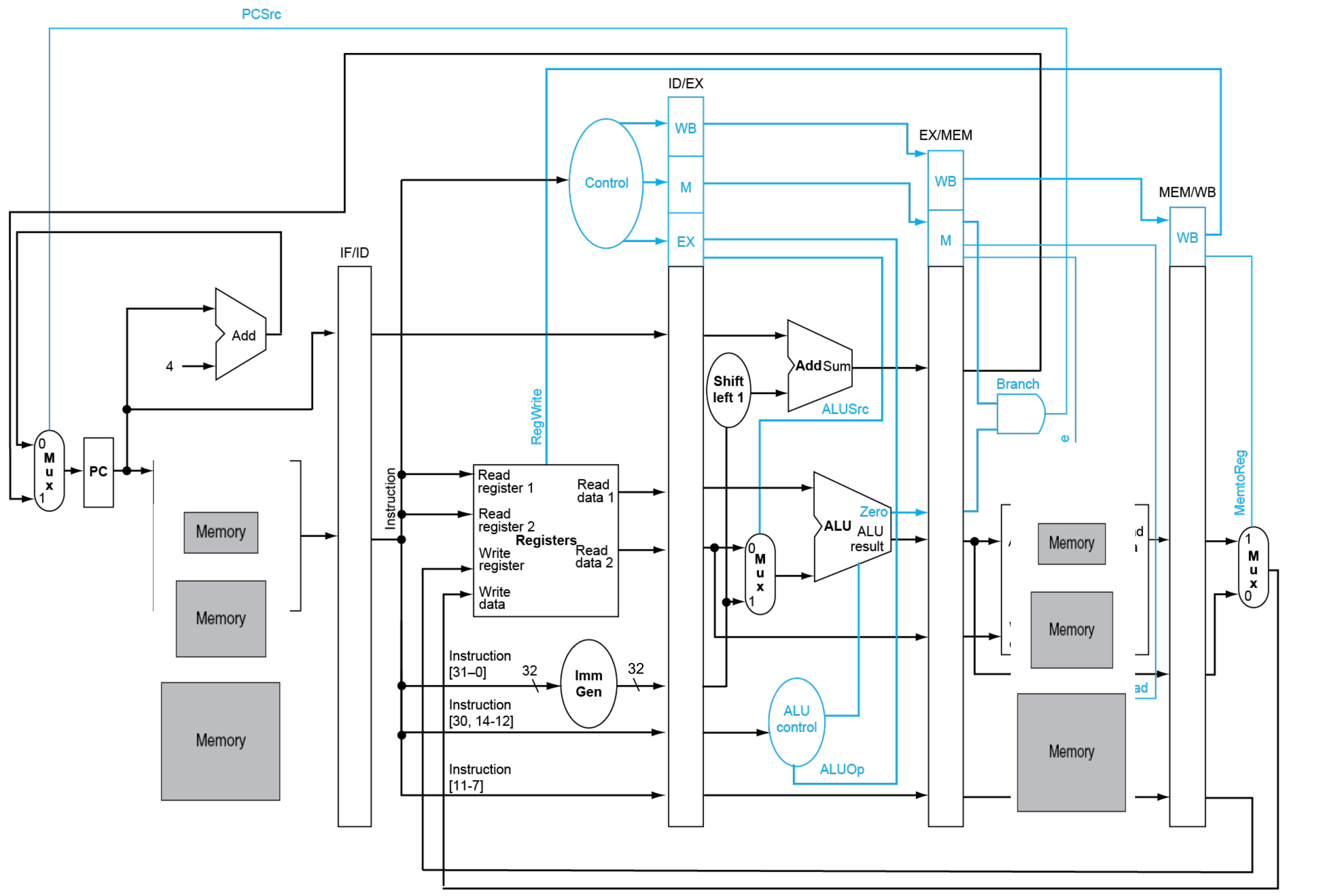
Topic 12

Memory Hierarchy

- Virtual Memory (1)

RISC-V Pipeline Architecture



Issues with Memory

- Computer may have a huge program (GByte)
  - Stored on a tera bytes of hard drive (TByte) – slow
  - But has to run on smaller cache/main memory – fast
- Computer may run multiple programs
  - Sharing the same main memory
  - We might not want them to talk to each other
- CPU interacts with main memory (through cache)
  - CPU already has many other issues, doesn’t want to be bothered by issues brought by memory

Solutions to the Issues

- Make the programmer aware of the issues
  - Write smaller program
  - Carefully allocate different main memory sections to different programs

- Well, maybe a solution decades ago!

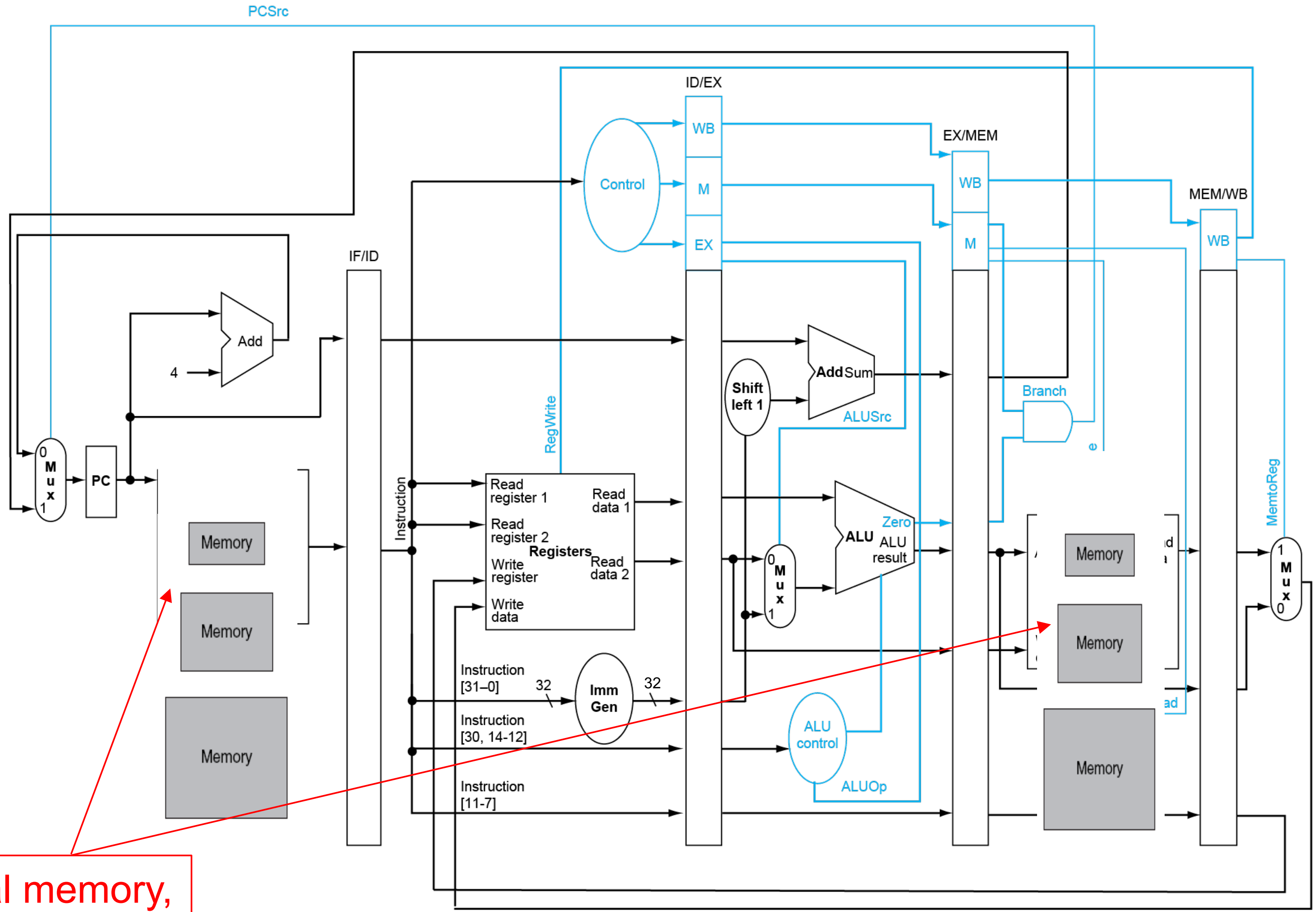
Solutions to the Issues

- Virtual Memory (VM)
  - An **imaginary, huge and fast** memory from CPU's perspective – mapped to **physical** memory
  - Each program has a virtual (memory) space corresponding to a section of physical memory on hard drive
  - Mapping is done by CPU or OS translating specific **virtual addresses** to specific **physical addresses**

What is Virtual Memory (VM)

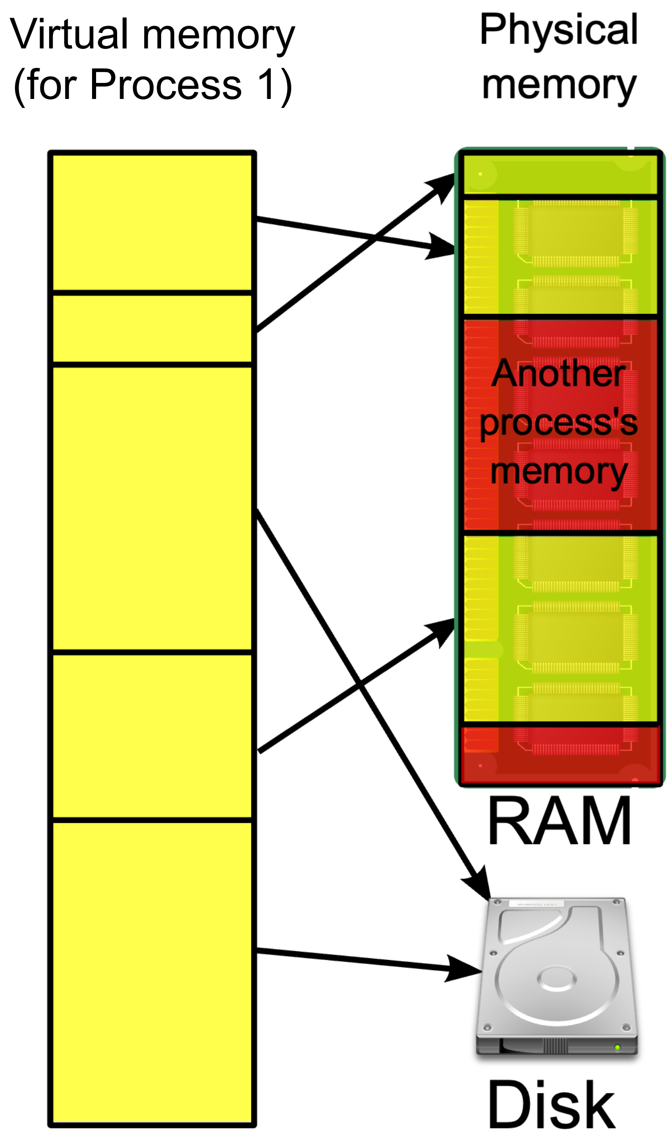
- Big
  - It can be as big as needed
  - It’s an illusion of a process
- Private
  - VM is a memory that a process owns entirely
  - Each process has a separate and private VM space holding its data and instructions
- A Cover
  - It hides constrains and complications of memory from the CPU and programmer

VM in Pipeline



Virtual memory,
Huge and fast

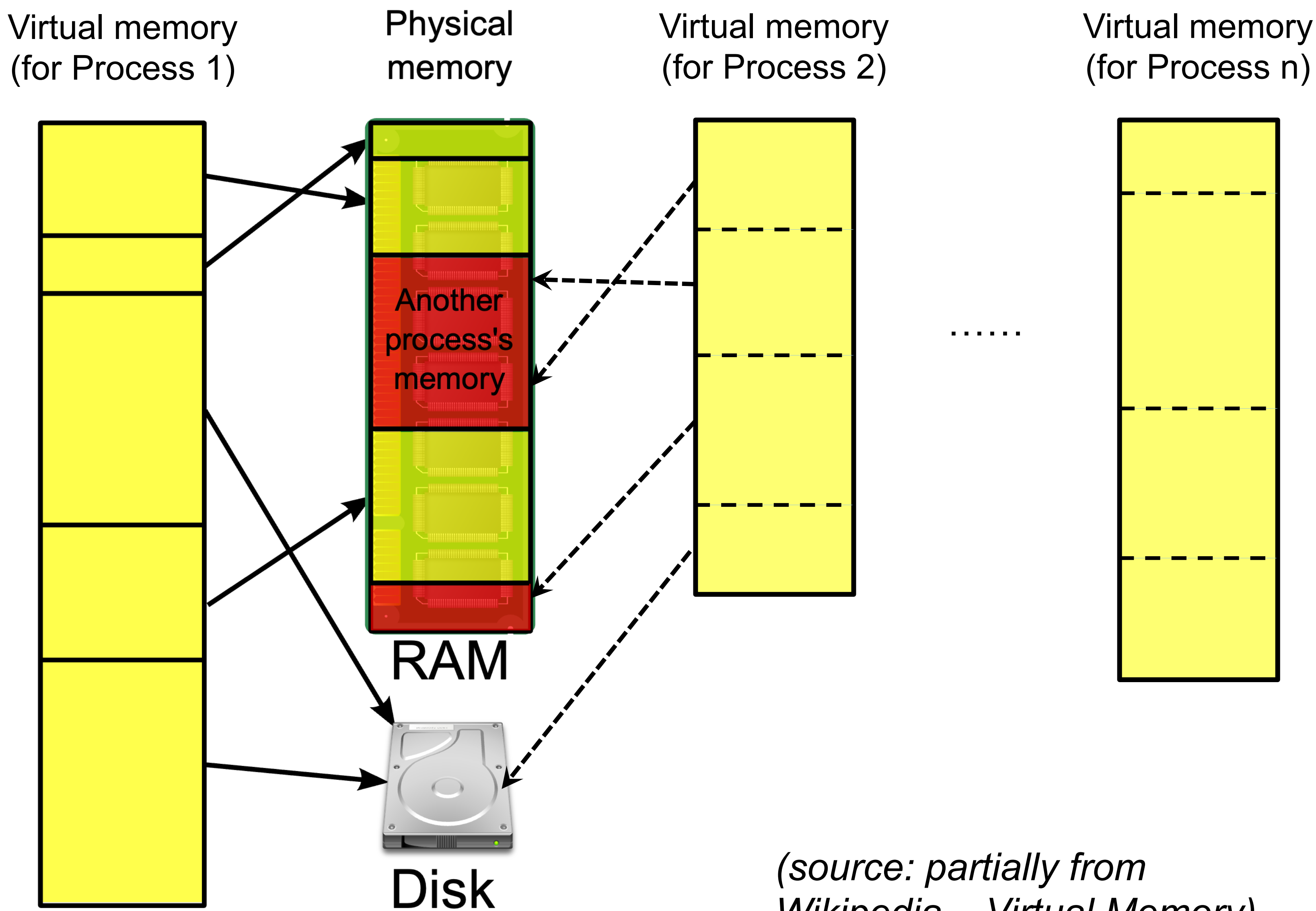
Virtual Memory is NOT Real



- Virtual Memory is an illusion from CPU's perspective, it's not real
- The imaginary memory has to be realized and supported by physical memory (cache, main memory, and hard drive)

(source: Wikipedia – Virtual Memory)

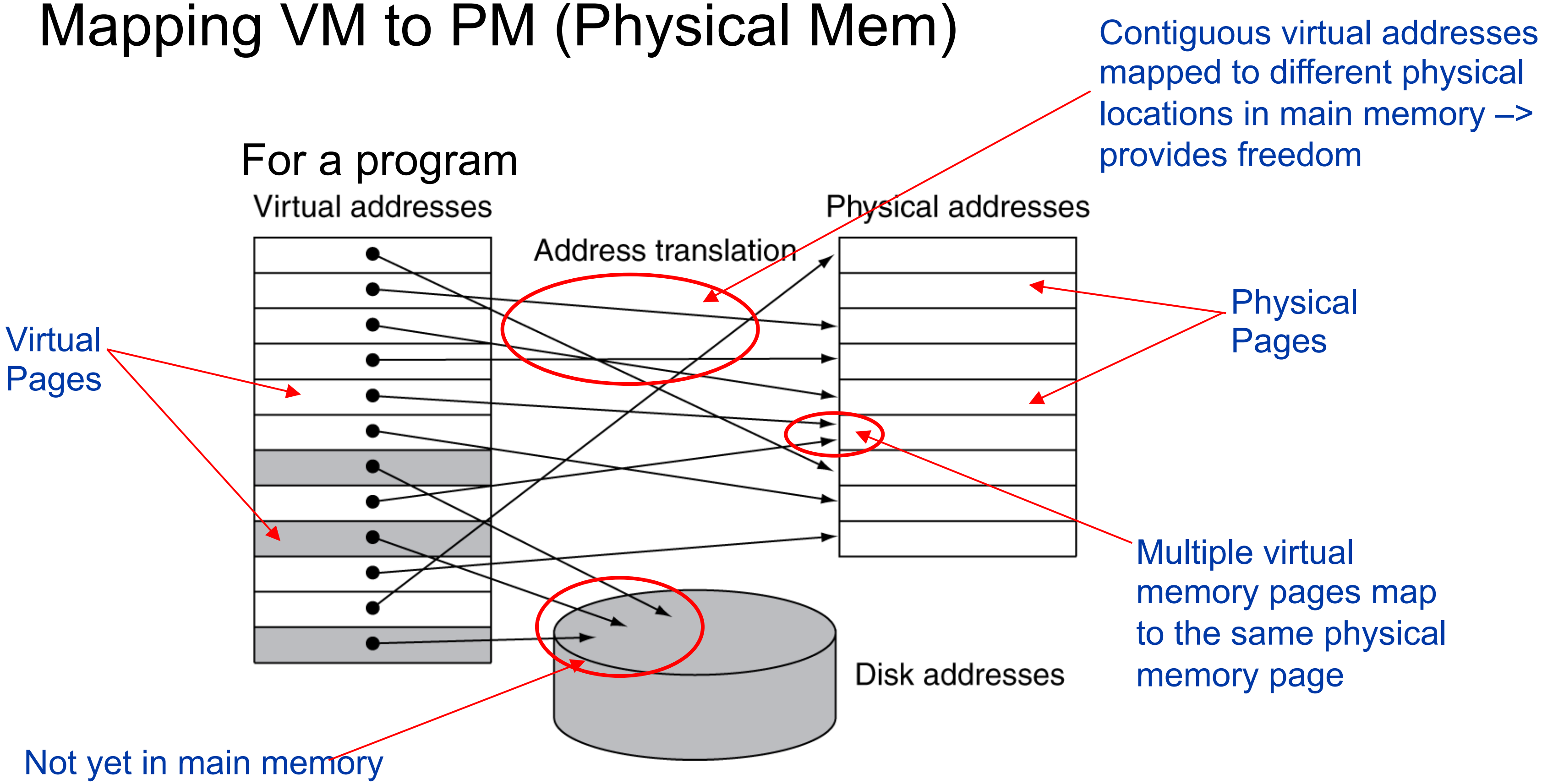
Shared Physical Memory



(source: partially from Wikipedia – Virtual Memory)

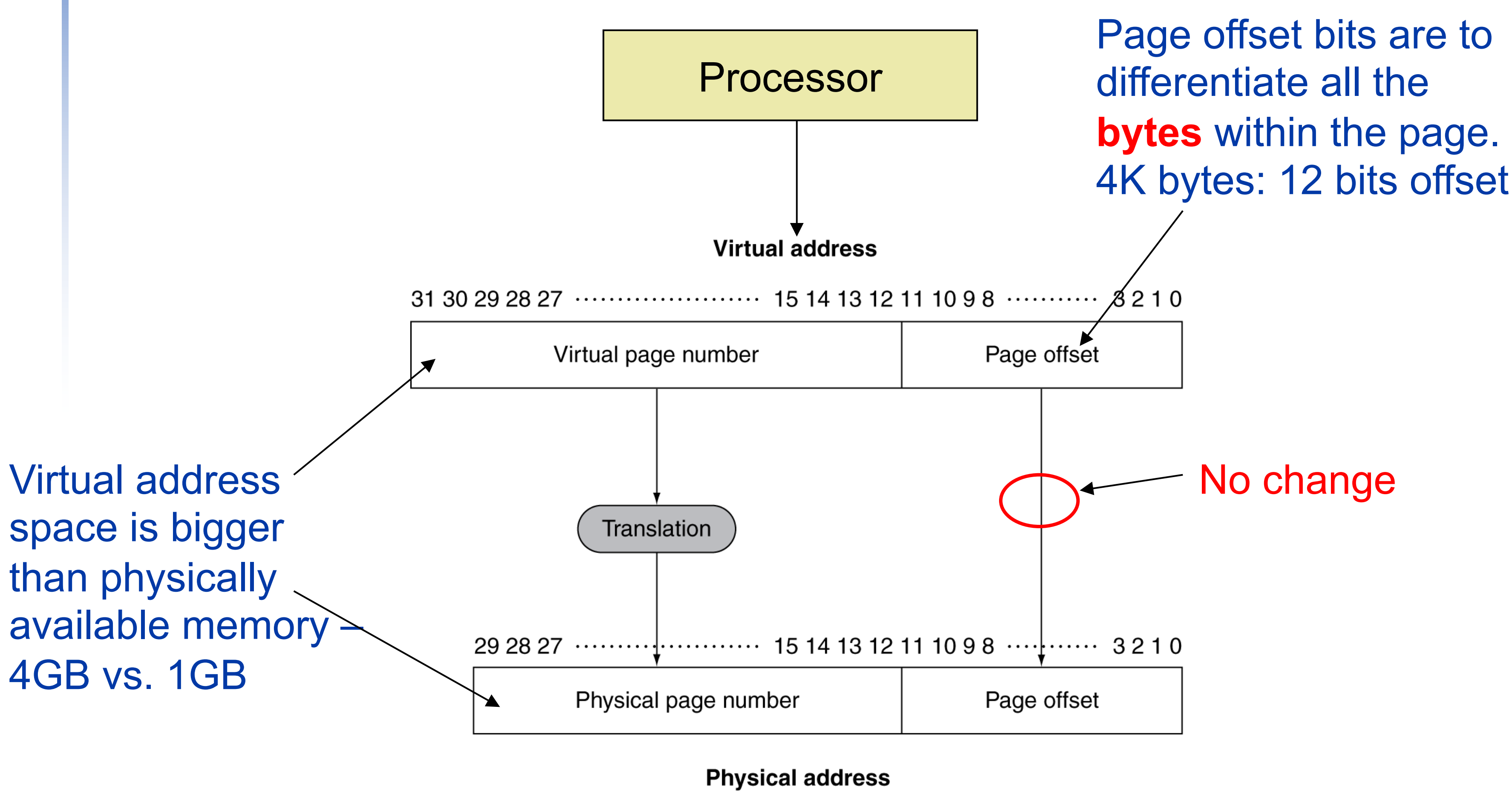
VM Terminology

- Concepts in VM context
  - Data transfer unit is now called a **page** (bigger) rather than "block"
- Mapping VM to PM (Physical Mem)



Address Translation

- Assuming fixed-size pages (e.g., 4K Bytes)



Address Mapping

Translator: Page Tables

- Each program (process) has one translator – page table
  - Stores the mapping (translation) of all virtual to physical addresses
  - Indexed by **virtual page numbers (VPN)**
  - Located in main memory
  - A **page table register (PTR) or page table base register (PTBR)** in CPU points to the beginning of page table for the program that is currently running

Page Table

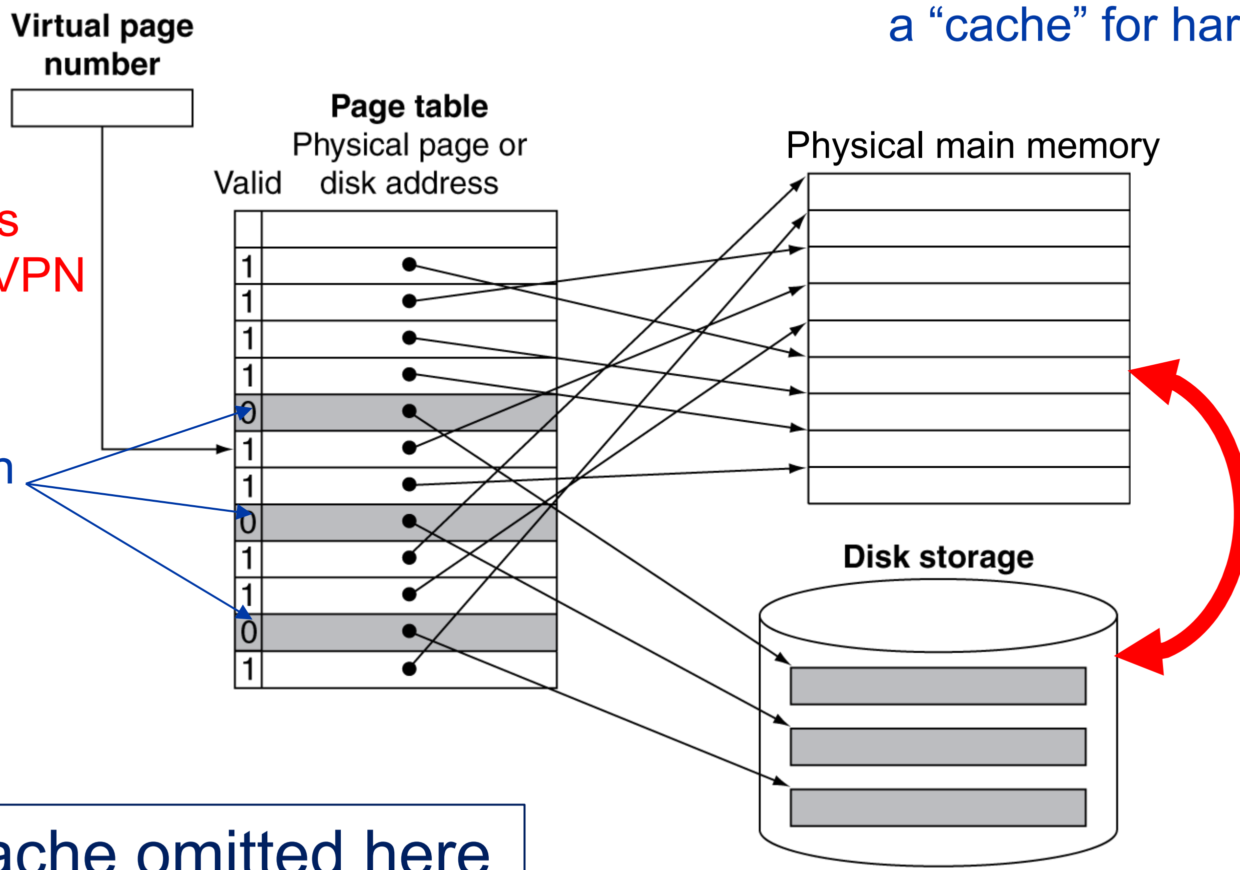
- If page is present in main memory
  - Page table stores the physical page address of the main memory
  - Valid bit is set
  - Plus other status bits (dirty, reference…)
- If page is not in main memory – **page fault**
  - It’s on hard drive (disk)
  - All virtual pages for a program are stored in a unique **swap space** on disk
  - Page table can refer to locations in the swap space of a program on disk

Mapping Pages to Storage

Physical main memory is a “cache” for hard disk

Page table is indexed by VPN

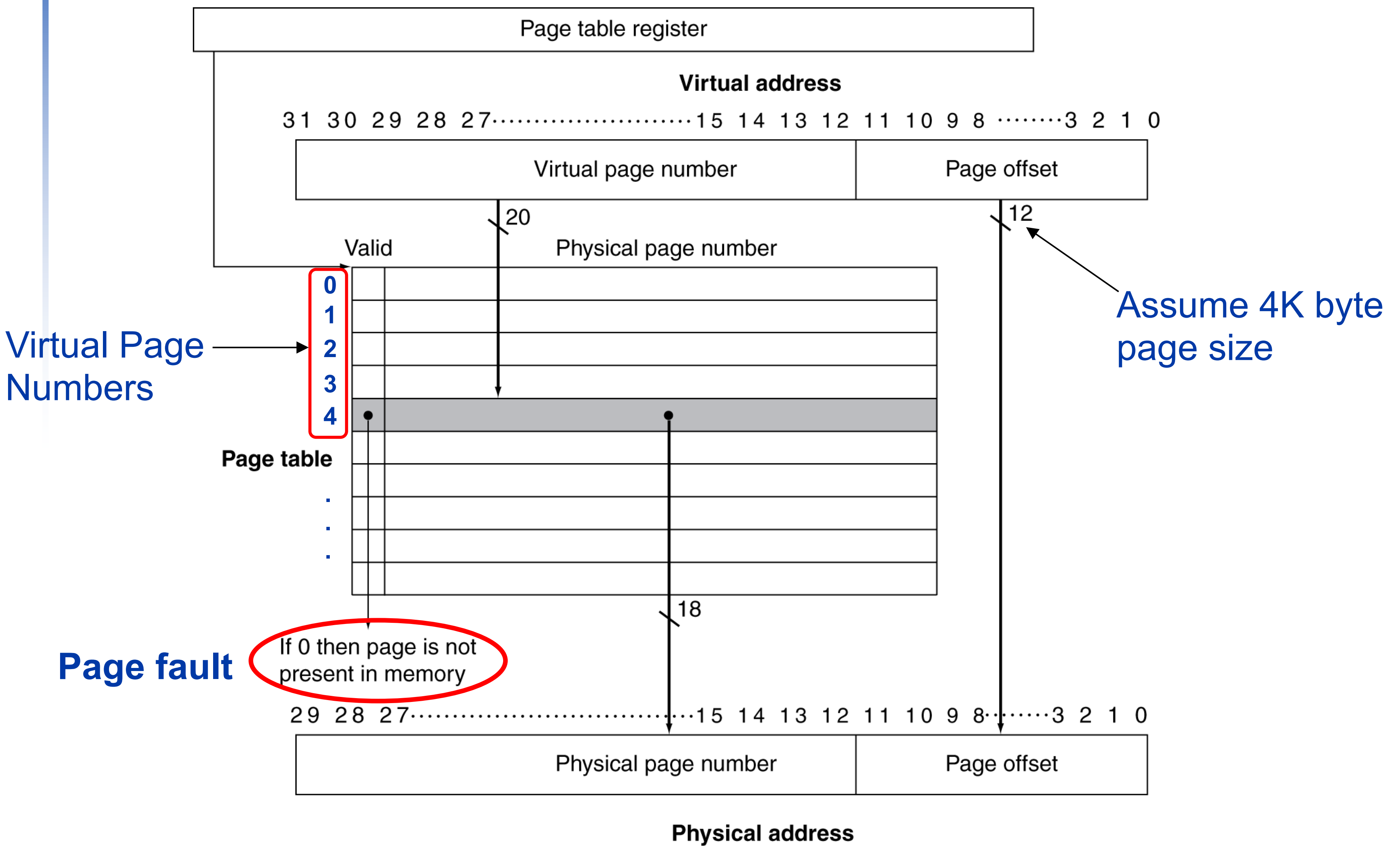
Page not in memory, page fault



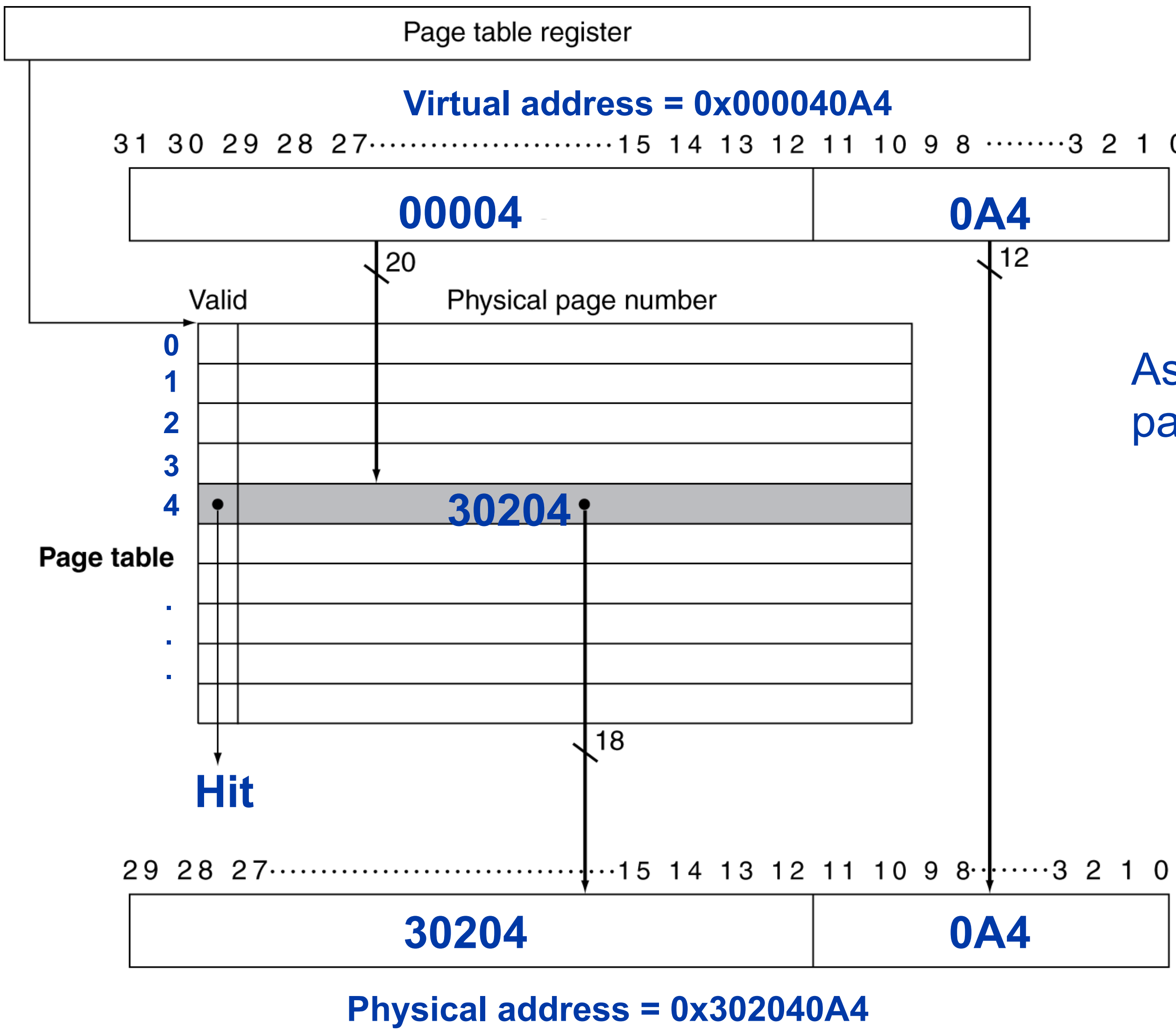
Subset

Cache omitted here for now

Translation Using a Page Table



Example: Translate Virtual to Physical



Assume 4K byte page size

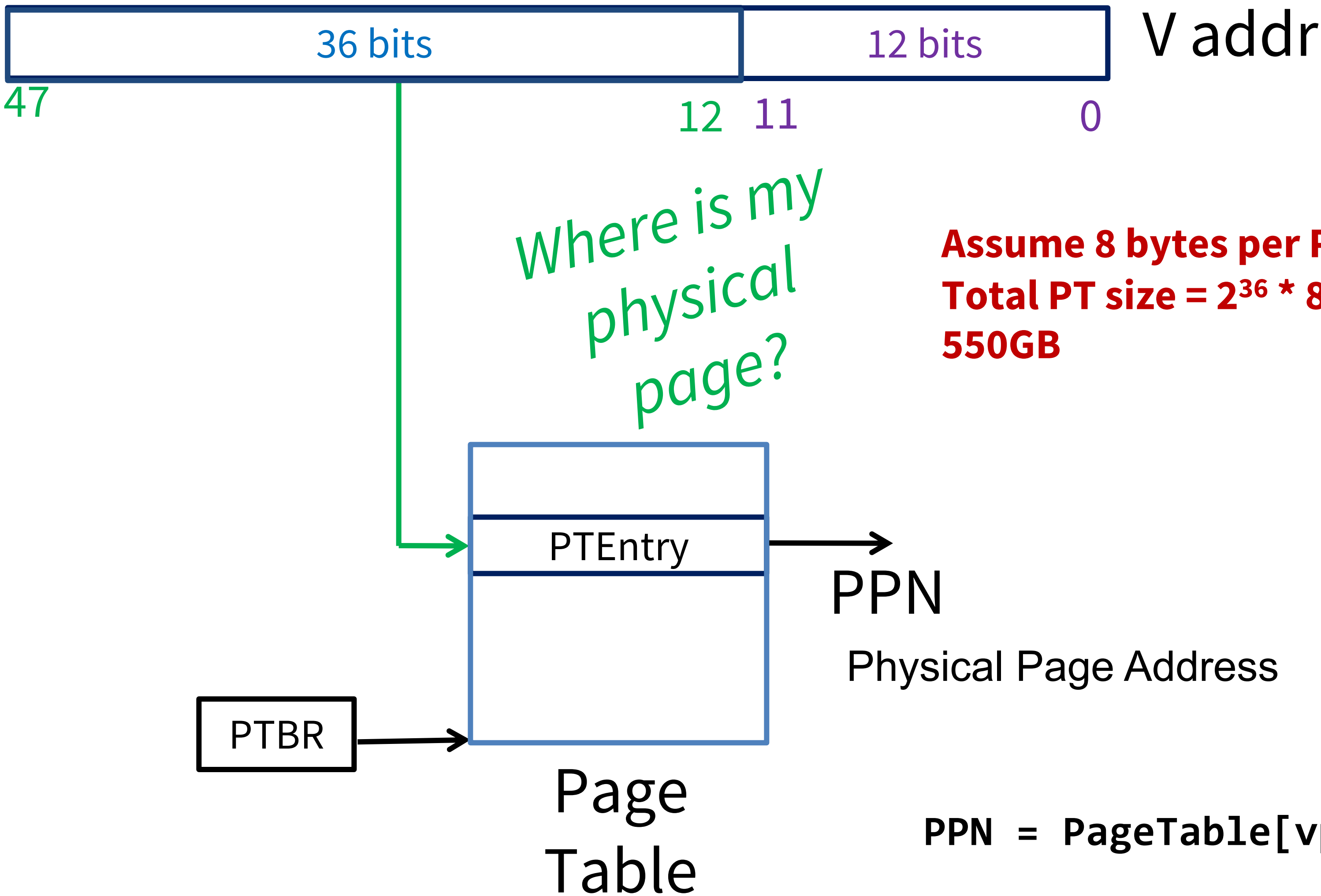
Page Table Size

- Example:
  - Page size: 4KB
  - 32-bit virtual byte address (4G Bytes)
  - 4 bytes per page table entry
- Number of page table entries = number of virtual pages = $2^{(32-12)} = 2^{20}$
  - Page table indexed by virtual page numbers
- Size of page table = number of page table entries x bytes/page table entry
  - Page table size = 2^{20} x 4 = 4 MB

Reducing Page Table Size

- Limit register
  - Restricts number of page table entries
  - Page table expands as needed
- Multiple levels of page table
  - E.g. segment table → page table
  - Total size not smaller, but non-contiguous storage for page table
- Allow page table to go virtual
  - While only having part of the page table in memory
  - The other part in disk

Single-Level Page Table

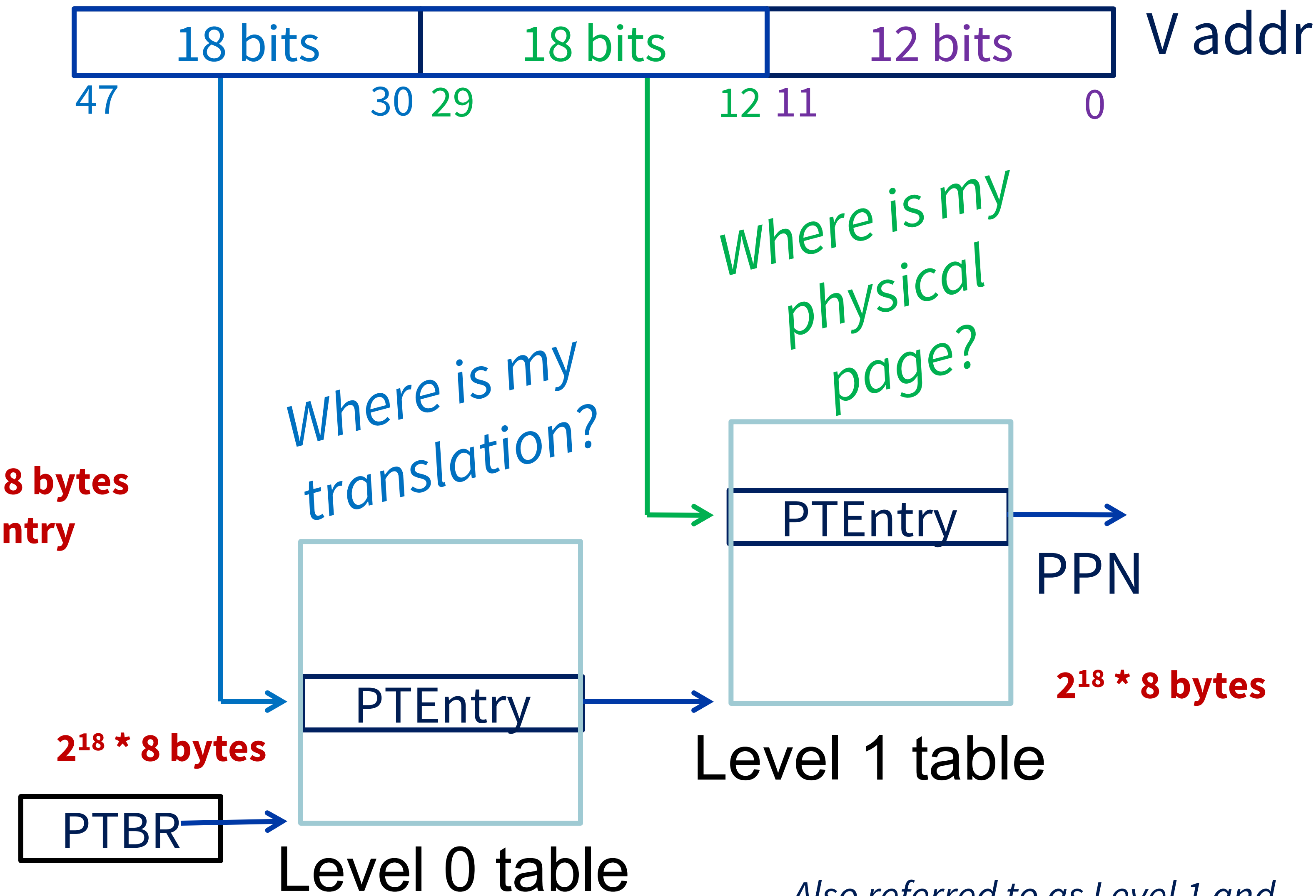


Assume 8 bytes per PT entry
Total PT size = 2^{36} * 8 bytes = 550GB

`PPN = PageTable[vpn]`

Image: cs.cornell.edu/courses/cs3410/

Multi-Level Page Table



Assume 8 bytes per PT entry

2^{18} * 8 bytes (Level 0 table)

2^{18} * 8 bytes (Level 1 table)

Also referred to as Level 1 and Level 2 Page Tables

Image: cs.cornell.edu/courses/cs3410/

Multi-Level Page Table

- Looking level 0 table, using the highest-order bits of the address -> If the address in this table is valid, the next set of high-order bits is used to index the page table indicated by the segment table entry, and so on.
- Allows the address space to be used in a sparse fashion
- Drawback - Performance: Longer lookups

Page Fault

- Page Fault
  - Valid bit is cleared
  - Requested page in virtual memory is not mapped to a page in main memory
- What should we do on page fault?

Handling Page Fault

- On page fault
  - Find the page on disk
  - Fetch and put it in main memory
- Fetching a page from disk to main memory is very expensive
  - Takes millions of clock cycles
  - Should be handled by OS – more sophisticated and less expensive
- Should try to minimize page fault rate

Reduce Page Fault Rate and Penalty

- Main memory should
  - Have **large page size**, so one access fetches more data, also reduces page fault rate
    - Most of the time is for getting the first word in the page – access time very long
    - May also reduce page fault rate
  - Reduce page fault rate by **full associativity**
  - Use **write-back**

Page Writes

- Disk writes take millions of cycles
  - Write through is impractical, even with write buffer
    - Millions of processor clock cycles
  - Use **write-back**
    - Dirty bit in page table is set when page is written
    - Write-back first if dirty bit is on
    - Writing entire page is more time efficient than writing a word
  - CPU switches to another process/program while waiting – **context switch**

Page Replacement

- A page in main memory need be replaced when the main memory is full
- Least-recently used (LRU) replacement
  - Lower page fault rate – temporal locality
  - **Reference bit** (aka use bit) in page table
    - Set to 1 on access to page
    - Periodically cleared to 0 by OS
    - A page with reference bit = 0, means it has not been used recently – to be replaced

Class Exercise

- Given
  - 4KB page size, 16KB physical memory, LRU replacement
  - Virtual address: byte addressable, 20 bits (how many bytes?)
  - Page table for program A stored in page #0 of physical memory, starting at address 0x0100, assume only 2 valid entries in page table:
    - Virtual page number 0 => physical page number 1
    - Virtual page number 1 => physical page number 2
- Show physical memory including page table
- Complete following table

Virtual Address	Virtual page number	Page fault?	Physical Address
0x00F0C			
0x01F0C			
0x20F0C			
0x00100			
0x00200			
0x30000			
0x01FFF			
0x00200			