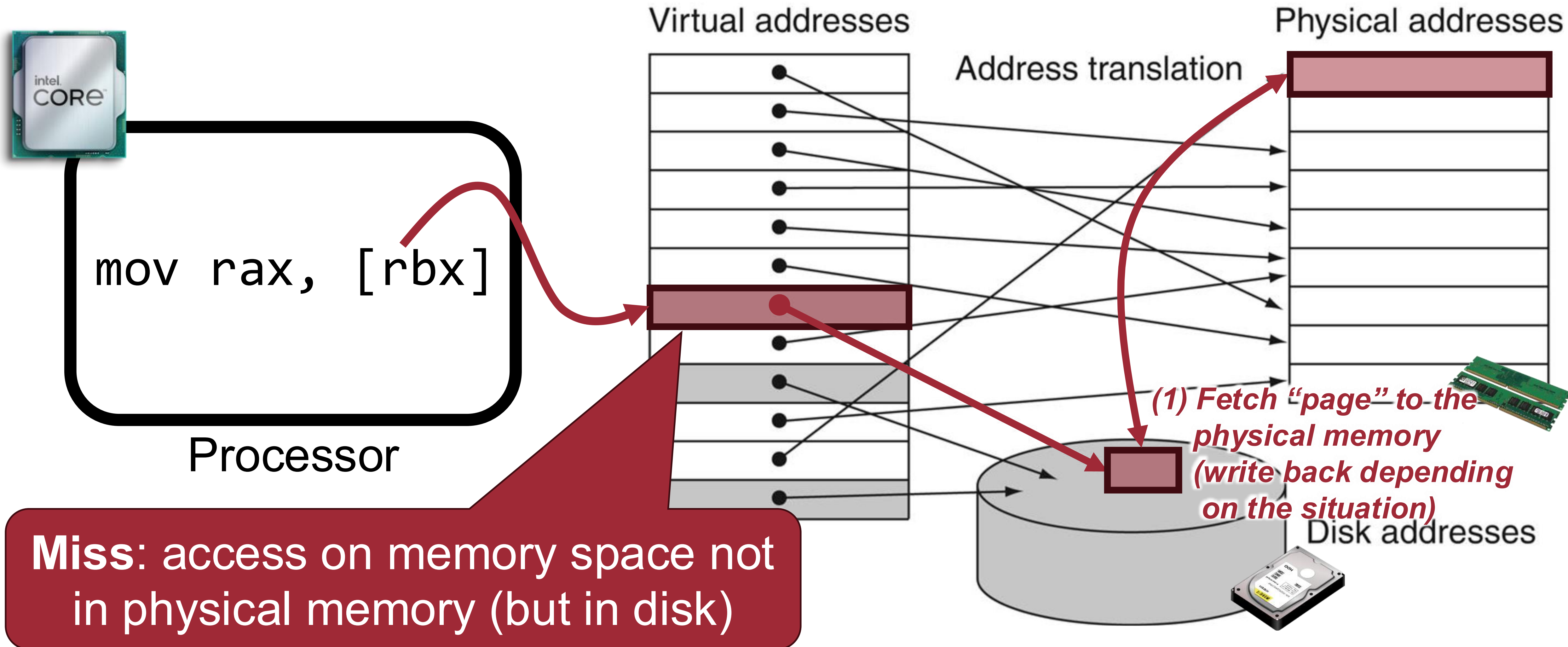
**Miss**: access on memory space not in physical memory (but in disk)

# Virtual Memory Miss (a.k.a., Page Fault)



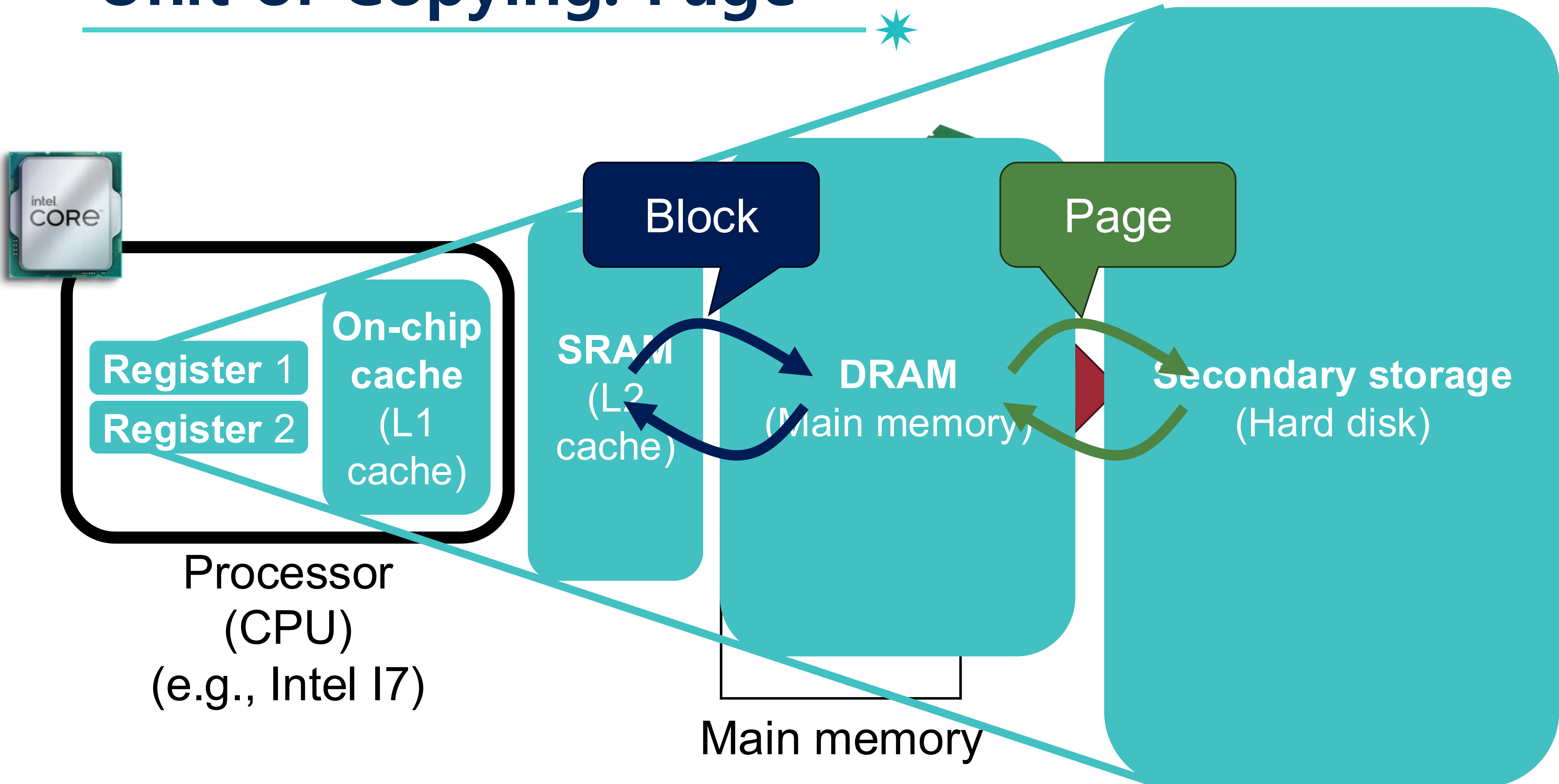


# Virtual Memory Miss (a.k.a., Page Fault)



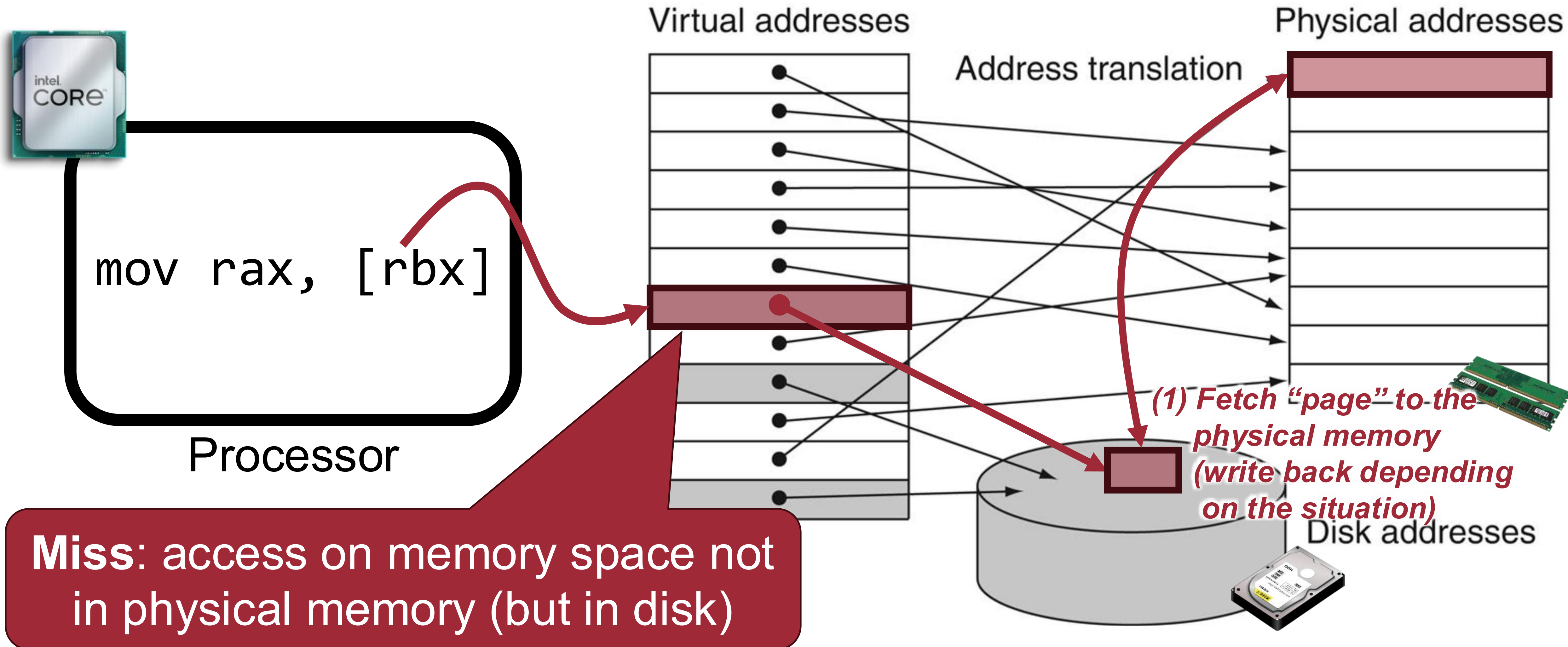
# Unit of Copying: Page



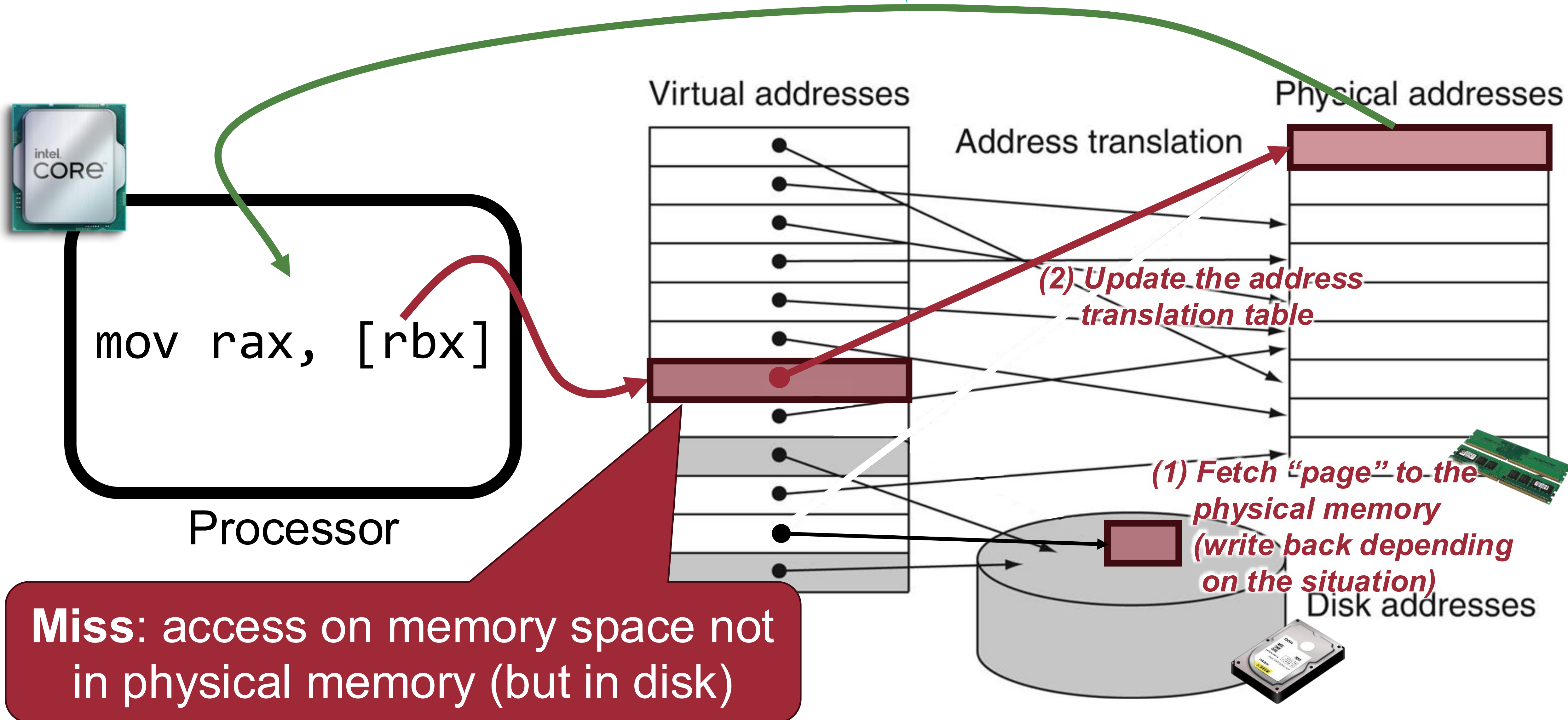


# Virtual Memory Miss (a.k.a., Page Fault)





# Virtual Memory Miss (a.k.a., Page Fault)



# Virtual Memory

- Give programmers **an illusion of a large memory space** irrespective of actual capacity
- Uses main memory as a “cache” for the hard disk

- Other benefits?
  - **Program simplification**: give each program the illusion that it has its *own private memory* (e.g., `0x00000000~0xffffffff`)

# Program Simplification

*Virtual address 0x100*

| P1's virtual address space |
| --- |
| Code |
| Data |
| Heap |
| |
| Stack |

P1 can use the entire address space (starting from 0x0)

Core #0

Core #1

Each process thinks it has its ***own memory space!***

*Virtual address 0x100*

| P2's virtual address space |
| --- |
| Code |
| Data |
| Heap |
| |
| Stack |

P2 can use the entire virtual address space (starting from 0x0)

# Program Simplification

Virtual address 0x100

Core #0

Core #1

Code
Data
Heap
Stack

P1's virtual address space

Address translation for P1

physical address

Physical memory

Code
Data
Heap
Stack

Code
Data
Heap
Stack

Virtual address 0x100

Code
Data
Heap
Stack

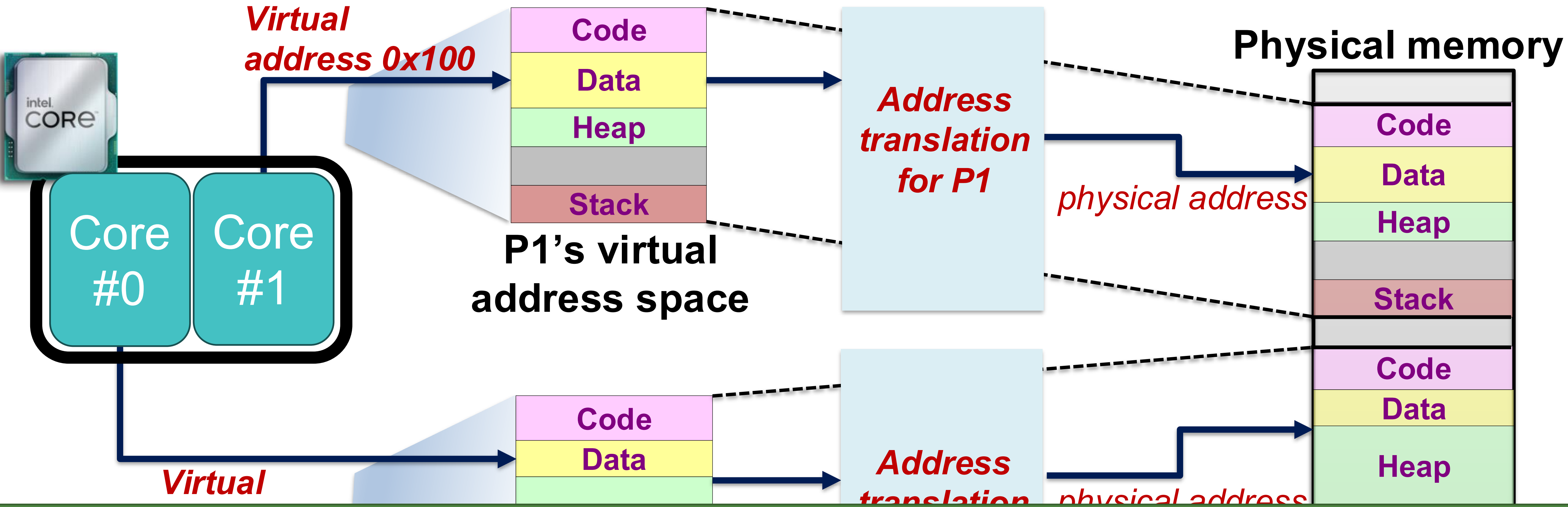
P2's virtual address space

Address translation for P2

physical address

# Program Simplification






Programming and storage management ease

# Virtual Memory

- Give programmers **an illusion of a large memory space** irrespective of actual capacity
- Uses main memory as a “cache” for the hard disk

- Other benefits?
  - **Program simplification**: give each program the illusion that it has its *own private memory* (e.g., `0x00000000~0xffffffff`)
  - **Security**: memory protection between processes

# Memory Protection

Virtual address 0x100

Core #0

Core #1

Code

Data

Heap

Stack

P1's virtual address space

Address translation for P1

physical address

Physical memory

Code

Data

Heap

Stack

Code

Data

Heap

Stack

Virtual address 0x100

Code

Data

Heap

Stack

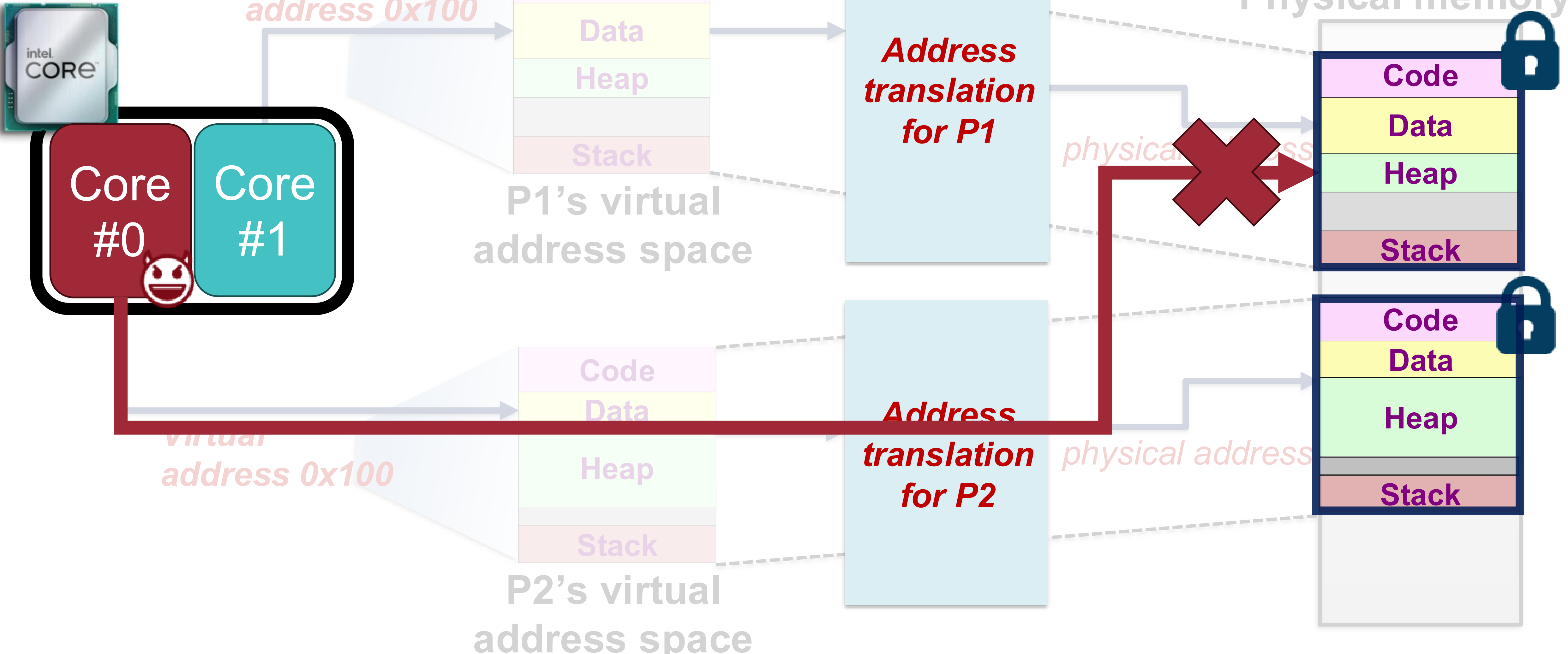
P2's virtual address space

Address translation for P2

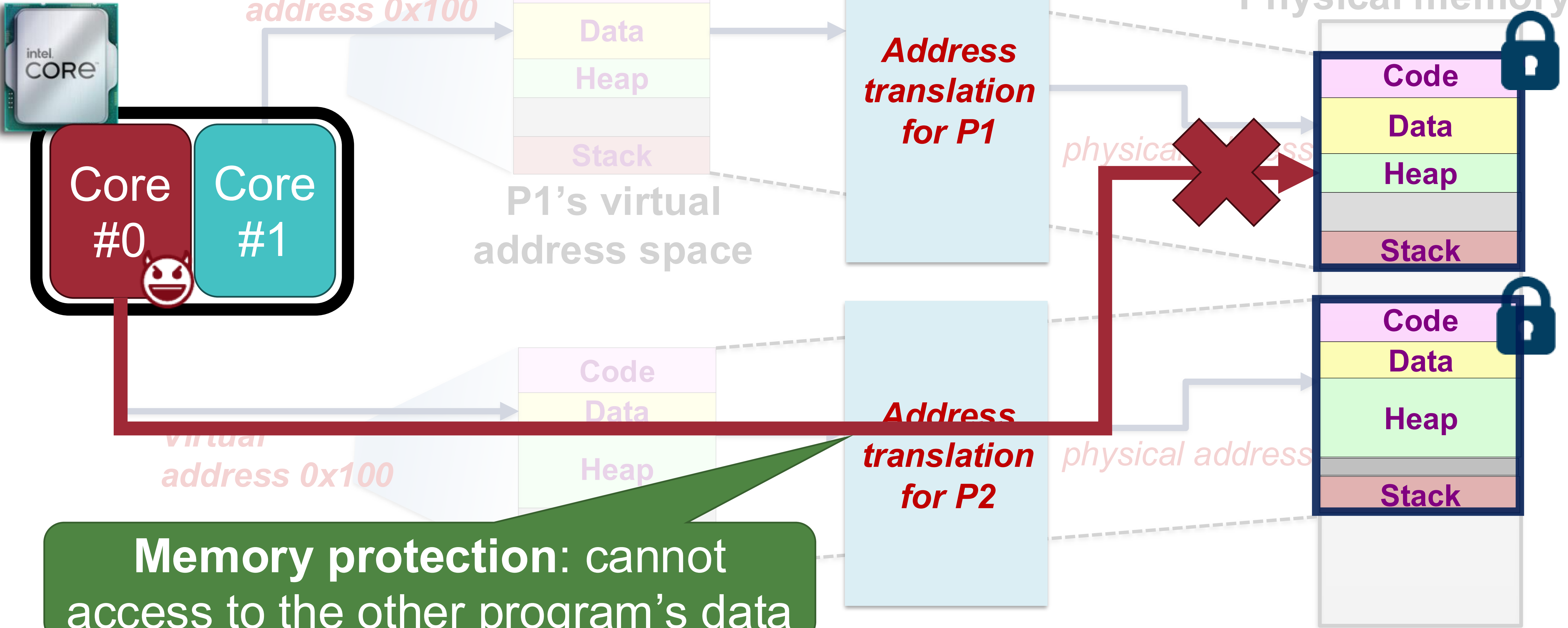
physical address

# Memory Protection





# Memory Protection



# Virtual Memory Summary

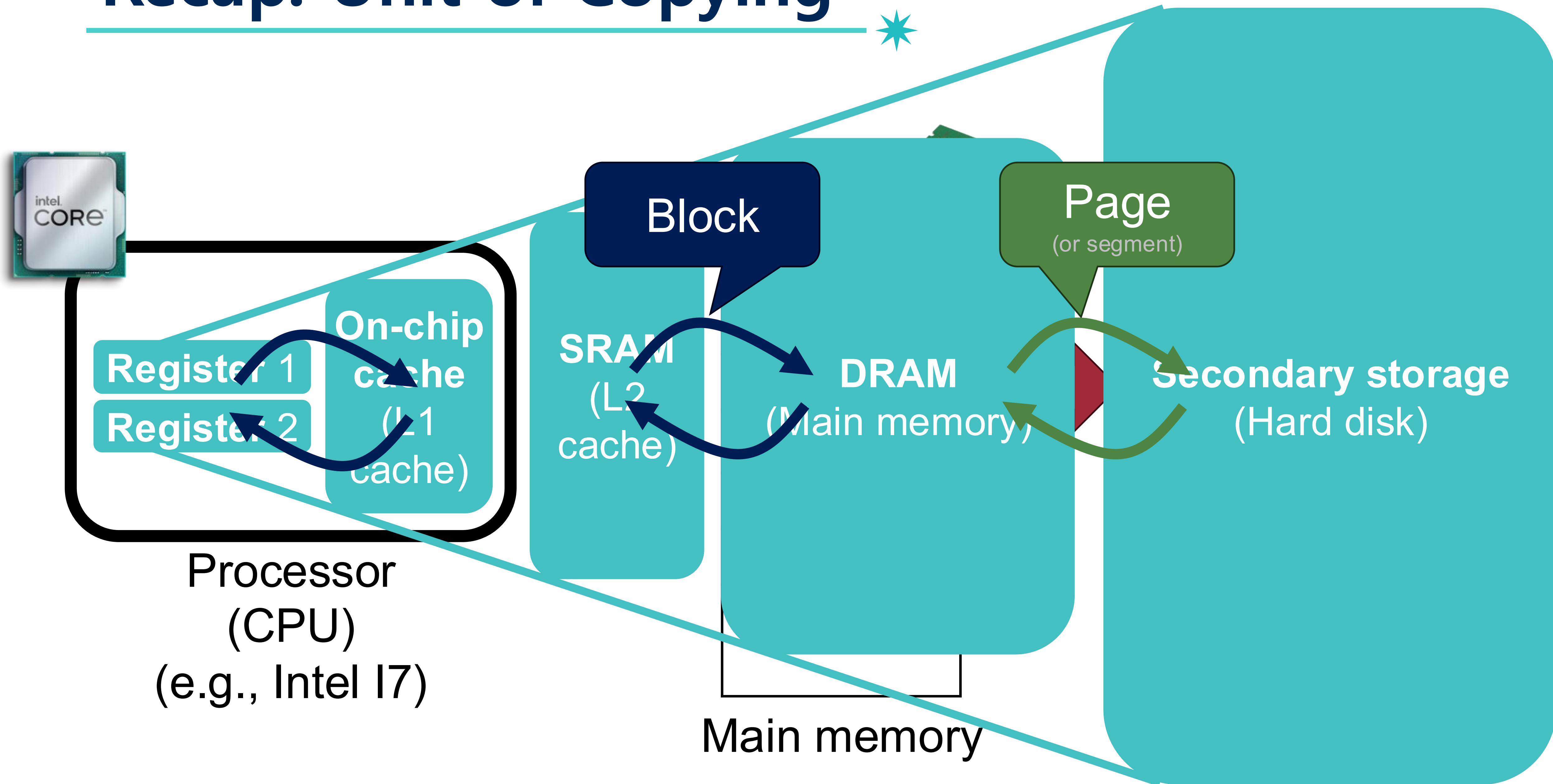
- Idea: Give programmers **an illusion of a large memory space** irrespective of actual capacity
  - Programmers assume they have “infinite” amount of physical memory
  - Programmers do not need to worry about how to manage it (simplifies memory management for multi-processing system)

- CPU and OS cooperatively manage physical memory space to provide the illusion on behalf of users
  - Illusion is maintained for each independent process
  - Let OS to protect programs or share memory (e.g., library code) from each other

# Unit of Copying: Page

# Recap: Unit of Copying





# Virtual Memory Block Type #1: Page

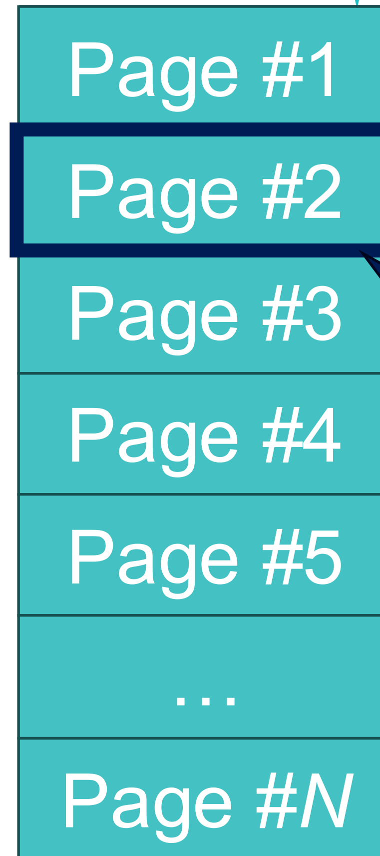


- A block of virtual memory
- Unit of copying
  - between DRAM and Disk
- Fixed length
  - e.g., 4KB
  - Generally, 4KB ~ 4MB

# Virtual Memory Block Type #1: Page



- A block of virtual memory
- Unit of copying
  - between DRAM and Disk
- Fixed length
  - e.g., 4KB
  - Generally, 4KB ~ 4MB

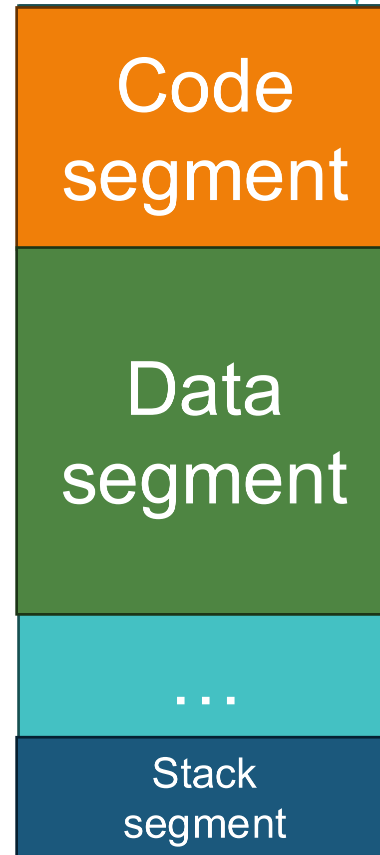



Fixed-length virtual memory block

# Virtual Memory Block Type #2: Segment

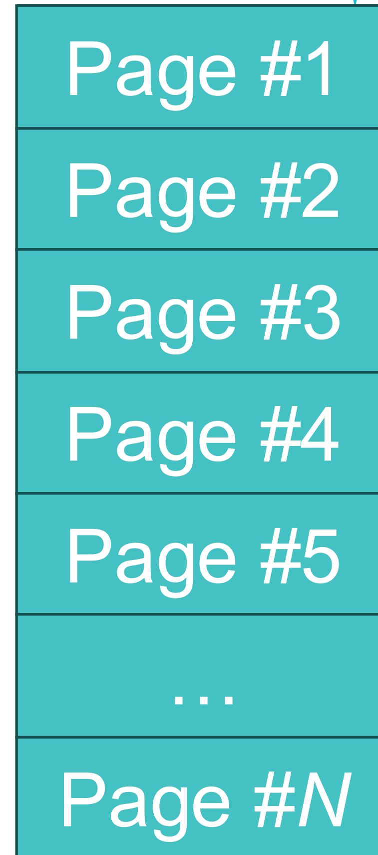


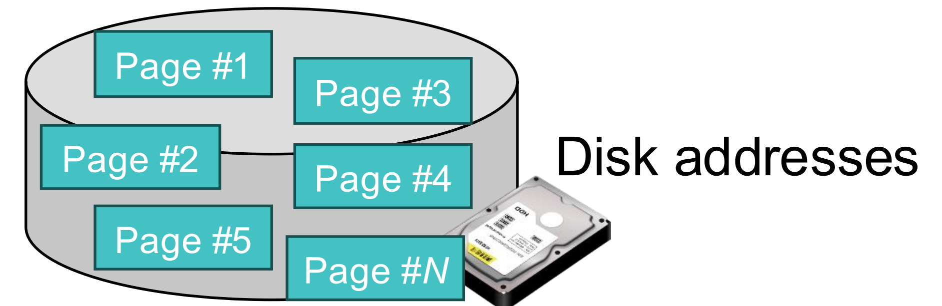
Segment length is variable!



# Page

- A block of virtual memory
- Unit of copying
  - between DRAM and Disk
- Fixed length
  - e.g., 4KB
  - Generally, 4KB ~ 4MB



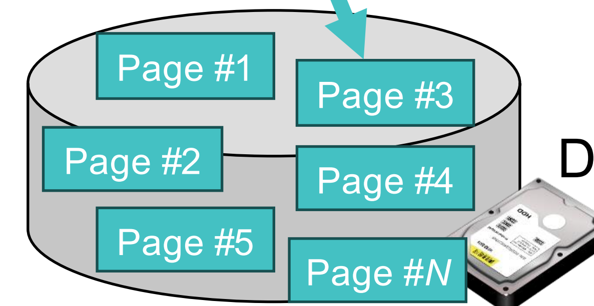
# Memory Paging (a.k.a., Swapping)

Memory management scheme by which a computer **stores and retrieves data** from secondary storage for use in main memory

Page: unit of copying!

Physical addresses

*Paging (Swapping)*

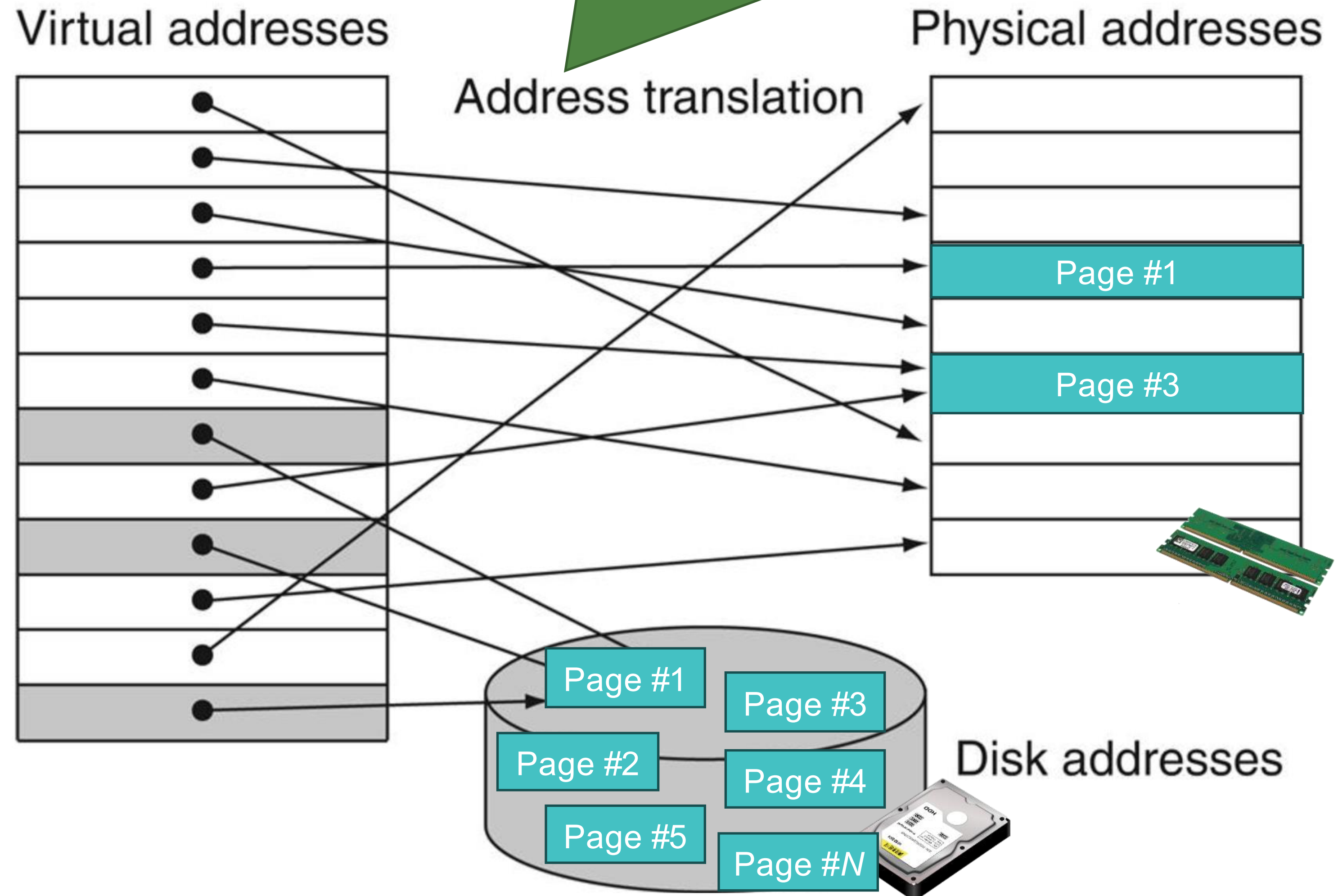



Disk addresses

# Address Translation

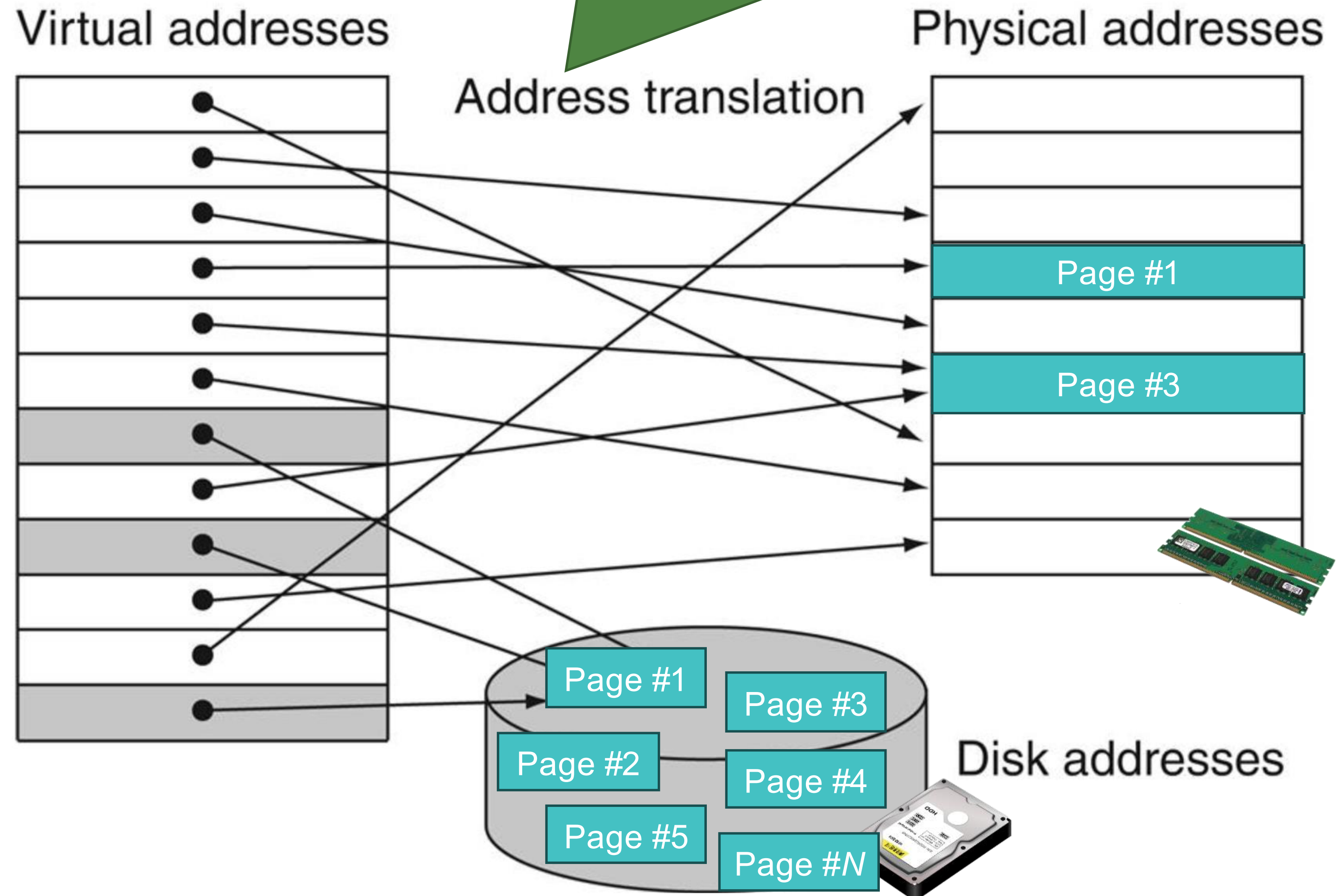
# Memory Hi

Address translation table (OS-managed table)
Virtual address → Physical address



# Memory Hi

“Page Table”

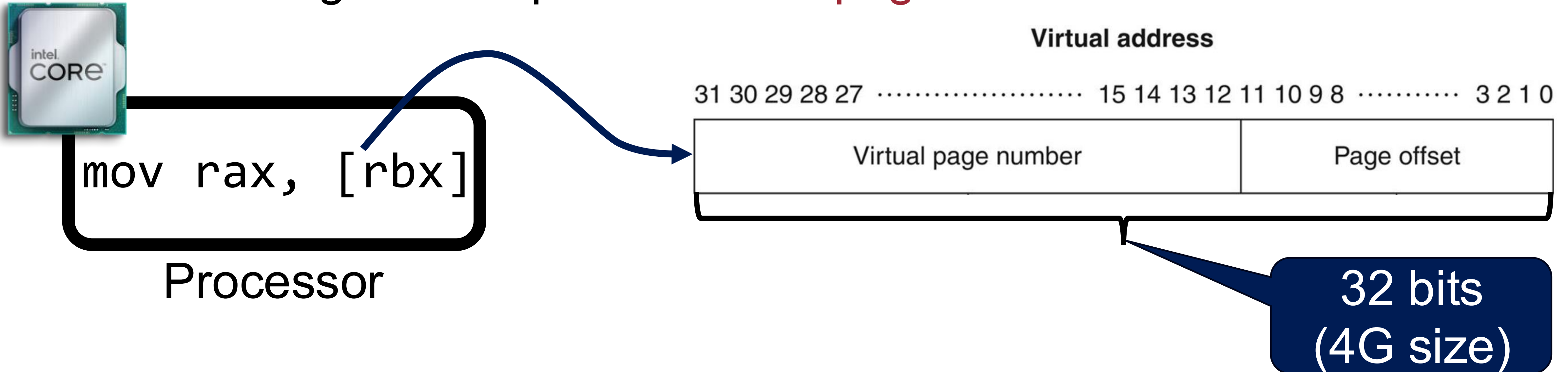


# Page Table: Address Translation

CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table

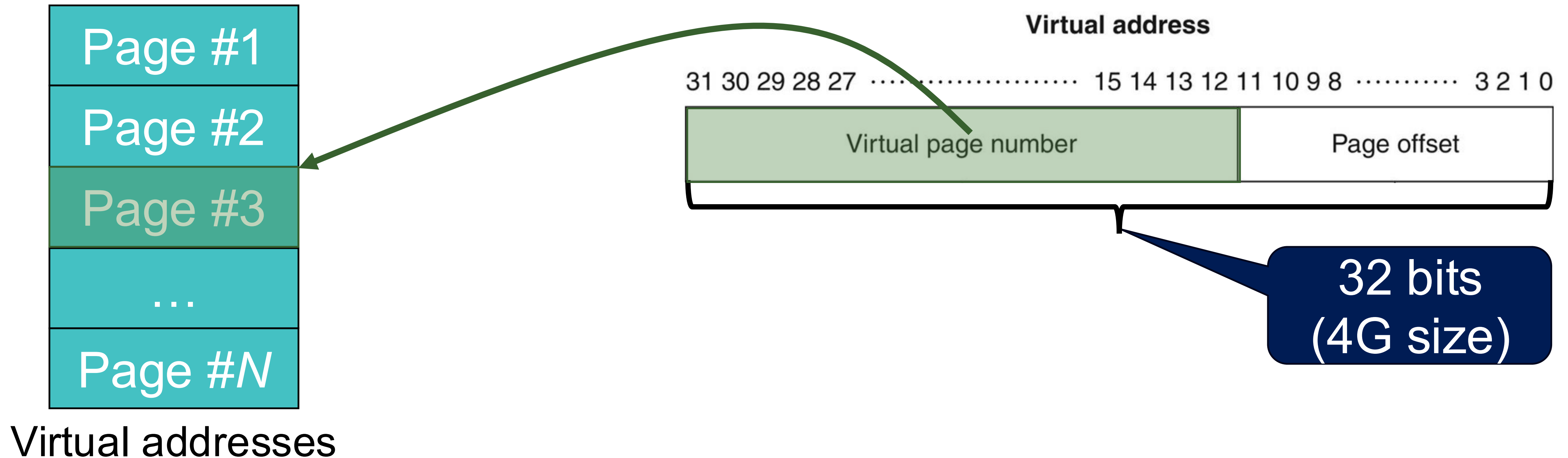
# Page Table: Address Translation

CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table



# Programmer's Point of View

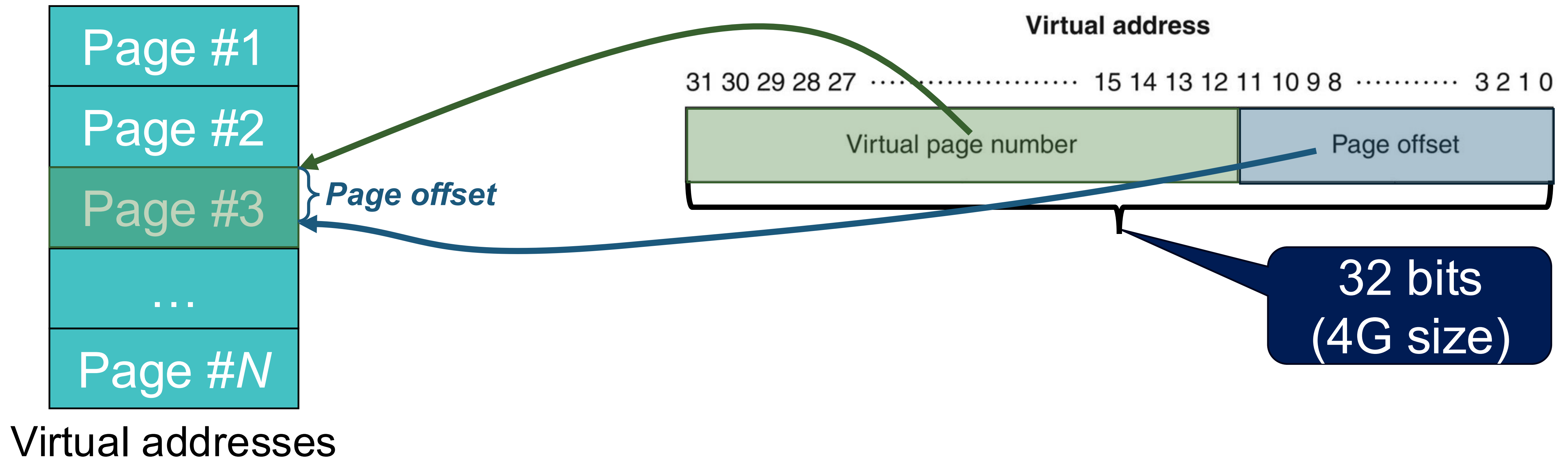
CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table



# Programmer’s Point of View



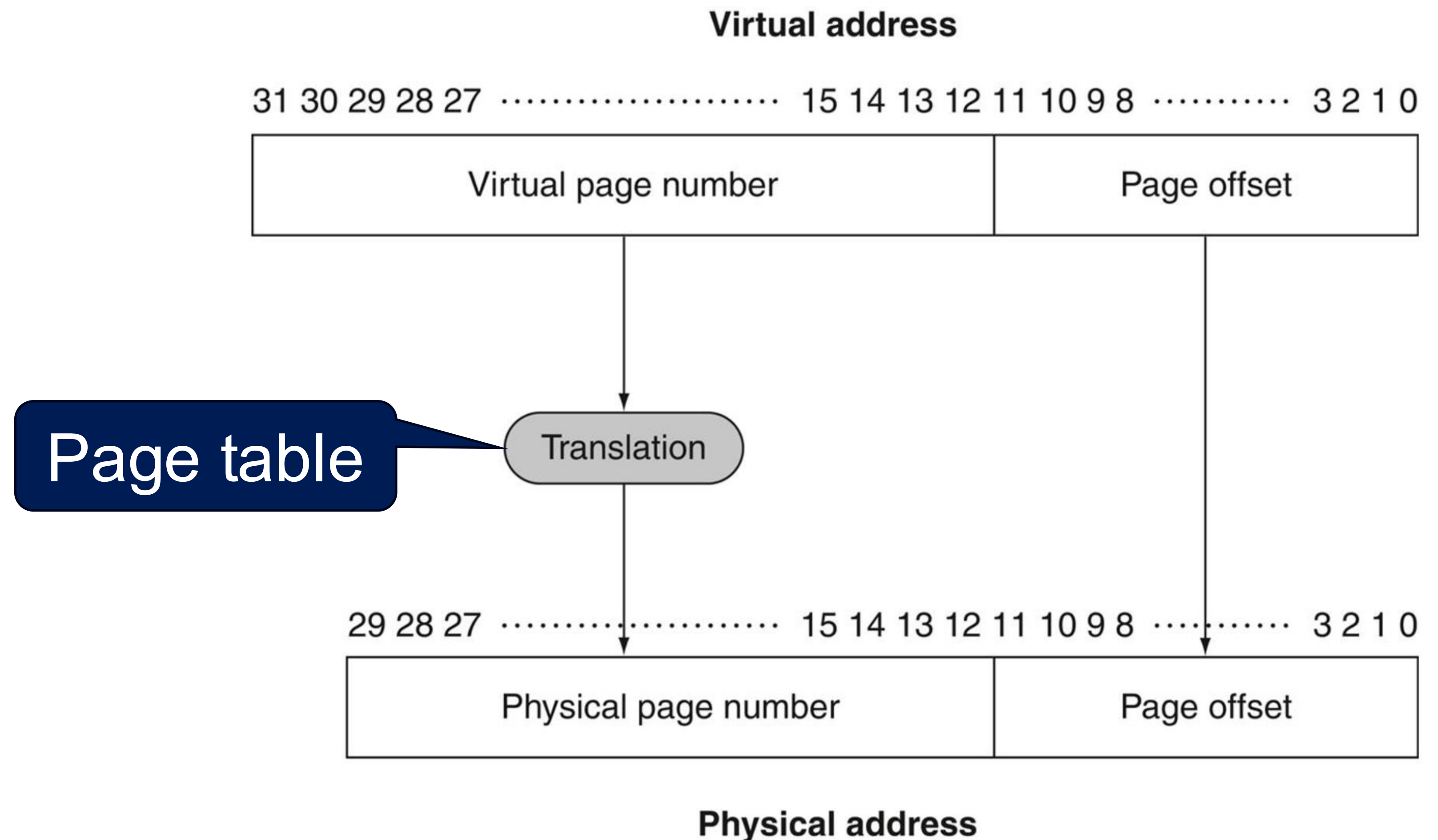
CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table



# Page Table: Address Translation

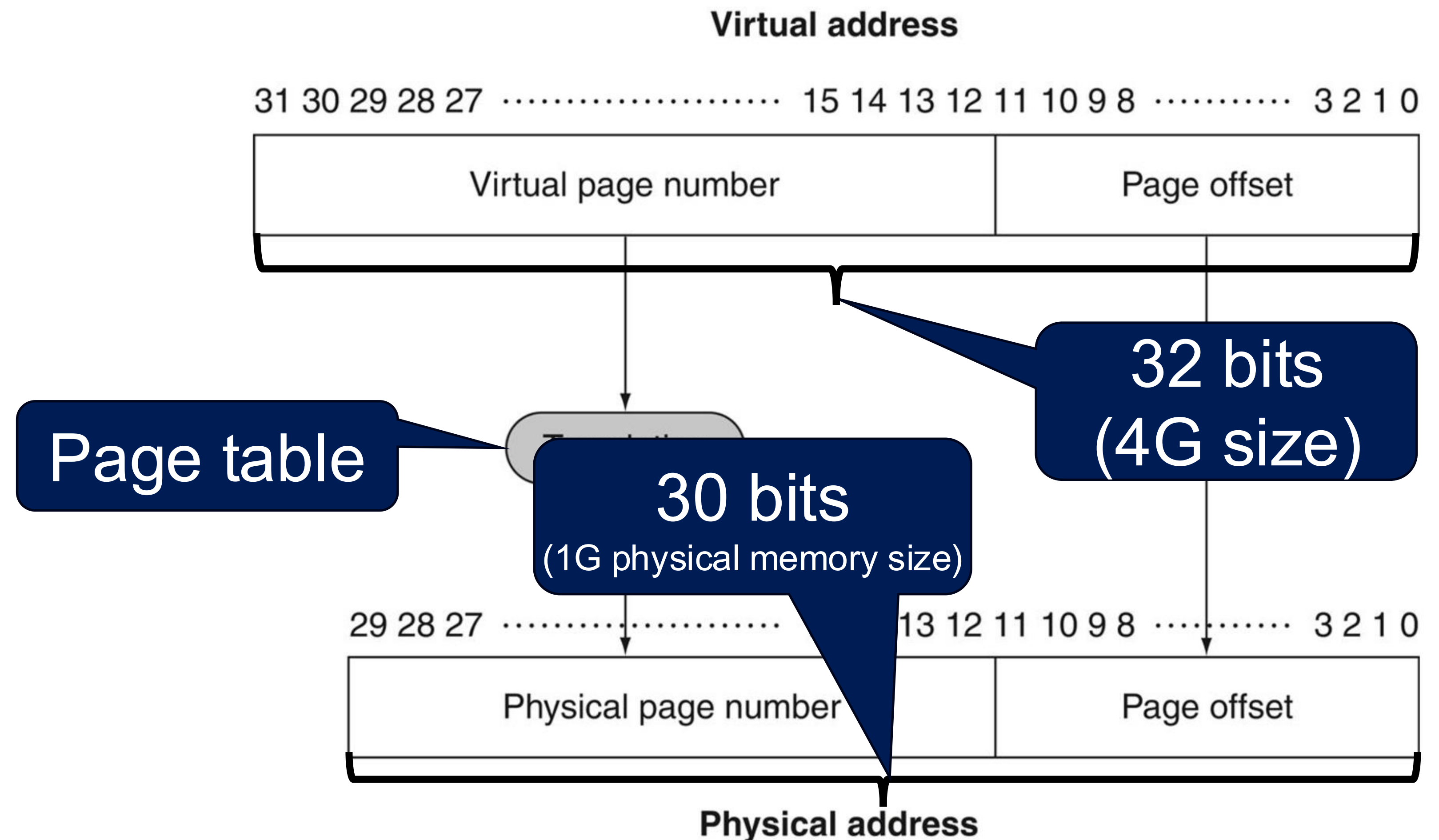


CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table



# Page Table: Address Translation

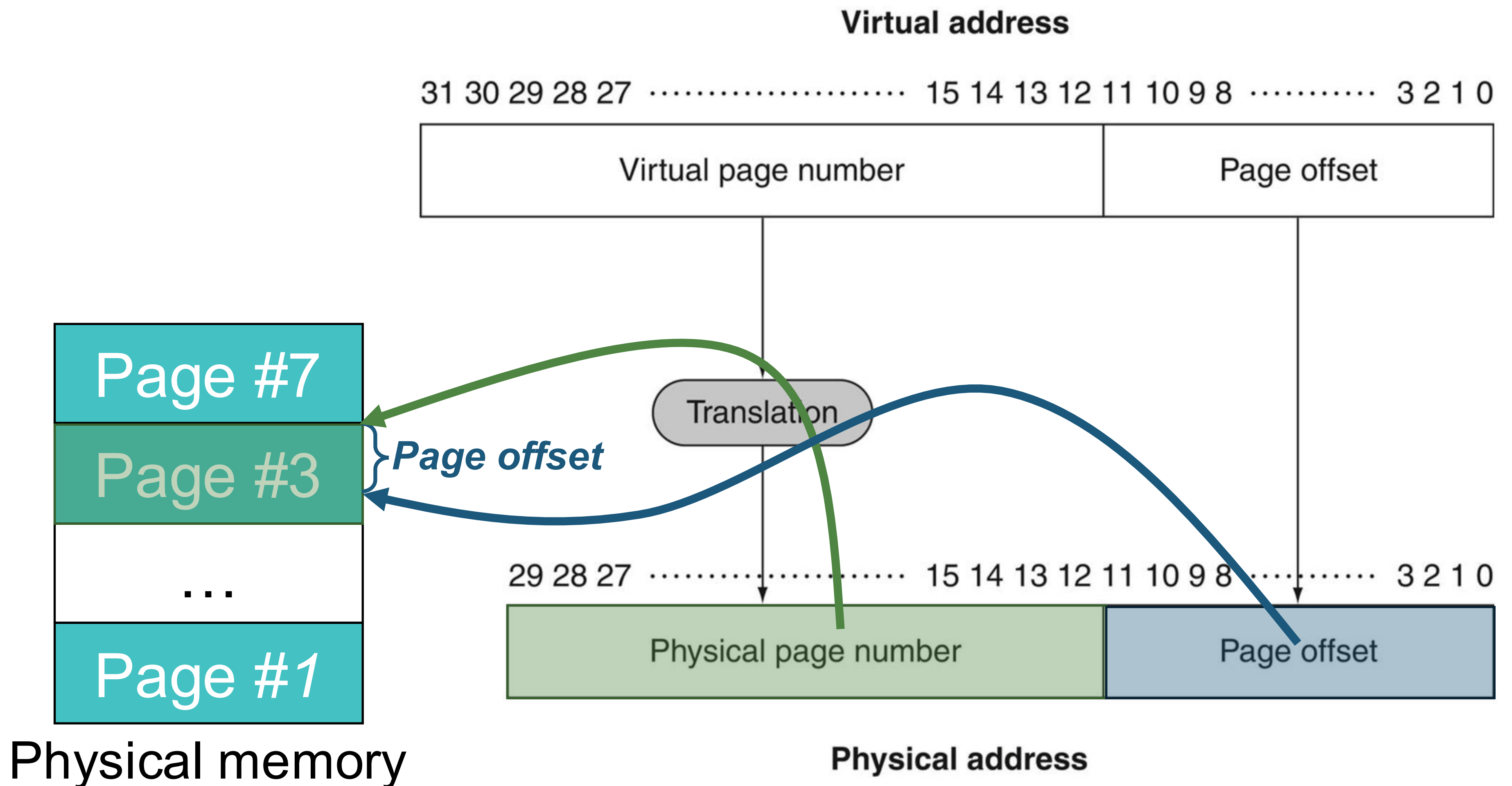
CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table



# Page Table: Address Translation

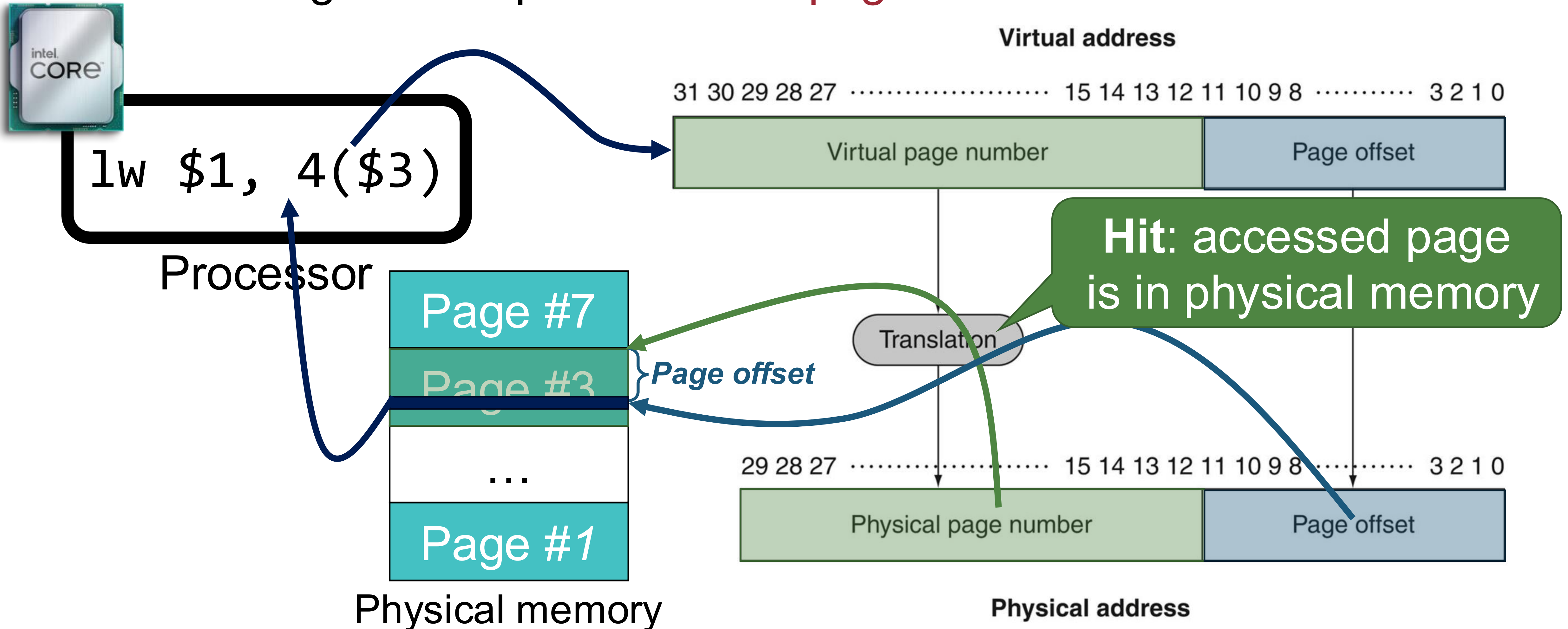


CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table



# Address Translation Flow (Hit Case)

CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table



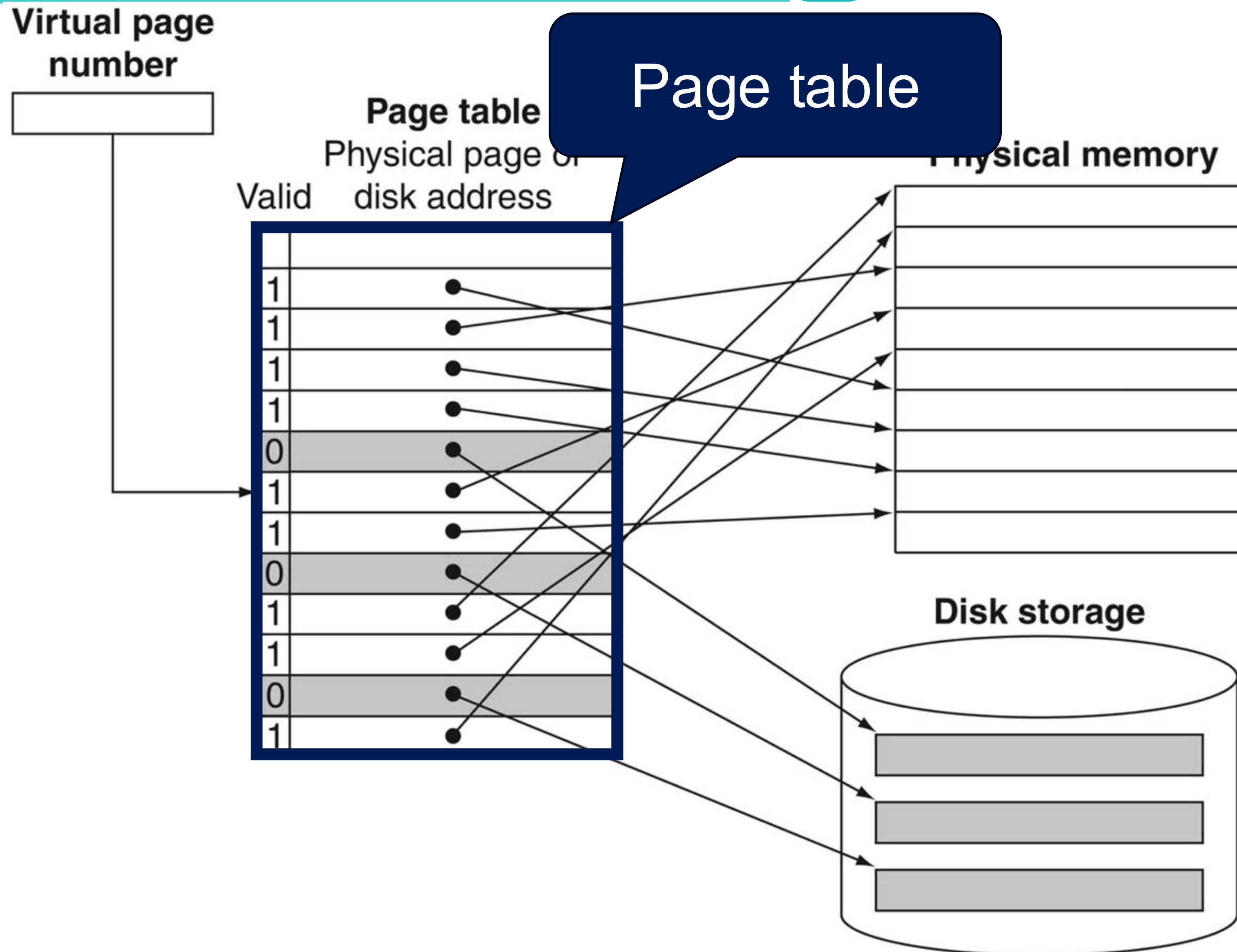
# Page Table: Address Translation

CPU converts virtual addresses into physical addresses via an OS-managed lookup table called page table

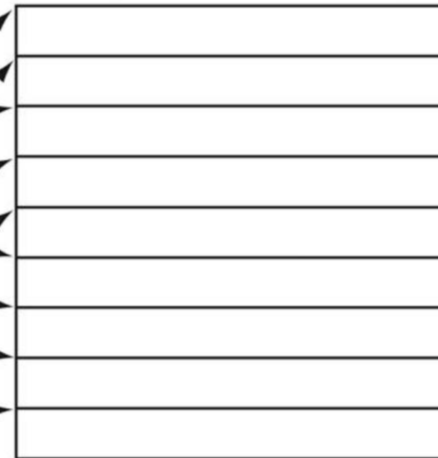
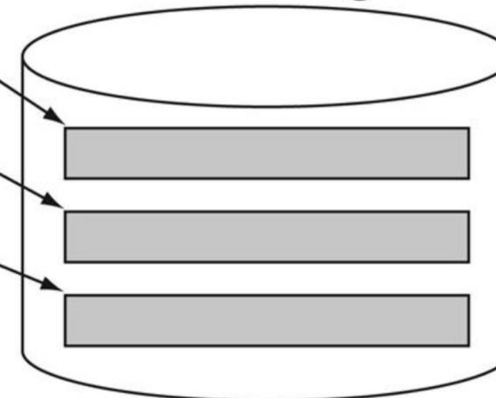
- **Mapping from a virtual to physical address**
  - **Virtual address** = virtual page number + page offset
  - **Physical address** = physical page number + page offset

- Each program has its own page table!

# Page Table: Details

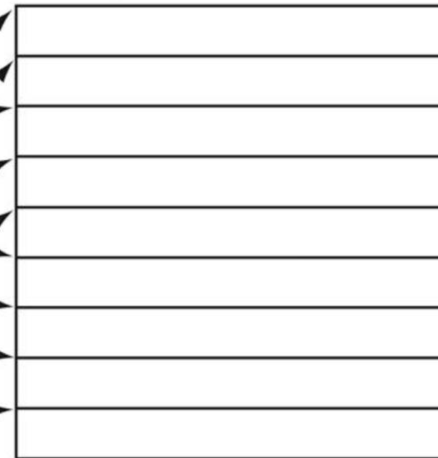


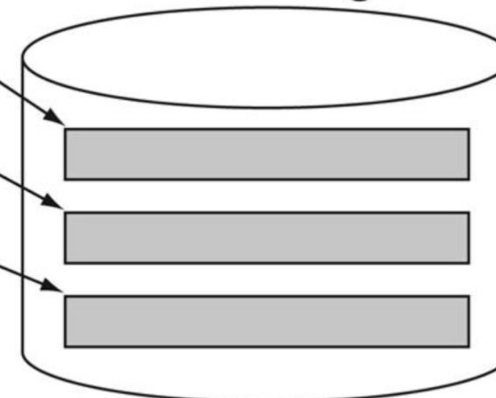
# Page Table: Details

# Page Table: Details

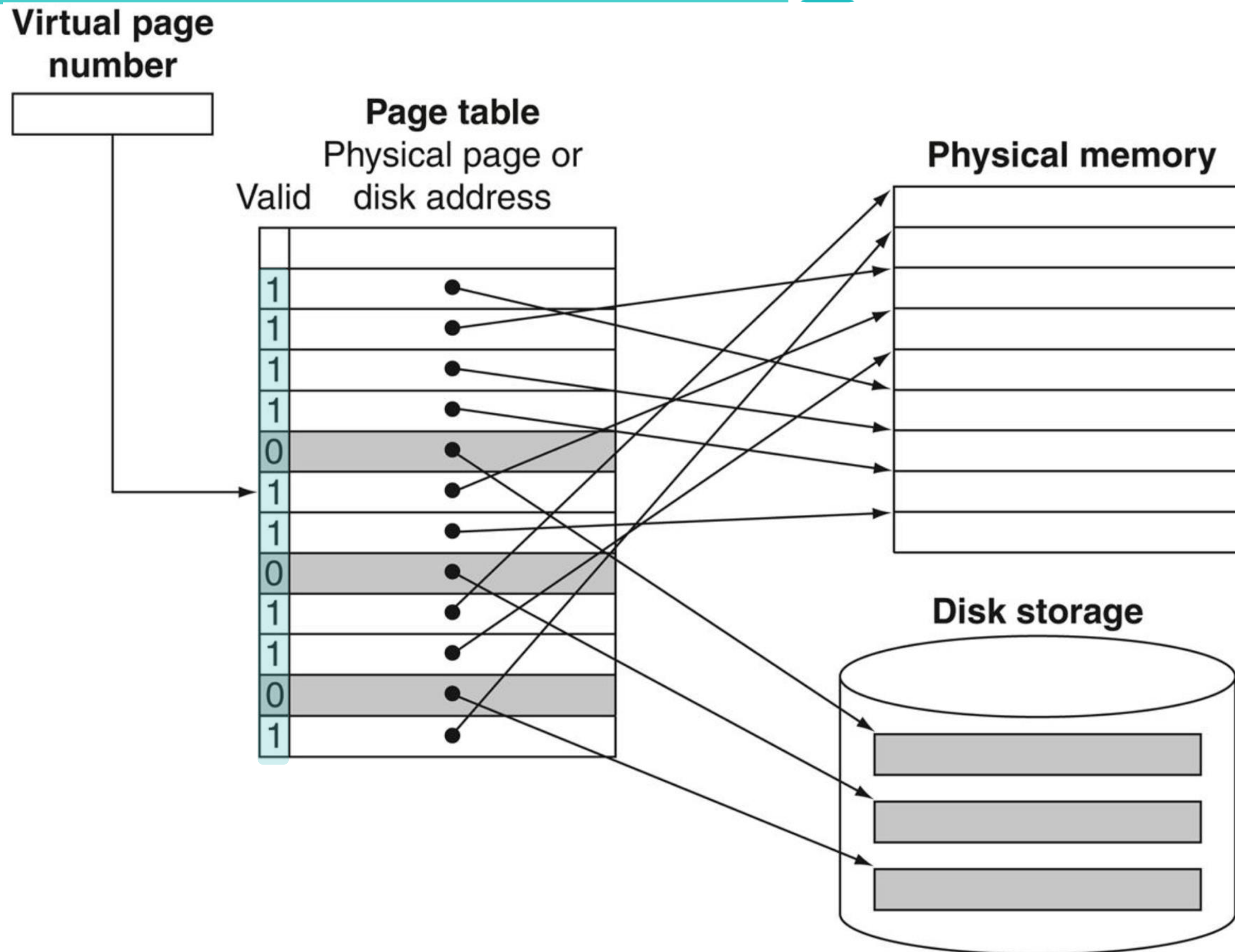
Entry: Physical page number (or disk address)



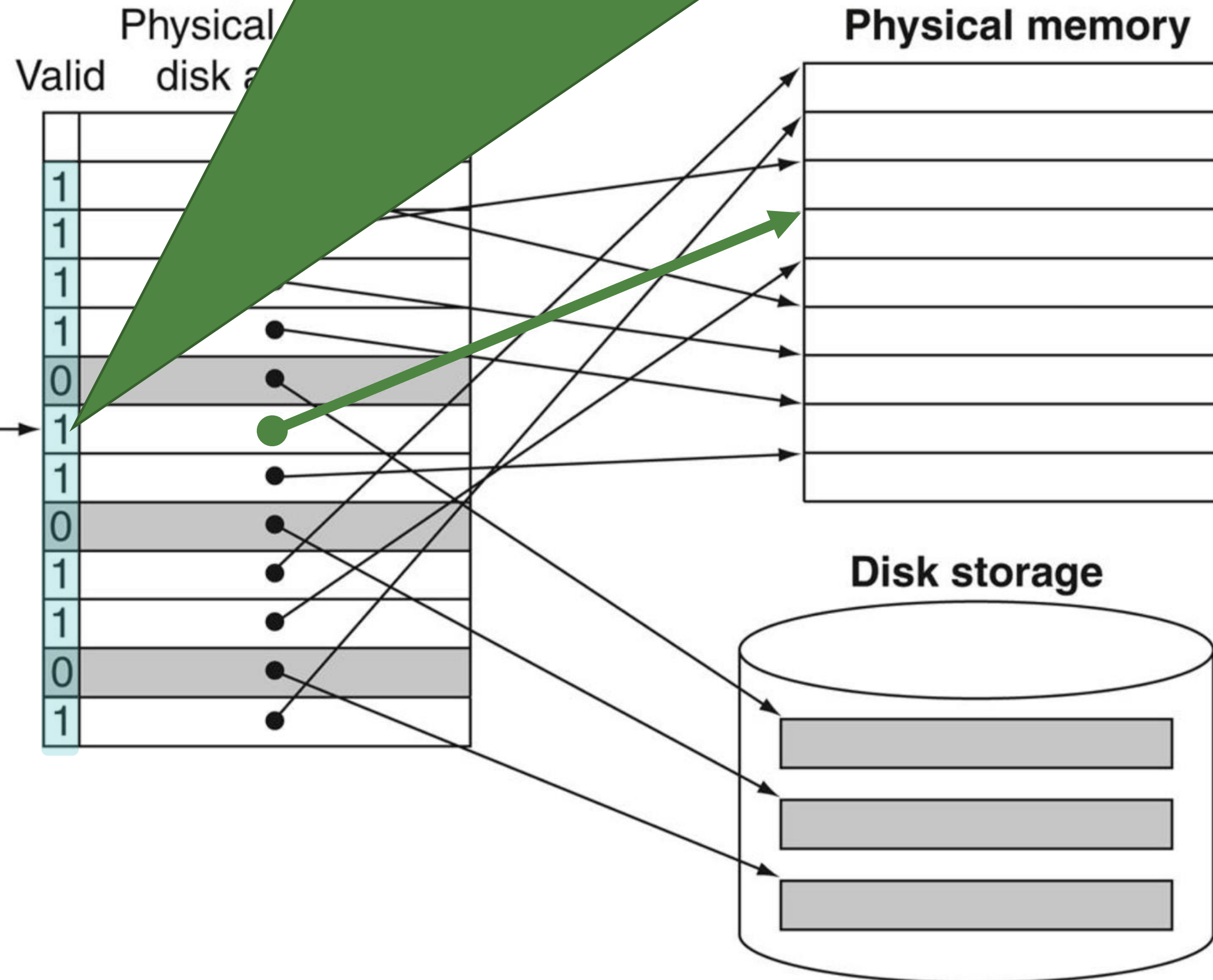

Index: virtual page number (as offset)
(+ page table base address)

# Valid Bit in Page Table

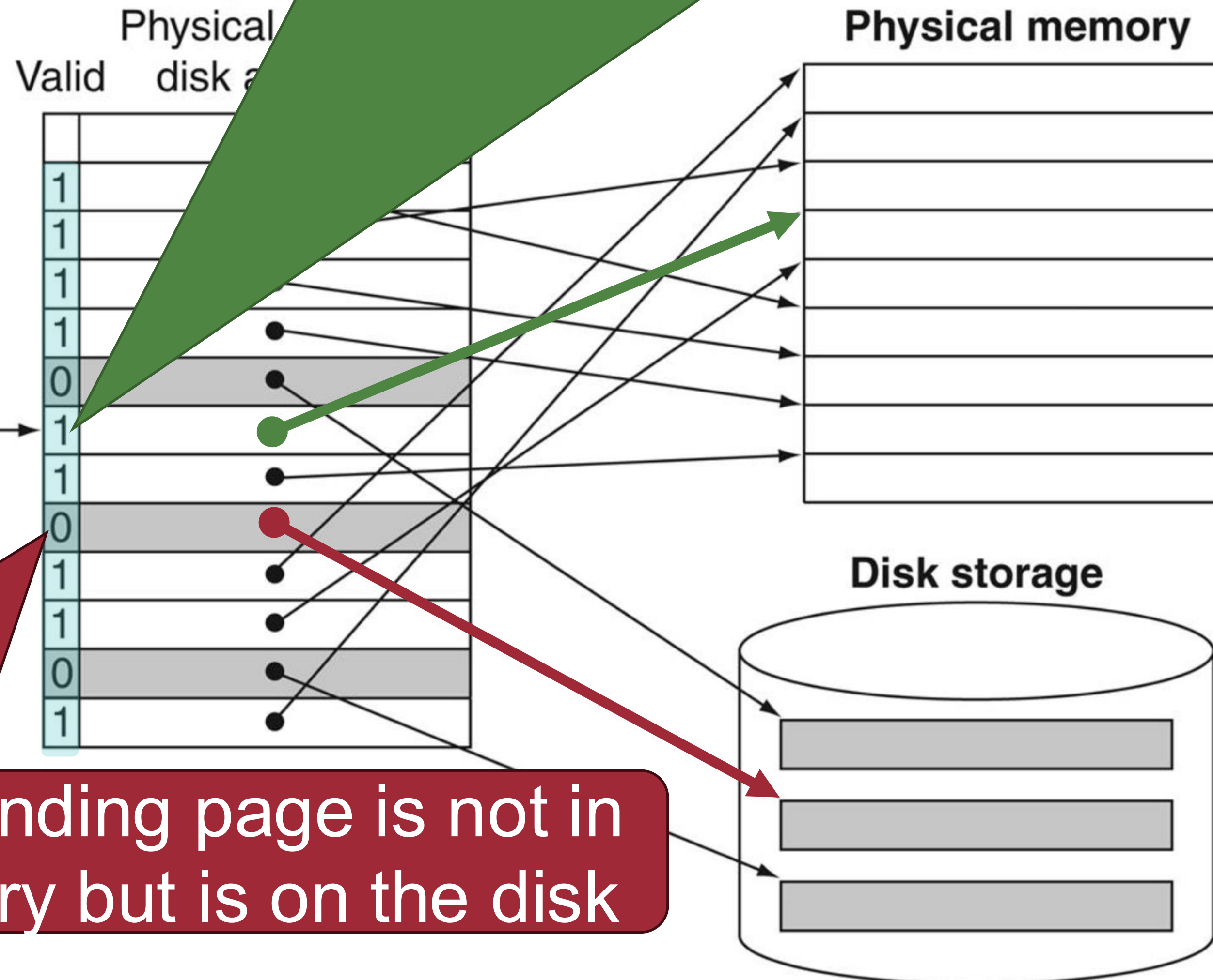


# Valid Bit in Page Table

**1:** the corresponding page is in physical memory (i.e., mapped physical address exists in the entry)



# Valid Bit in Page Table




**1:** the corresponding page is in physical memory (i.e., mapped physical address exists in the entry)

**0:** the corresponding page is not in physical memory but is on the disk

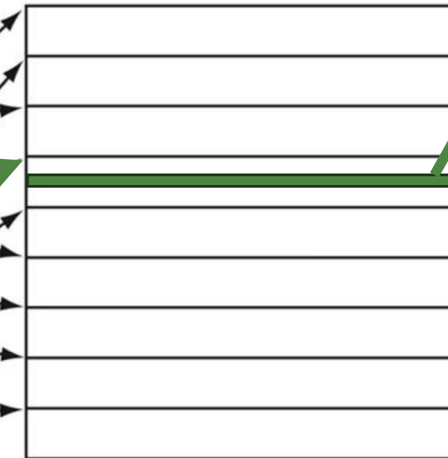
# Virtual Memory Hit

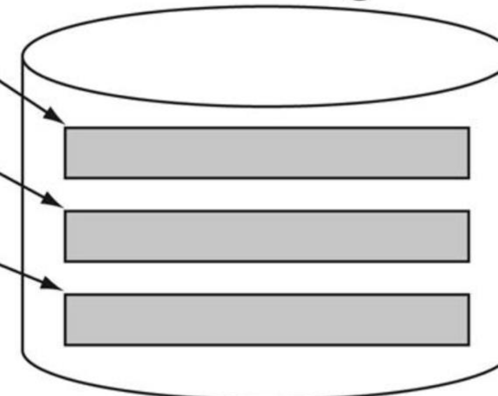
Virtual page number

#5

Page table

| | Valid | Physical page or disk address |
|---|---|---|
| | | |
| #0 | 1 | • |
| #1 | 1 | • |
| #2 | 1 | • |
| #3 | 1 | • |
| #4 | 0 | • |
| #5 | 1 | • |
| #6 | 1 | • |
| #7 | 0 | • |
| . | 1 | • |
| . | 1 | • |
| . | 0 | • |
| #N | 1 | • |

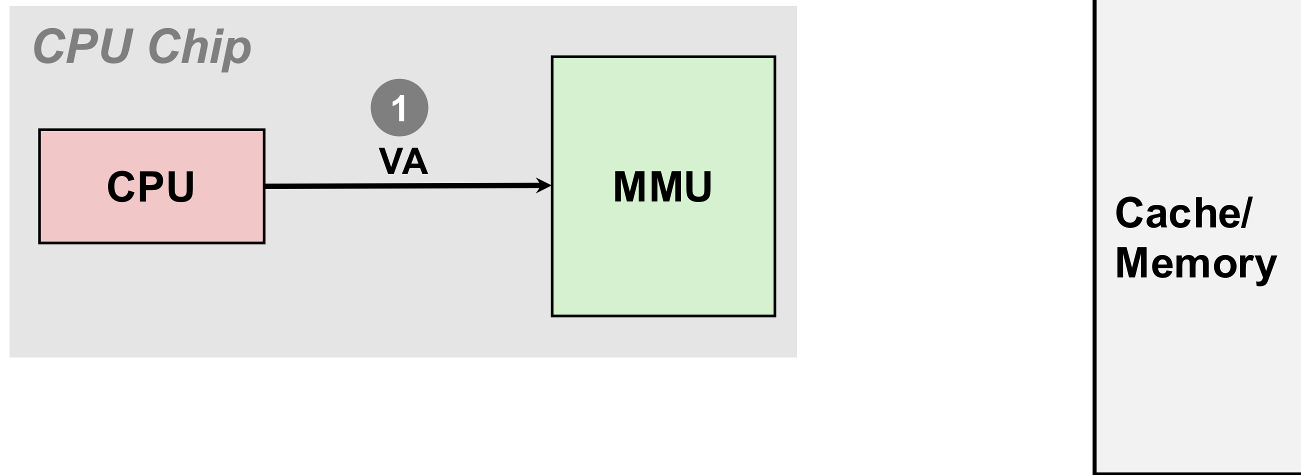



Do not need to access to the disk

# Summary+Details: Virtual Memory Hit



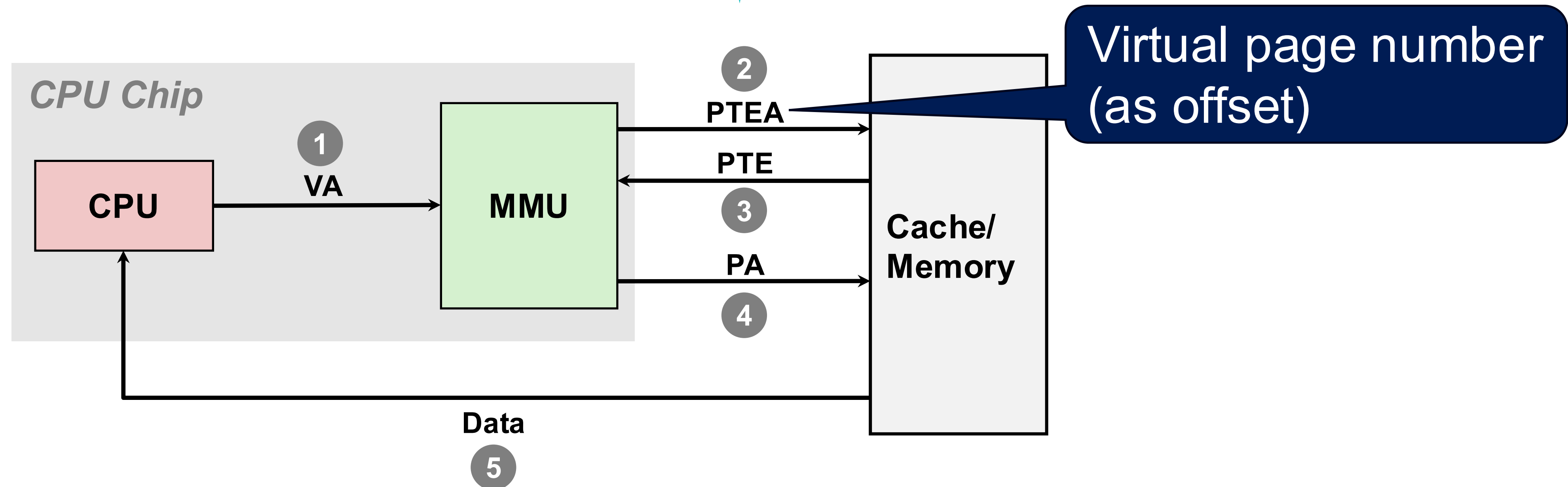



1) Processor sends virtual address to MMU

# FYI: Memory Management Unit (MMU)

A computer hardware unit that translates the virtual address being referenced into physical addresses

# Summary+Details: Virtual Memory Hit




1) Processor sends virtual address to MMU

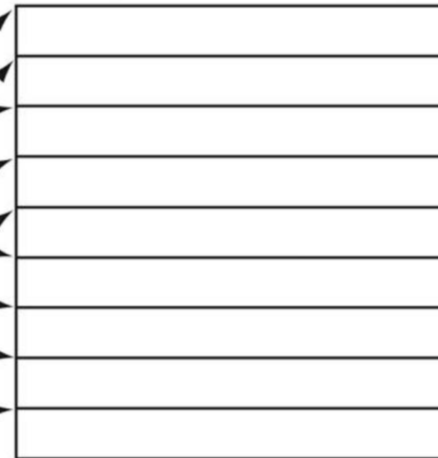
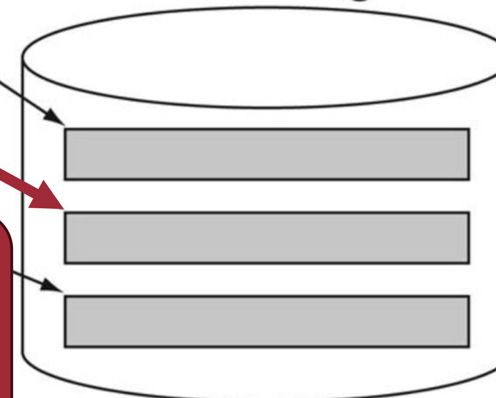
2-3) MMU fetches page tale entry (PTE) from page table in memory

4) MMU sends physical address to cache/memory

5) Cache/memory sends data word to processor

# Virtual Memory Miss (Page Fault)




**Miss (valid bit 0)**: access on memory space not in physical memory (but in disk)

# Virtual Memory Miss (Page Fault)

Virtual page number

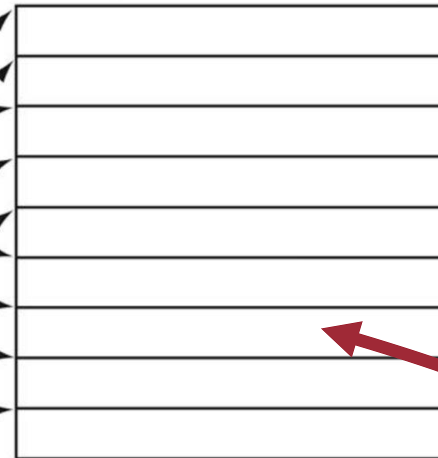
#7

Page table

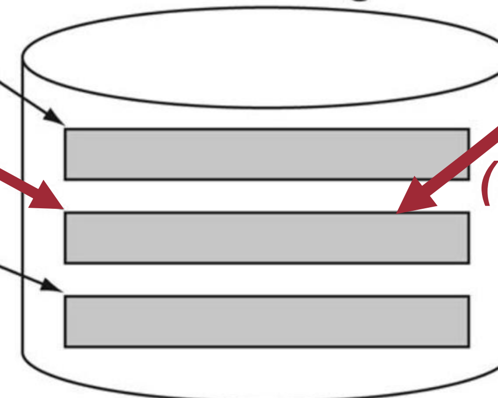
Valid | Physical page or disk address

#0 1
#1 1
#2 1
#3 1
#4 0
#5 1
#6 1
#7 0
. 1
. 1
. 0
#N 1

Physical memory



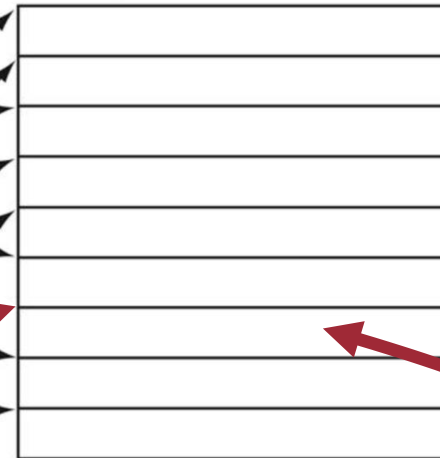
Disk storage

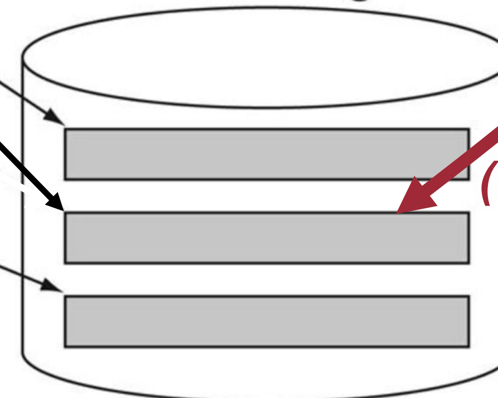


***Paging (swapping)***

***(1) Fetch "page" to the physical memory (write back depending on the situation)***

# Virtual Memory Miss (Page Fault)

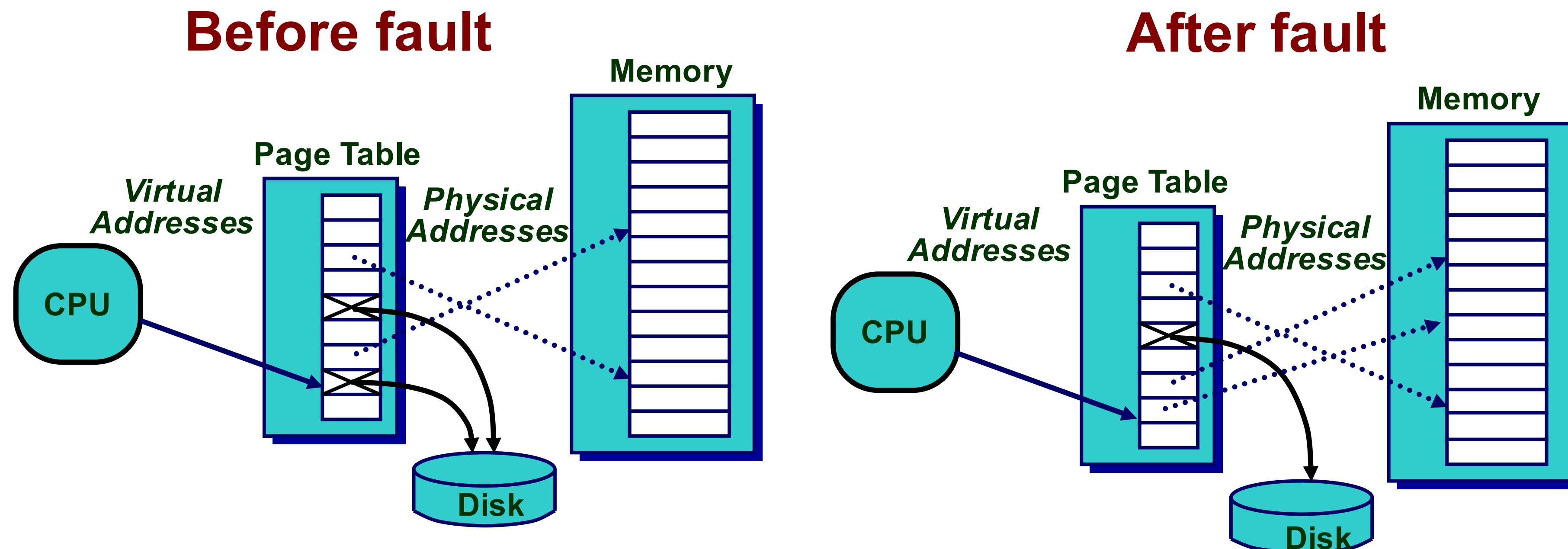


# Virtual Memory Miss (Page Fault)



- If a page is not in physical memory but disk (if valid == 0)
  - Page table entry (especially, valid bit) indicates that the page not in memory
  - Page fault exception is occurred! OS trap handler invoked to move data from disk into memory
    - OS has full control over placement!



# Page Fault Penalty

- On page fault, the page must be fetched from disk
  - Takes millions of clock cycles
- Try to minimize page fault rate and fault penalty
  - Prefer **write back** (write through is impractical)
    - **Dirty** bit in each page table entry

| Reference | Dirty | Valid | Physical page number |
| --- | --- | --- | --- |

Page table entry

# Recap: Write Policies

- Write hits
  - Cache and memory would be inconsistent! What can we do?

  **(1) Write through:** On each write hit, the information is written to both in the cache and in the memory
    - Pros: faster processing in case of 'miss'
    - Cons: make writes take longer (whenever data cache is updated, there should a memory write)

  **(2) Write back:** On each write hit, update only the block in cache. The modified cache block is written to memory ***only when it is replaced***
    - Keep track of whether each block is *dirty (additional bit)*.
    - Pros: faster processing in case of 'hit' (No repeated writes to memory)
    - Cons: slower in case of 'miss'

# Page Fault Penalty

- On page fault, the pa
  - Takes millions of cloc

- Set (1) if the corresponding page is written
- Cleared (0) when the page is replaced

- Try to minimize page fa ate and fault penalty
  - Prefer **write ba** ck (write through is impractical)
    - **Dirty** bit in each page table entry
  - Prefer **LRU** replacement
    - Reference bit in each page table set to 1 on access to page
    - Periodically cleared to 0 by OS
    - A page with **reference** bit = 0 has NOT been used recently

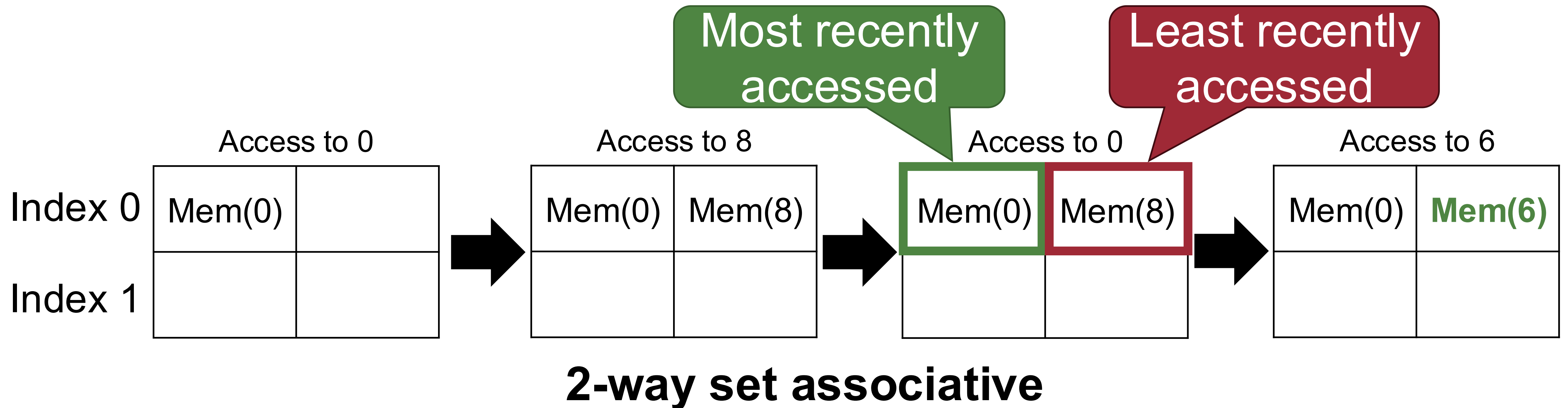
| Reference | Dirty | Valid | Physical page number |
|---|---|---|---|

Page table entry

# Recap: LRU



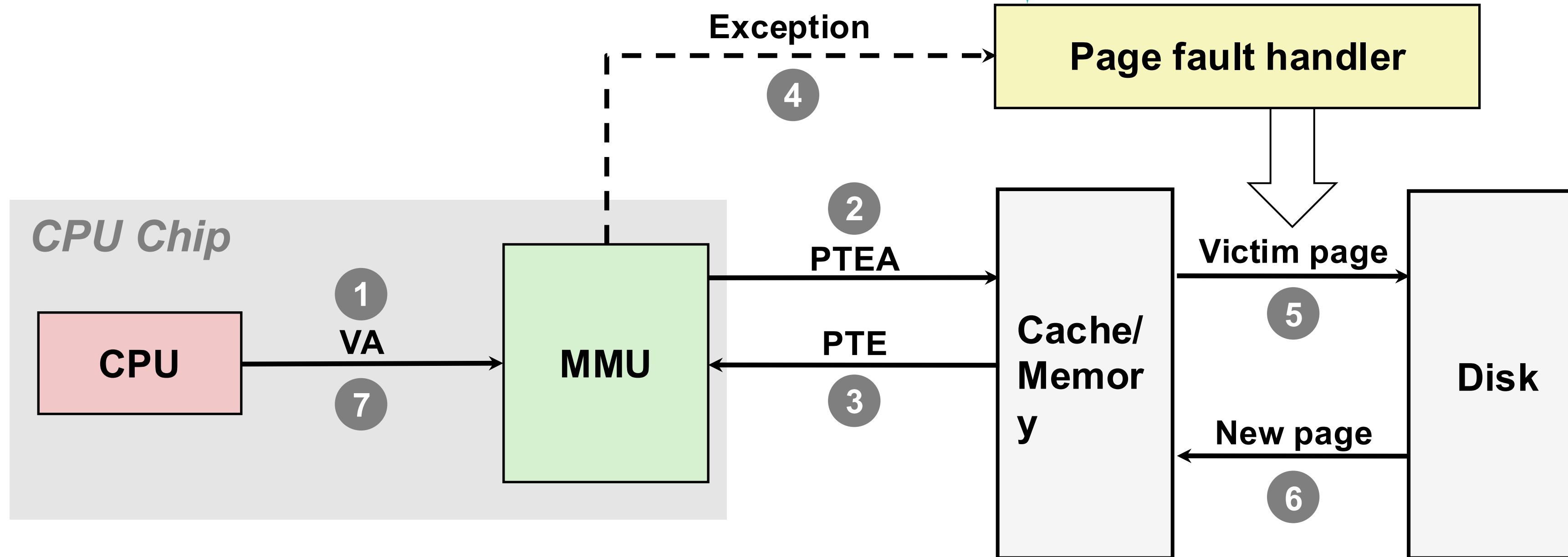
- **Least Recently Used (LRU)**: replace the one NOT used (accessed) for the longest time
  - Temporal locality of access is considered
  - Need a reference history information




**2-way set associative**

# Summary+Details: Page Fault

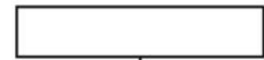





1) Processor sends virtual address to MMU
2-3) MMU fetches PTE from page table in memory
4) Valid bit is zero, so MMU triggers page fault exception
5) Handler identifies victim (and, if dirty, pages it out to disk)
6) Handler pages in new page and updates PTE in memory
7) Handler returns to original process, restarting faulting instruction

# Place of the Page Table?

Virtual page number

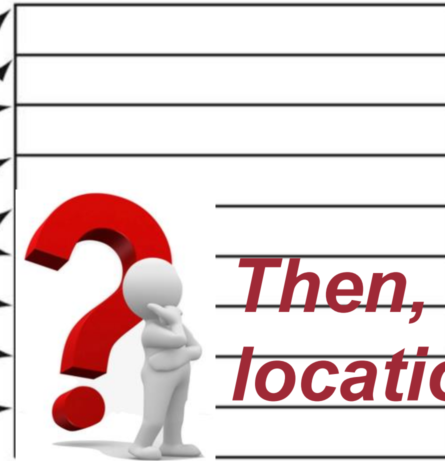


**Page table**

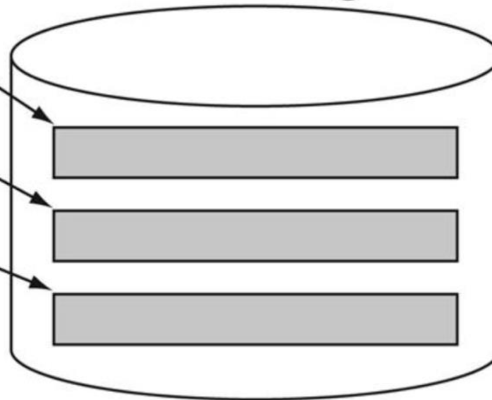
Valid | Physical page or disk address

1
1
1
1
0
1
1
0
1
1
0
1

**Physical memory**



**Disk storage**



***Where does the page table exist?***

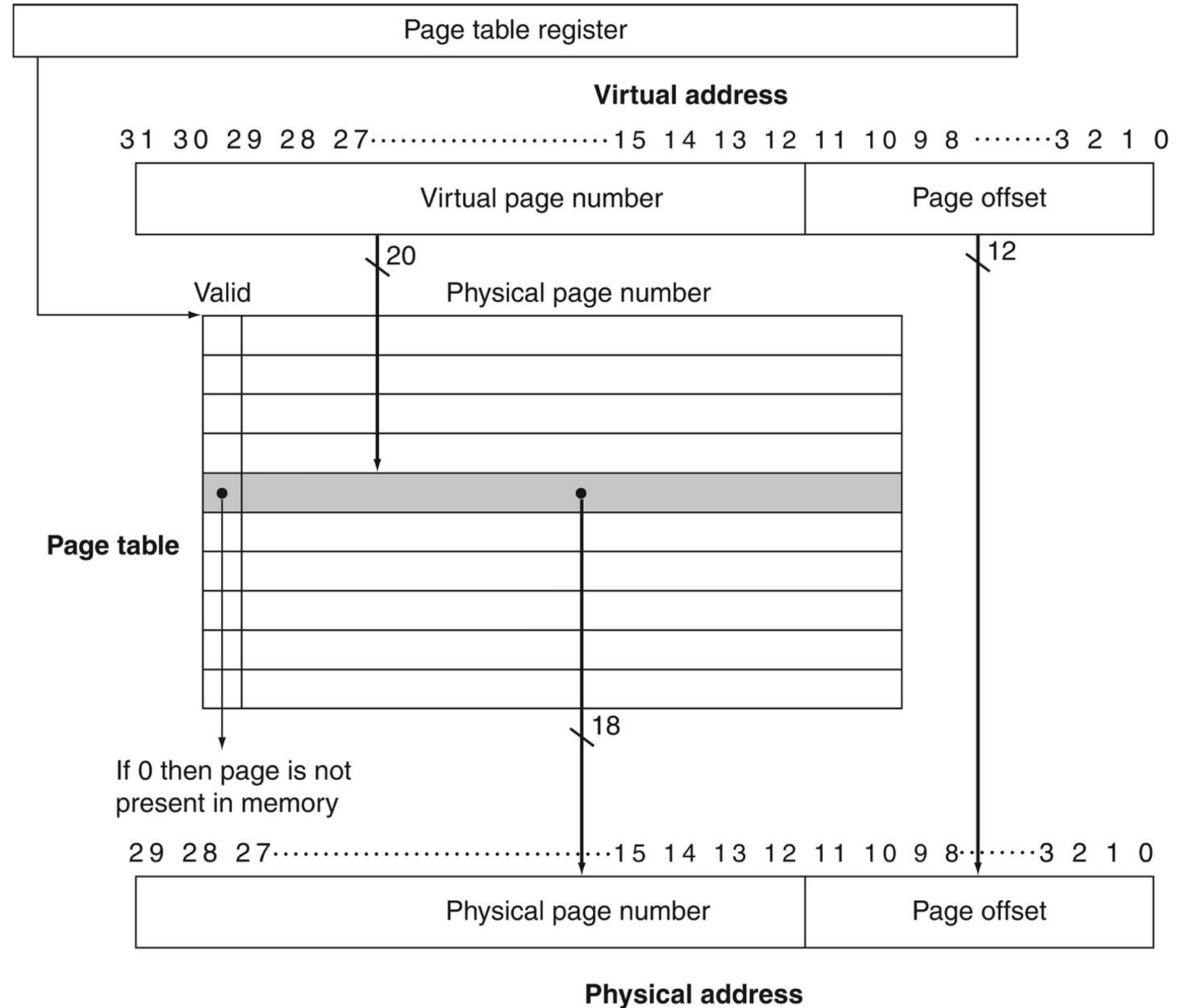
***Physical memory!***

***Then, how can we know the location of the page table itself?***

***Use a page table (base) register***

# Page Table Register





# Page Table Register



CPU register pointing to page table in physical memory (Absolute address)

**Index**: value of the page table register + virtual page number

Page table register

Virtual address

31 30 29 28 27 ········ 15 14 13 12 11 10 9 8 ········ 3 2 1 0

| Virtual page number | Page offset |
|---|---|

20

12

Valid

Physical page number

Page table

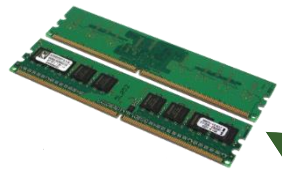
18

If 0 then page is not present in memory

29 28 27 ········ 15 14 13 12 11 10 9 8 ········ 3 2 1 0

| Physical page number | Page offset |
|---|---|

Physical address



# Question?