# Virtual Memory

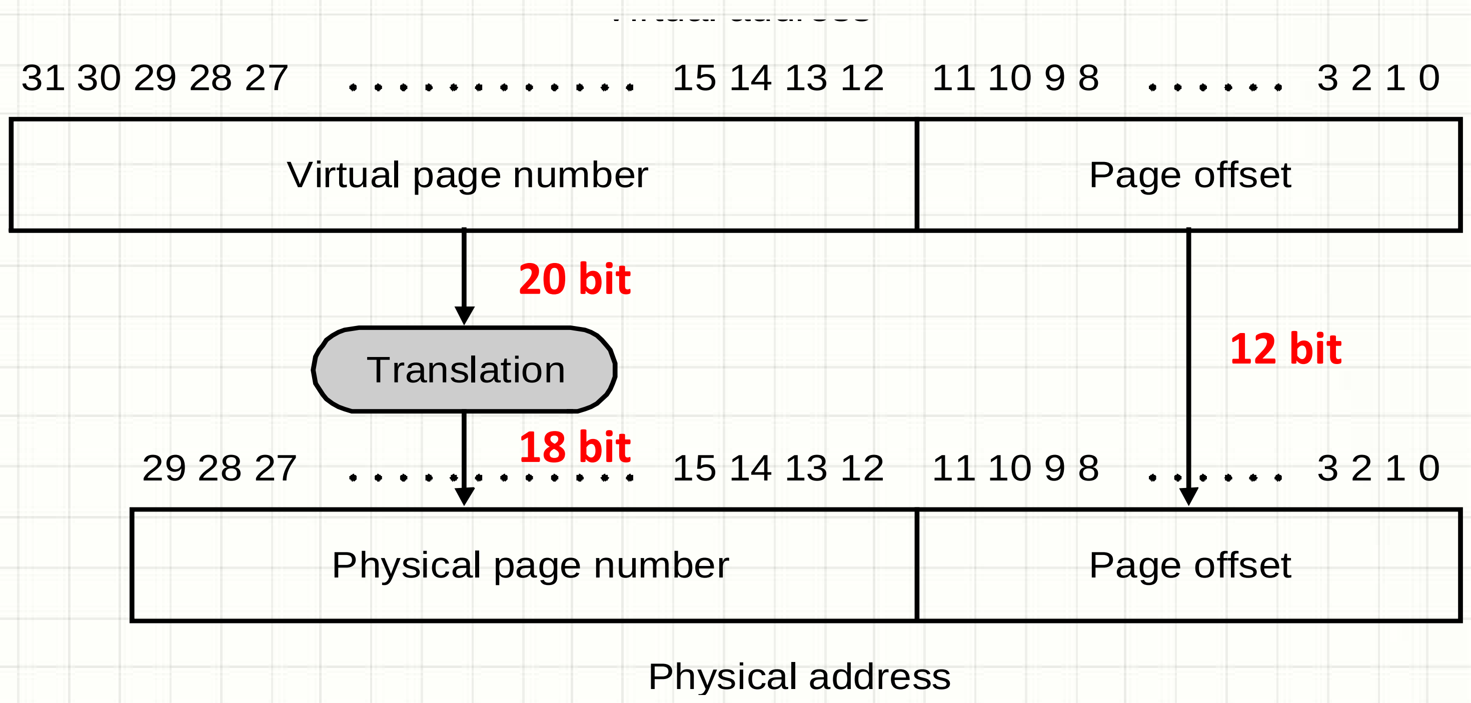
Prof. Cristina Silvano
Politecnico di Milano

## Virtual Memory

- The virtual memory is a mechanism to translate **virtual** addresses to the **physical** addresses **(memory mapping)**
- The **virtual** address space is much larger than the limits imposed on the **physical** address space by the limited amount of main memory (MM) in DRAMs.
- The total memory required by a program application may be **much larger** than the amount of MM available on the machine.
- Moreover, there **many** process applications on the machine sharing the MM
- MM contains only the active portions (called working set) of the many process applications

# From virtual to physical addresses

- The concept of **Virtual Memory** has mainly been introduced to separate and to map the virtual addressing space of the processor and the actual size of the physical memory in DRAMs

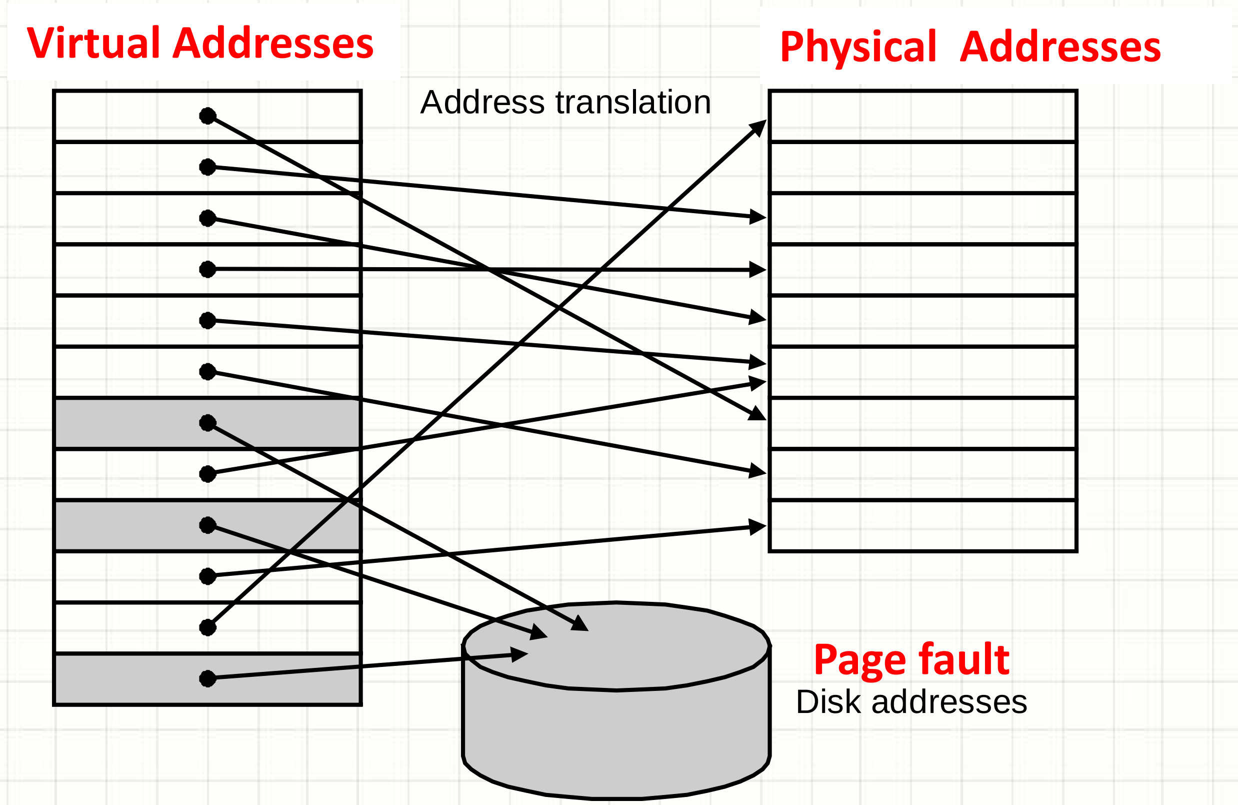


# Virtual Memory

- The virtual memory practically extends the concept of **caching:** the MM in DRAMs represents a sort of cache for the magnetic disk (secondary storage).
- Virtual memory practically treats memory as a cache for the disk, where blocks in this cache are called **"pages"** and a page miss in the MM is called a **"page fault".**
- A **page fault** happens when a virtual page has not been allocated in the MM and it resides on the disk
  - The CPU invokes the OS to activate a Page Fault Handler routine

# Virtual Memory

- The virtual memory extends the concept of **caching:** the MM DRAMs represents a sort of cache for the disk
- Virtual memory treats memory as a cache for the disk: blocks are called **“pages”** & a page miss is a **“page fault”.**

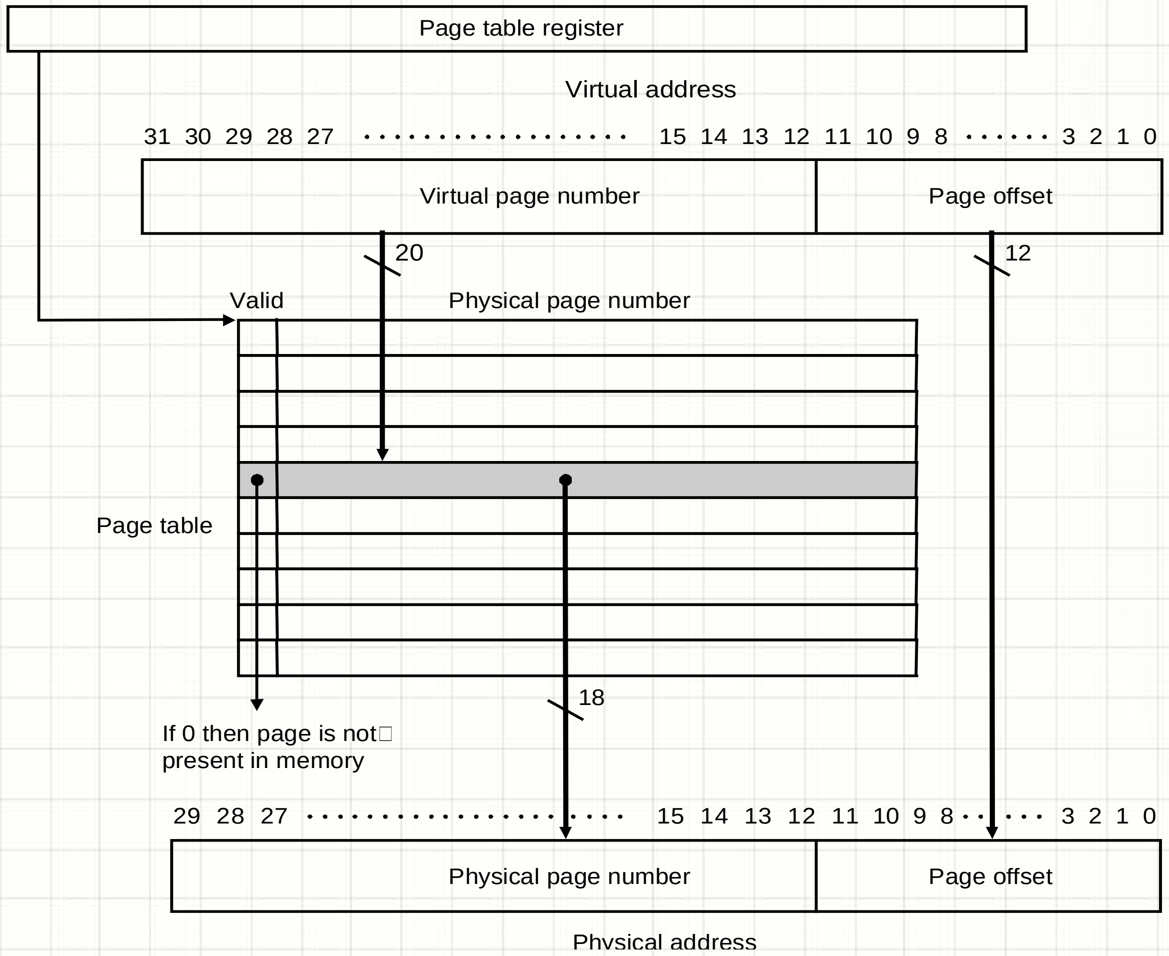


# Page Table

- The **translation mechanism** between virtual pages to physical pages is based on a **Page Table** associated to each process:
  - *Each process needs its own page table!*
- The Page Table **maps** virtual page numbers (VPNs) to physical page numbers (PPNs)
- The Page Table for each process is located in the MM
  - The location of the PT in MM is given by the **Page Table Register** that points to the start address of the PT.
- The Page Table should contain an entry for each virtual page of the process: *no tags are required!*
  - The VPN is used as **index** to access the Page Table

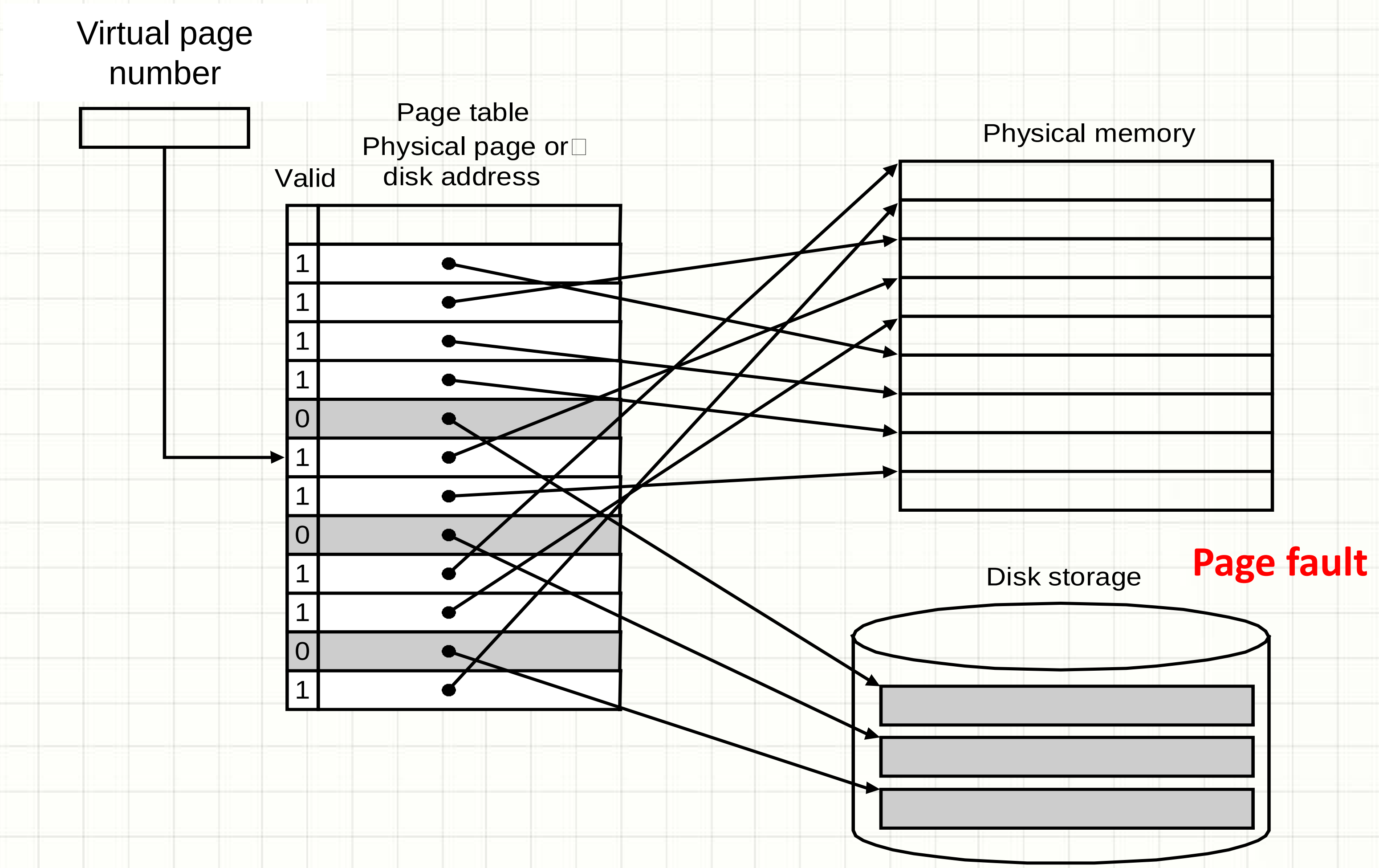
# Page Table

- The Page Table maps virtual page numbers to physical pages



# Pages out of memory: validity bit

- The Page Table requires a **validity bit** to distinguish between pages in memory and pages in disk storage

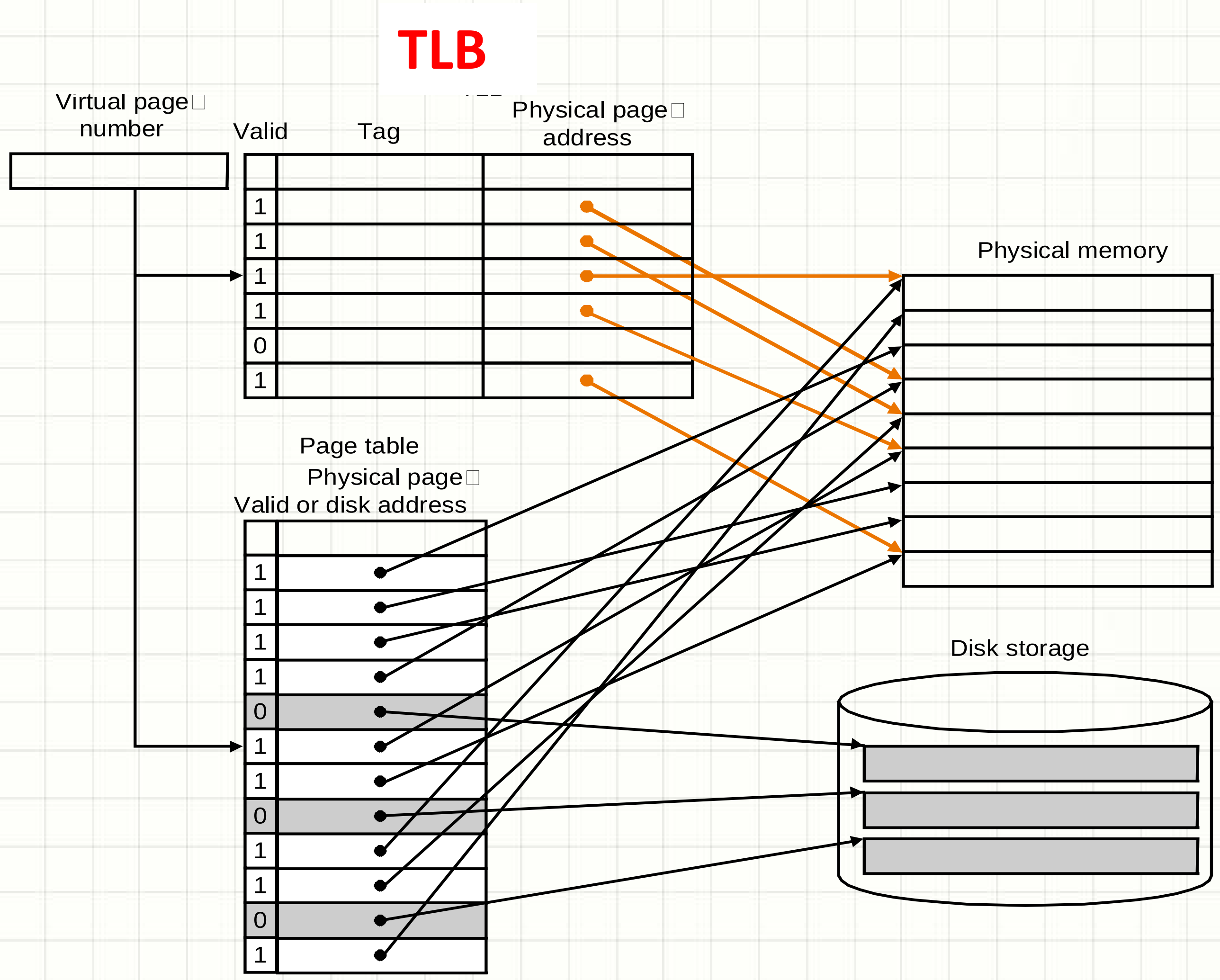


# How to make address translation fast

- Since the PT is located in MM, each memory address generated by the CPU requires 2 memory accesses:
  - First access to the PT to get the physical address
  - Second access to get the data.
- To make address translation fast we can rely on **locality of references** to the PT:
  - When we use a translation for a VPN, we will probably need it again in the near future
- **Basic idea** to speed up the address translation:
  - We add a ***special cache*** to keep track of recently used translations

# Translation Lookaside Buffer (TLB)

- Basic idea: to add a **special cache called TLB** to speed up the address translation

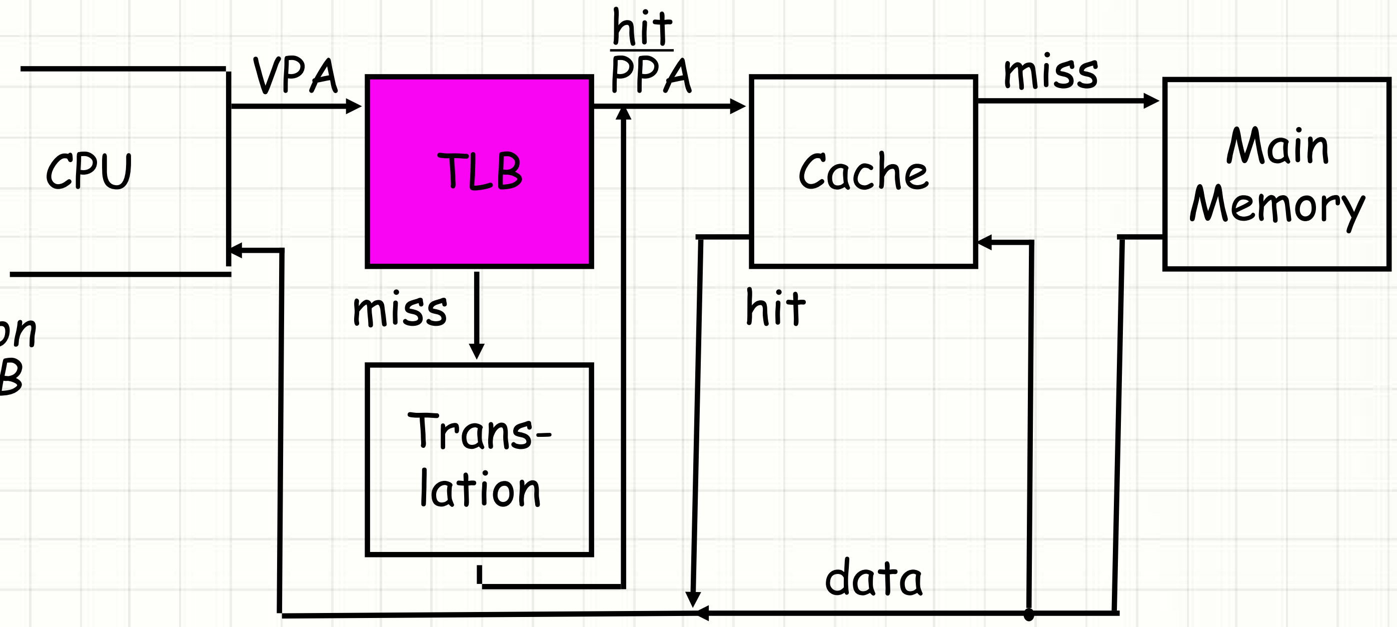


# Translation Lookaside Buffer (TLB)

- TLBs are :
  - Small caches: typically not more than 128 – 256 entries
  - Typically, fully associative caches
- If we get a TLB **hit:** the PA is used to access the MM
- A **TLB miss** maybe due to a real TLB miss or to a page fault
  - If it is a **real TLB miss:** the page exists in MM and we need to upload in the TLB the page translation from the PT.
  - It the miss is due to a **page fault:** the page is not in the MM  and the CPU invokes the OS to activate a Page Fault Handler routine.

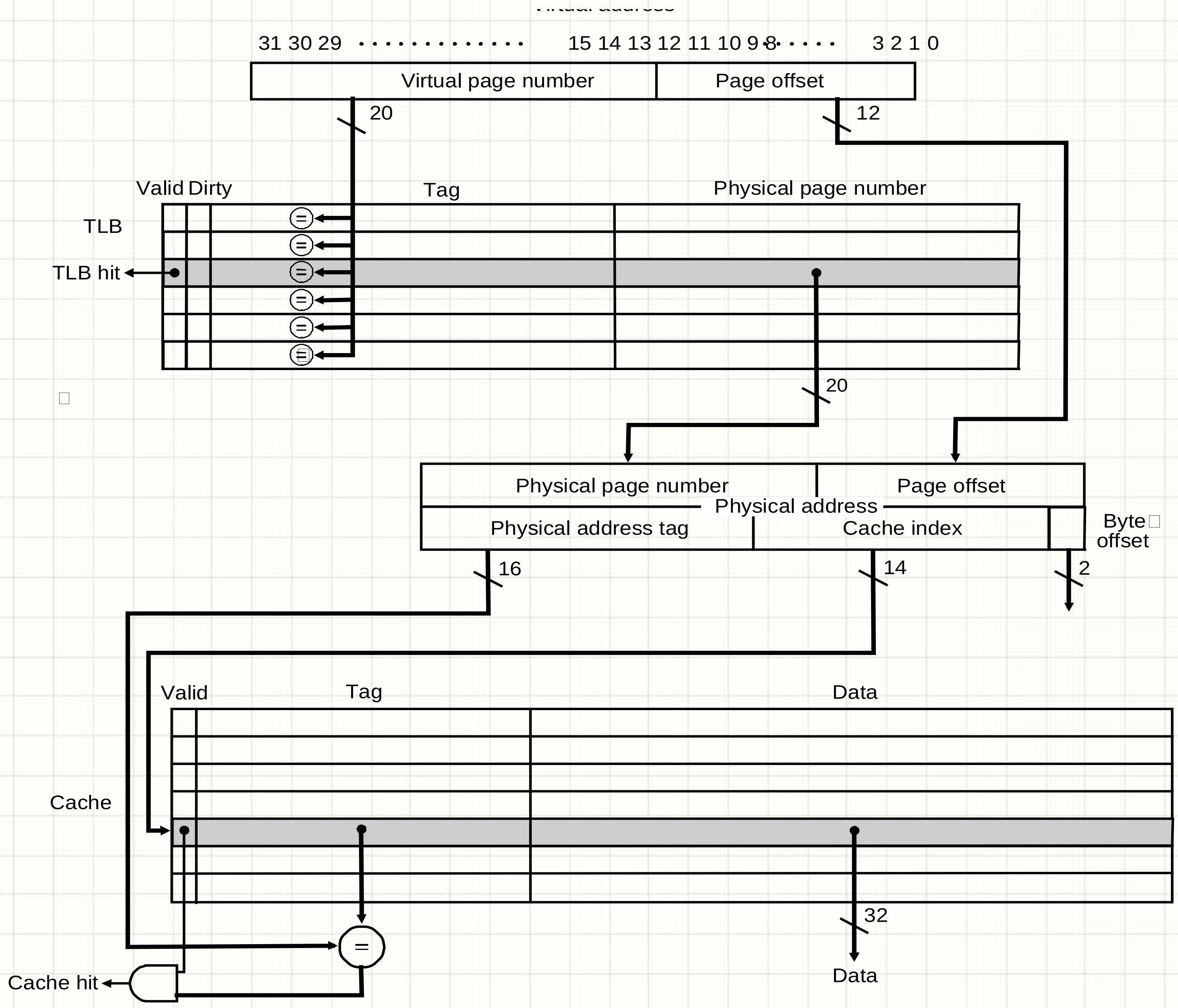
# Integrating TLB, data cache & MM

- Translation Look-Aside Buffer (TLB)
  - **Cache used on address translations**



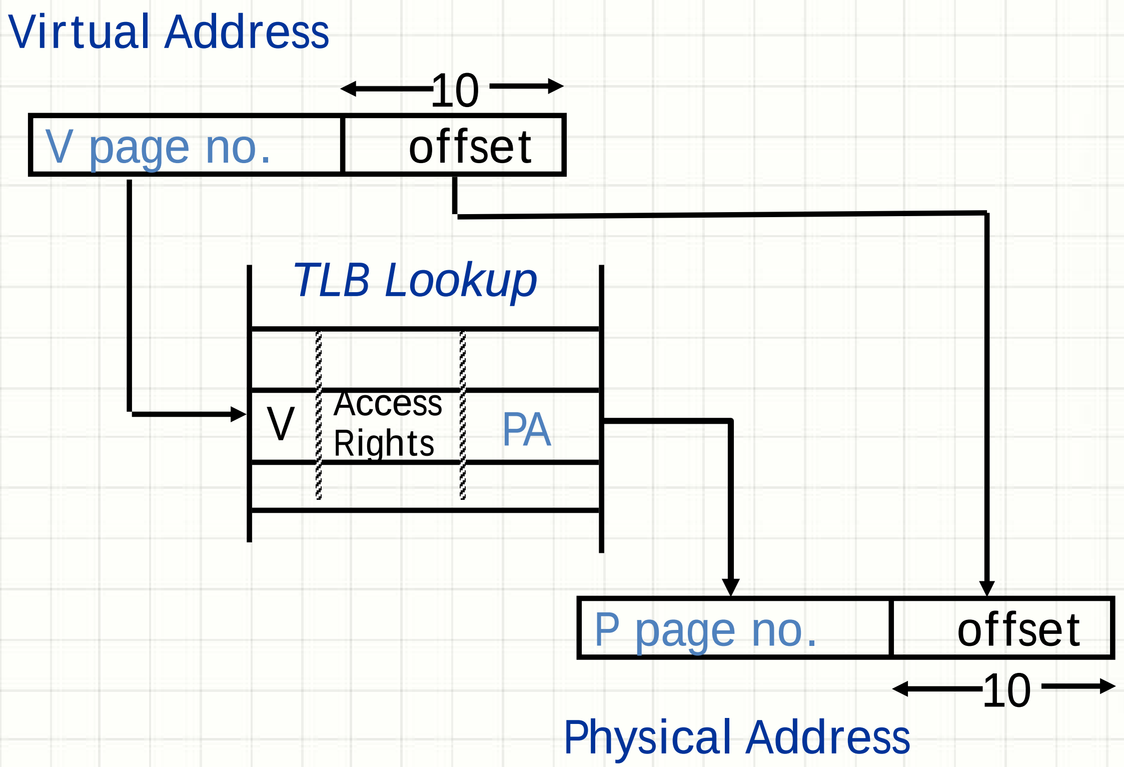
*Translation with a TLB*

# TLB and Cache



# Reducing translation time further

- As described, TLB lookup is serial with cache lookup:

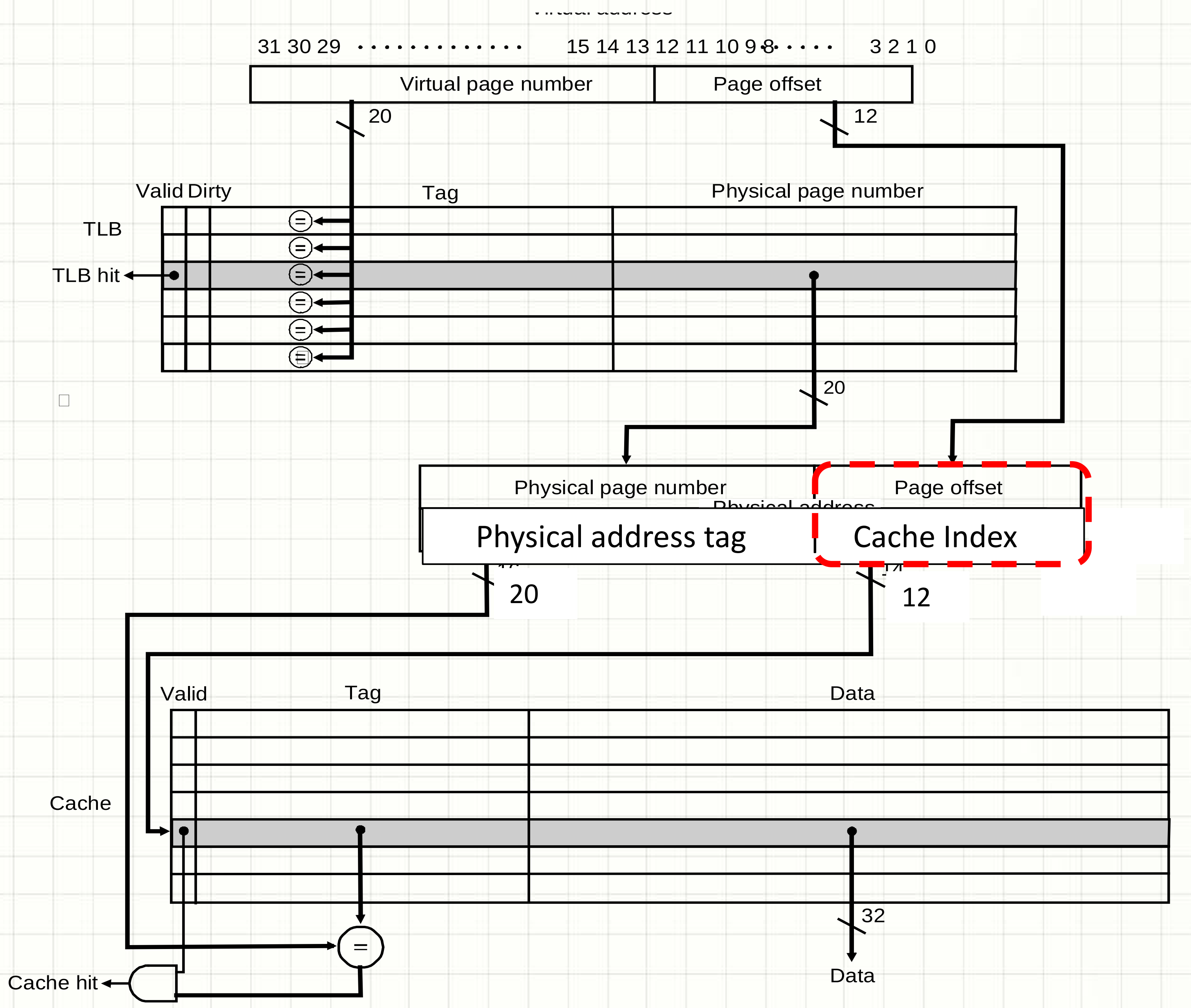


- Machines with TLBs can go one step further: they can overlap TLB lookup with cache access.
  - It works because page offset is available early

## Avoiding address translation during indexing of the cache to reduce hit time

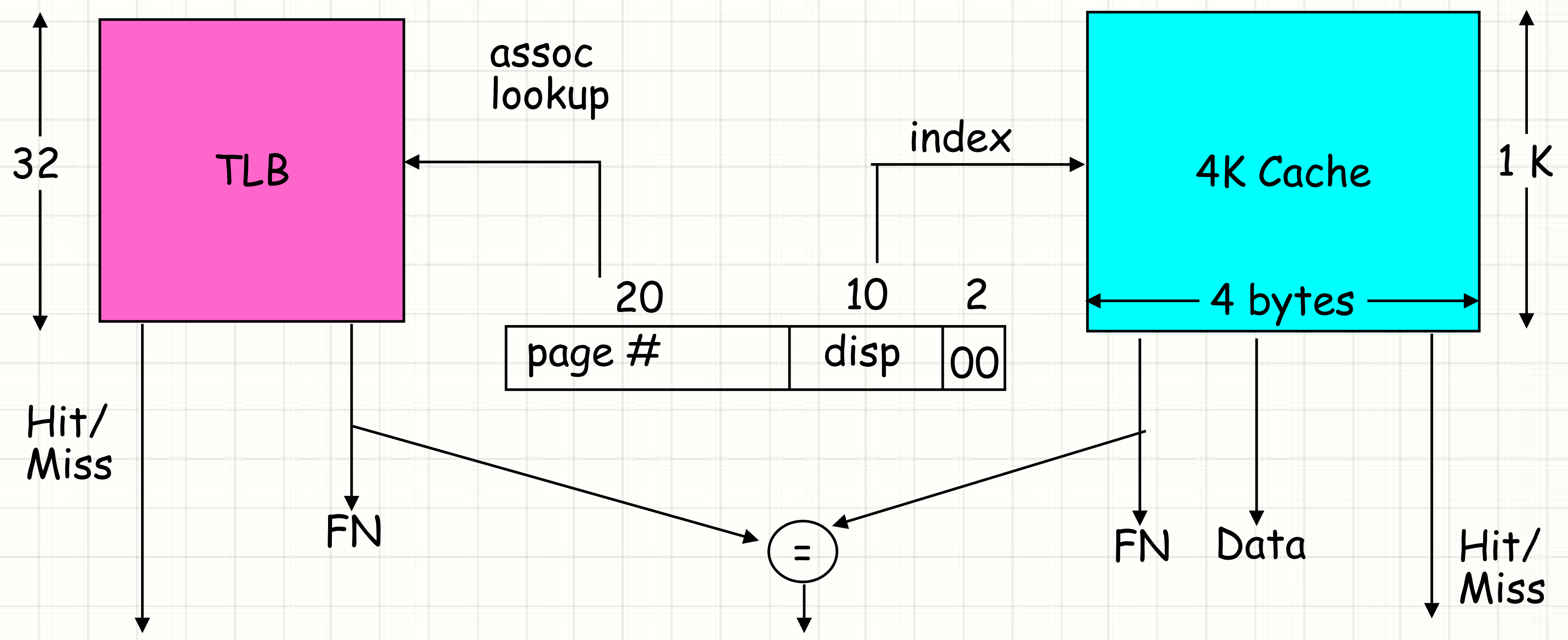
- Cache must cope with translation of virtual address from the processor to the physical address to access the memory
- Basic idea: to use **Page Offset** (the part of the address which is identical in the virtual and physical address) **as Cache Index**
- **Solution:** Cache **virtually indexed** and **physically tagged**
- Access to physical tags in cache can start immediately and in parallel with address translation so as to speed up the comparison of  physical tags of cache

# Overlapping TLB & Cache Access



# Overlapping TLB & Cache Access

- Here is how this might work with a 4K cache:



  - **What if cache size is increased to 8KB?**
    - Overlap not complete
    - Need to do something else

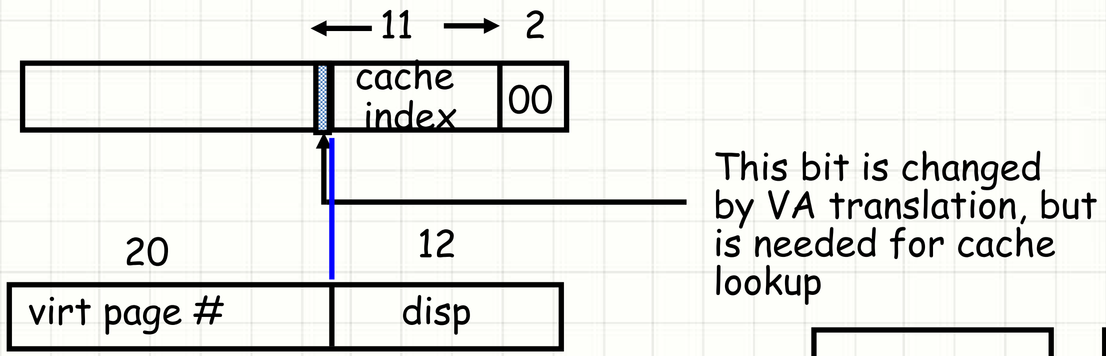
## Cache virtually indexed and physically tagged

- **Main drawback:** the size of the page limits the size for direct mapped caches
- What can we do, if we want to increase the cache size?
- **Solution:** Introduce higher associativity in the cache

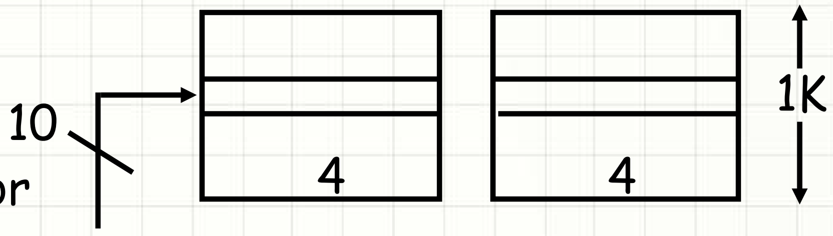
- Another option: **Virtual Caches (Virtually Addressed Caches)**
  - Tags in cache are virtual addresses
  - Translation only happens on cache misses

# Problems with Overlapped TLB Access

- Overlapped access requires address bits used to index into cache *do not change* as result translation
  - This usually limits things to small caches, large page sizes, or high
  - n-way set associative caches if you want a large cache
- Example: suppose everything the same except that the cache size is increased from 4 K bytes to 8 K bytes:


11
2
cache index
00
This bit is changed by VA translation, but is needed for cache lookup
20
12
virt page #
disp

Solutions:
go to 8K byte page sizes;
go to 2 way set associative cache; or
SW guarantee VA[13]=PA[13]



2 way set assoc cache

## Three Advantages of Virtual Memory

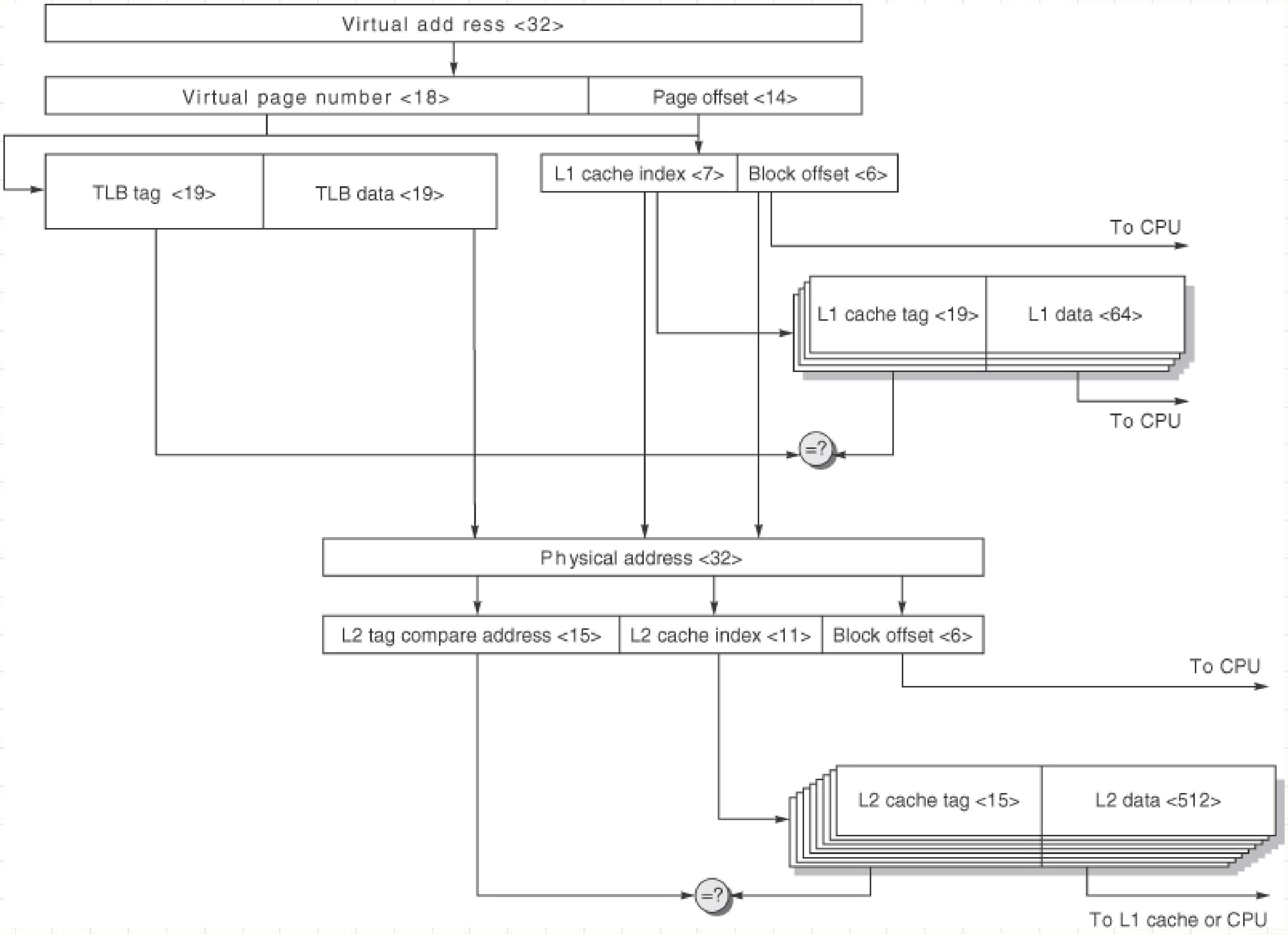
- Translation:
  - Program can be given consistent view of memory, even though physical memory is scrambled
  - Makes multithreading reasonable (now used a lot!)
  - Only the most important part of program (“Working Set”) must be in physical memory.
  - Contiguous structures (like stacks) use only as much physical memory as necessary yet still grow later.
- Protection:
  - Different threads (or processes) protected from each other.
  - Different pages can be given special behavior
    - (Read Only, Invisible to user programs, etc).
  - Kernel data protected from User programs
  - Very important for protection from malicious programs
- Sharing:
  - Can map same **physical** page to multiple users (“Shared memory”)

## Putting All Together

# ARM Cortex-A8: Data Caches and Data TLB

- ***Next page:*** The virtual address, physical address, indexes, tags, and data blocks for the ARM Cortex-A8 data caches and data TLB.
  - Since the instruction and data hierarchies are symmetric, we show only one.
- The TLB (instruction or data) is fully associative with 32 entries.
- The L1 cache is 4-way set associative with 64-Byte blocks and 32 KB capacity.
- The L2 cache is 8-way set associative with 64-Byte blocks and 1 MB capacity.
- This figure doesn’t show the valid bits and protection bits for the caches and TLB, nor the use of the way prediction bits that would dictate the predicted bank of the L1 cache.

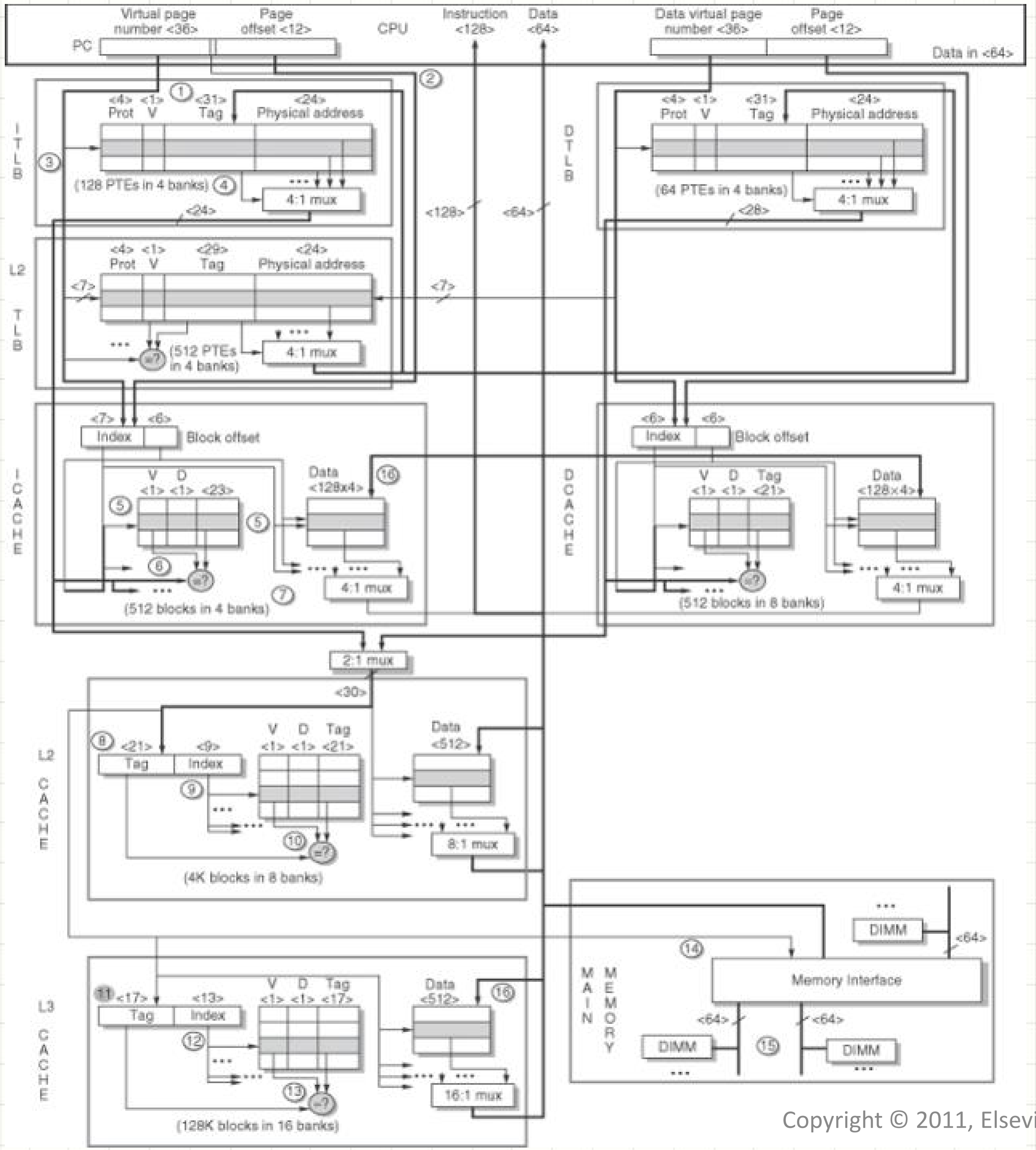
# ARM Cortex-A8: Data Caches and Data TLB



# Intel Core i7: Memory Hierarchy

- **Next figure:** The Intel Core i7 memory hierarchy and the steps in both instruction and data access.
- We show only reads for data.
- Writes are similar, in that they begin with a read (since caches are write back).
- Misses are handled by simply placing the data in a write buffer, since the L1 cache is not write allocated.

# Intel i7: Memory Hierarchy



CPU
PC
Virtual page number <36>
Page offset <12>
Instruction <128>
Data <64>
Data virtual page number <36>
Page offset <12>
Data in <64>
I TLB
D TLB
<4> Prot
<1> V
<31> Tag
<24> Physical address
(128 PTEs in 4 banks)
(64 PTEs in 4 banks)
4:1 mux
<24>
<28>
<128>
<64>
L2 TLB
<29> Tag
<7>
(512 PTEs in 4 banks)
=?
I CACHE
D CACHE
<7> Index
<6> Block offset
V D
<1> <1> <23>
Tag <21>
Data <128x4>
(512 blocks in 4 banks)
(512 blocks in 8 banks)
2:1 mux
<30>
L2 CACHE
<21> Tag
<9> Index
Data <512>
8:1 mux
(4K blocks in 8 banks)
L3 CACHE
<17> Tag
<13> Index
16:1 mux
(128K blocks in 16 banks)
MAIN MEMORY
Memory Interface
DIMM
<64>
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16