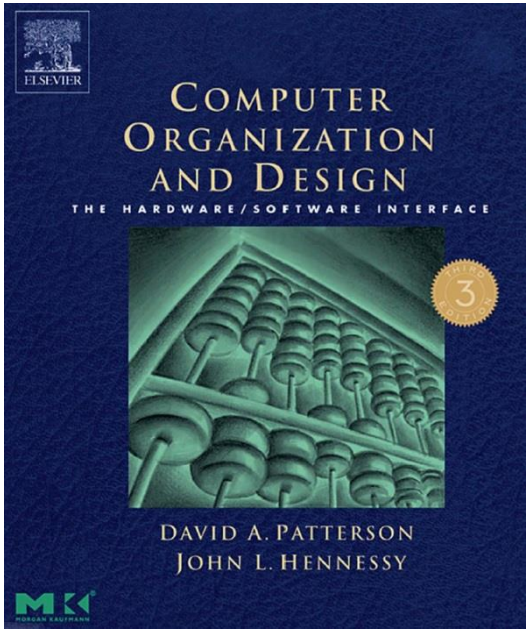
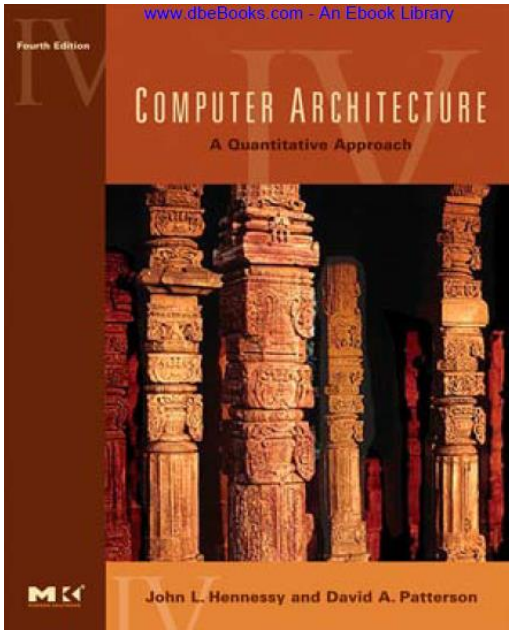
Virtual Memory

prepared and instructed by
Shmuel Wimer
Eng. Faculty, Bar-Ilan University





Motivation

Virtual memory (VM): A technique using main memory as a “cache” for secondary storage (disk).

Consider a collection of programs is simultaneously running on a computer.

The total required memory is much larger than the amount of main memory.

Main memory needs to contain only the active portions of the programs.

The principle of locality applies.

VM allows to efficiently share the processor and the main memory.

We must be able to protect the programs from each other.

A program can only read and write the portions of main memory that have been assigned to it.

Virtual memory:

- Allows efficient and safe sharing of memory among multiple programs.
- Removes the programming burdens of a small, limited amount of main memory (past, less relevant today).

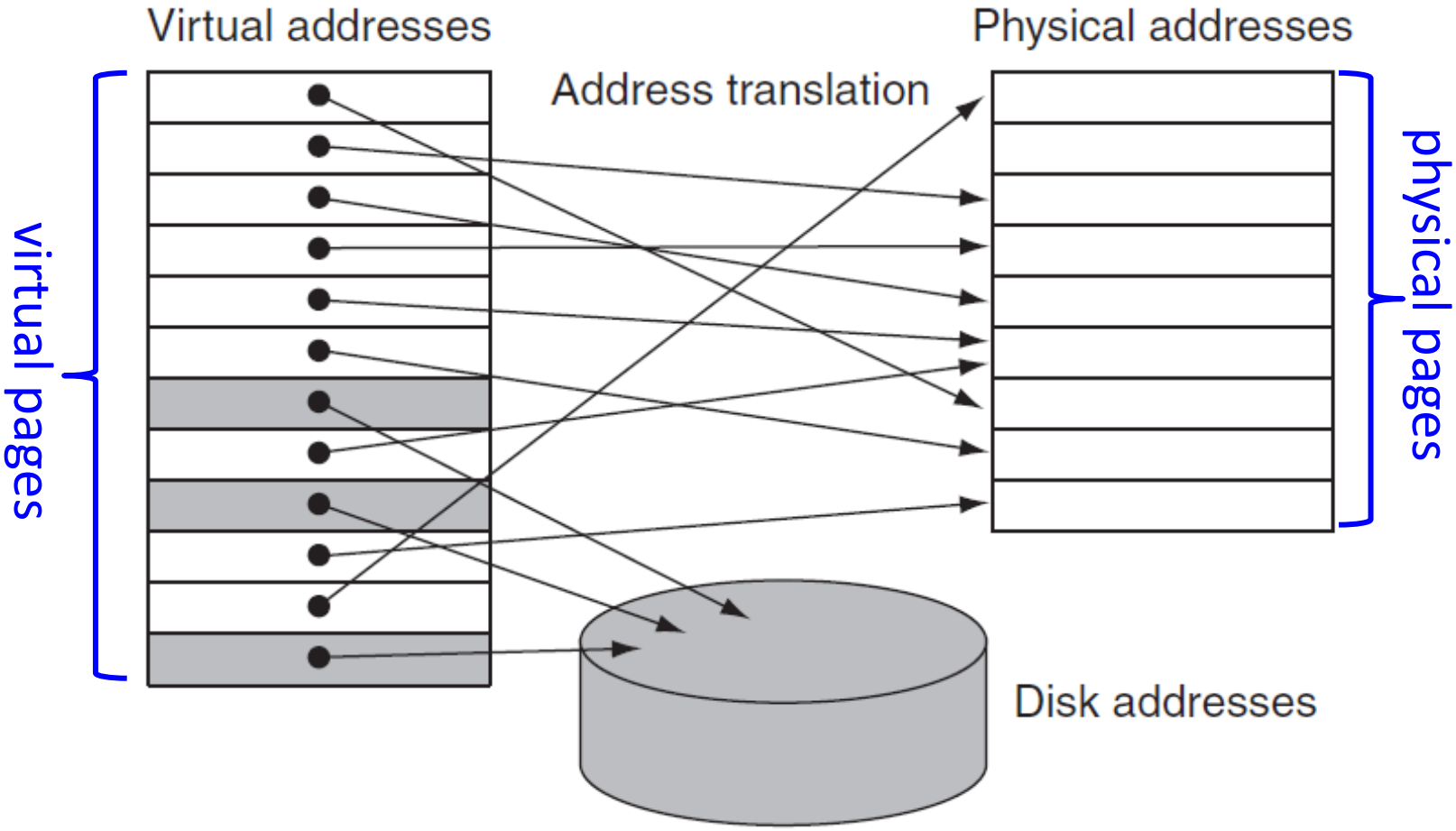
The programs sharing the memory change dynamically while the programs are running. The compiler sets each program into its own **address space**.

VM translates the program's address space to **physical addresses**, enforcing **protection** of a program's address space from other programs.

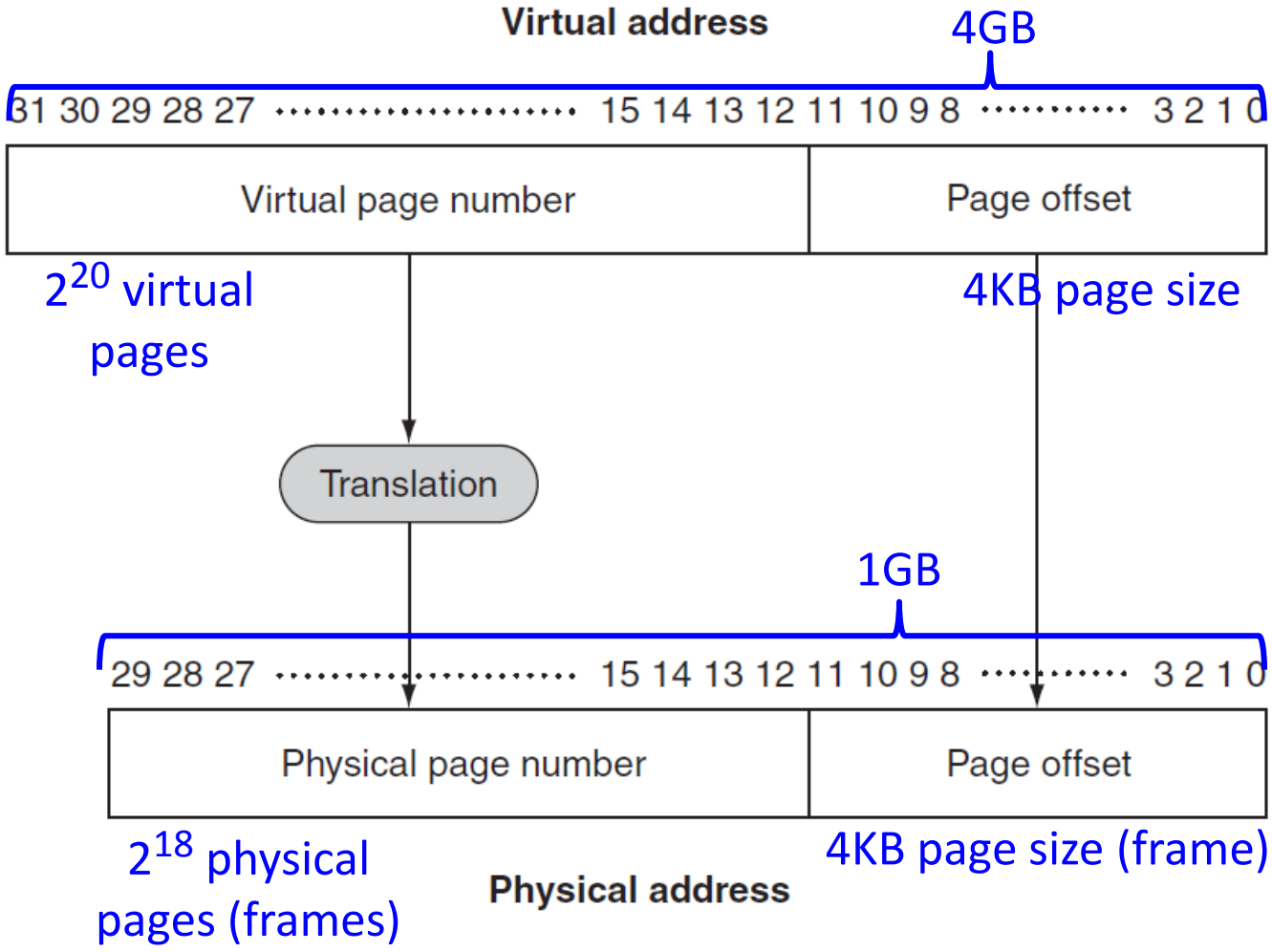
VM allows a single user program to exceed the size of primary memory.

A VM block is called a **page**, and a virtual memory miss is called a **page fault**.

VM produces a **virtual address**, translated by a SW and HW combination to a **physical address**.


Virtual addresses
Physical addresses
Address translation
virtual pages
physical pages
Disk addresses

Virtual to Physical Address Mapping



Illusion of an unbounded amount of virtual memory.

Design choices in VM systems are motivated by the high cost of page fault, taking **millions** of clock cycles to process.

- Pages should be large enough to amortize the high access time. Size ranges from 1 KB (embedded), 16 KB (PC) to 64 KB (severs).
- Organizations that reduce the page fault rate, e.g., fully associative placement of pages in memory.
- Page faults can be handled in SW because the overhead will be small compared to the disk access time. SW can use smart algorithms for page placement.
- Write-through will not work for VM, since writes take too long. VM uses **write-back**.

Placing a Page and Finding it Again

Reducing page fault frequency is critical due to the high page fault penalty.

The operating system (OS) maps the virtual to any physical page (associative placement of pages).

The OS can use smart algorithms and complex data structures to track page usage.

A **page table** stored **in memory**, indexed by the virtual page number, contains the virtual to physical address translations.

An entry contains the physical page number for that virtual page **if the page is currently in memory**.

To indicate the location of the page table in memory, the hardware includes a **page table register**, pointing to the start of the page table.

The page table + PC + registers specify the **state** of a program (process). To allow another program to use the processor, this state is saved. Restoration of this state enables the program to continue execution.

A process is **active** when it is in possession of the processor. Otherwise it is **inactive**.

OS activates a process by loading the process's state, including the PC.

The process's address space, and hence all the data it can access in memory, is defined by its page table, residing in the memory.

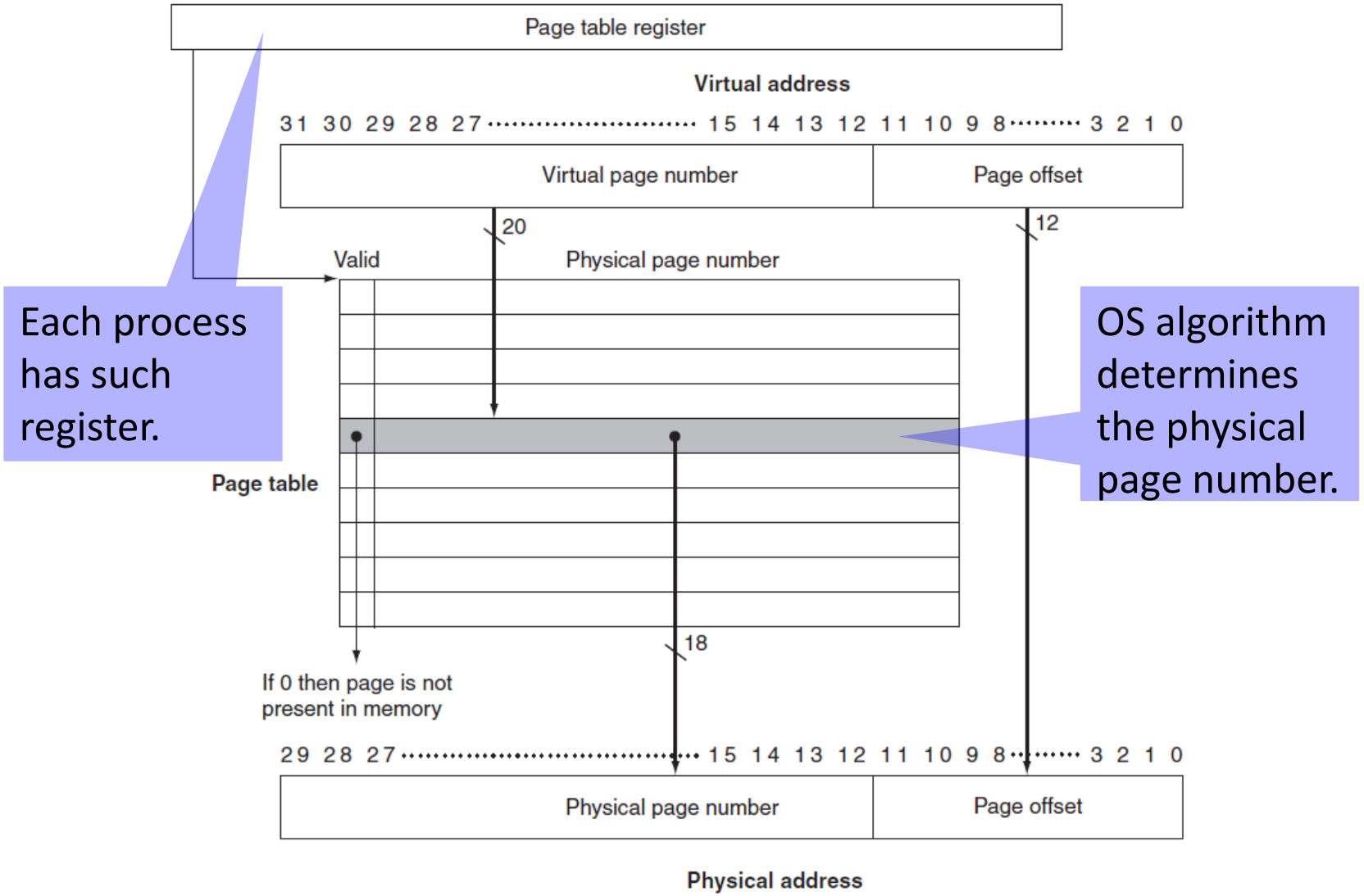
The OS needs to load only the page table register with the pointer to the table of the process to be run.

Each process has its own page table (+register), since different processes use the same virtual addresses.

The OS allocates the physical memory and updates the page tables, to avoid **collision** of the virtual address spaces of different processes.

Separate page tables provide **protection** of one process from another.

The page table



Each process has such register.

OS algorithm determines the physical page number.

The physical page bit-width is extended to 32 for ease of indexing. The extra bits are used to store additional information.

The OS creates the space on **disk** for all the pages of a process when it creates the process.

This disk space is called the **swap space**.

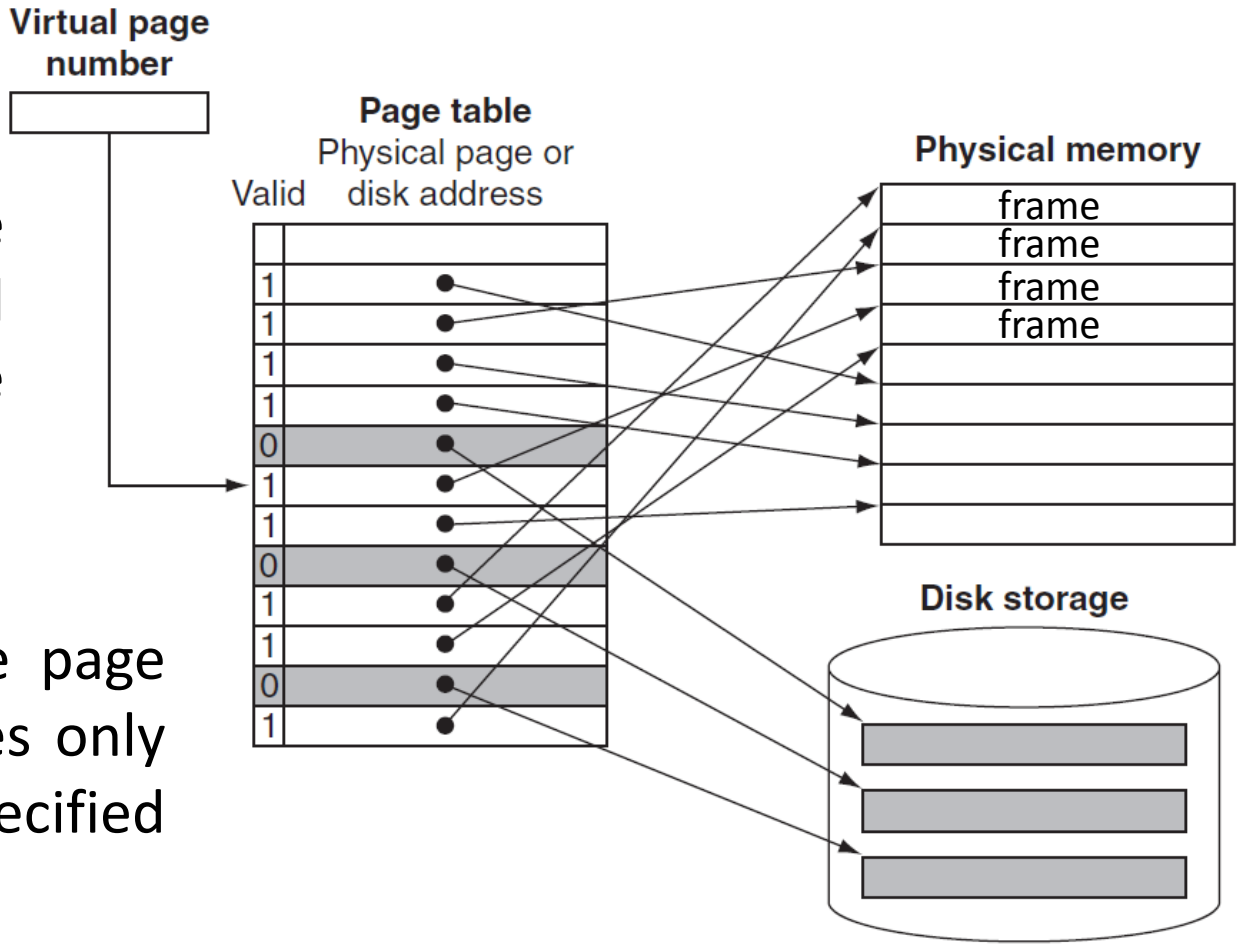
OS also creates a data structure to record where each virtual page is stored on disk.

It may be part of the page table or an auxiliary data structure indexed in the same way as the page table.

The page table maps each VM page to either a page in main memory or a page stored on disk.

Virtual page number is used to index the page table.

Valid bit 0: the page currently resides only on disk, at a specified disk address.



OS handles data structure to track which processes and which virtual addresses use each physical page.

The OS is another process, and these tables controlling memory reside in memory.

When all the pages in main memory are in use, the OS chooses a page to replace. Replaced pages are written to swap space on the disk.

To minimize the number of page faults **LRU** is used. But it is too expensive, requiring updating a data structure on **every** memory reference. Instead, an approximation is used.

Some computers provide a **use bit** or **reference bit**, (in HW) which is set whenever a page is accessed.

OS periodically clears the use bits and records them. It can determine which pages were touched during a certain time period.

OS then evicts a page whose reference bit is off.

With 32-bit virtual address, 4 KB page size, and 4 bytes per page table entry, the total page table size is: # page table entries = $2^{32} / 2^{12} = 2^{20}$.

Size of page table = 2^{20} x 4bytes = 4MB **(for every process! There may be 100's!)**

Writes in Virtual Memory

Access time difference between cache and main memory is 10's – 100's cycles. Write-through with a write buffer to hide the latency of the write from the processor worked.

In a VM system, writes to disk take millions processor cycles, so write-through is impractical.

Write-back, called **copy back**, is copying the page back to disk when it is replaced in the memory.

Disk **transfer time** is small compared to **access time**, so copy back is far more efficient than write-through.

A write-back is still costly.

We would like to know whether at a replacement the page needs to be copied back.

A **dirty bit** is added to the page table, being set when any word in a page is written.

The dirty bit indicates whether the page needs to be written out before its location in memory can be given to another page.

Hierarchical Paging

Modern computer support logical address space of 2^{32} to 2^{64}, making page table excessively large.

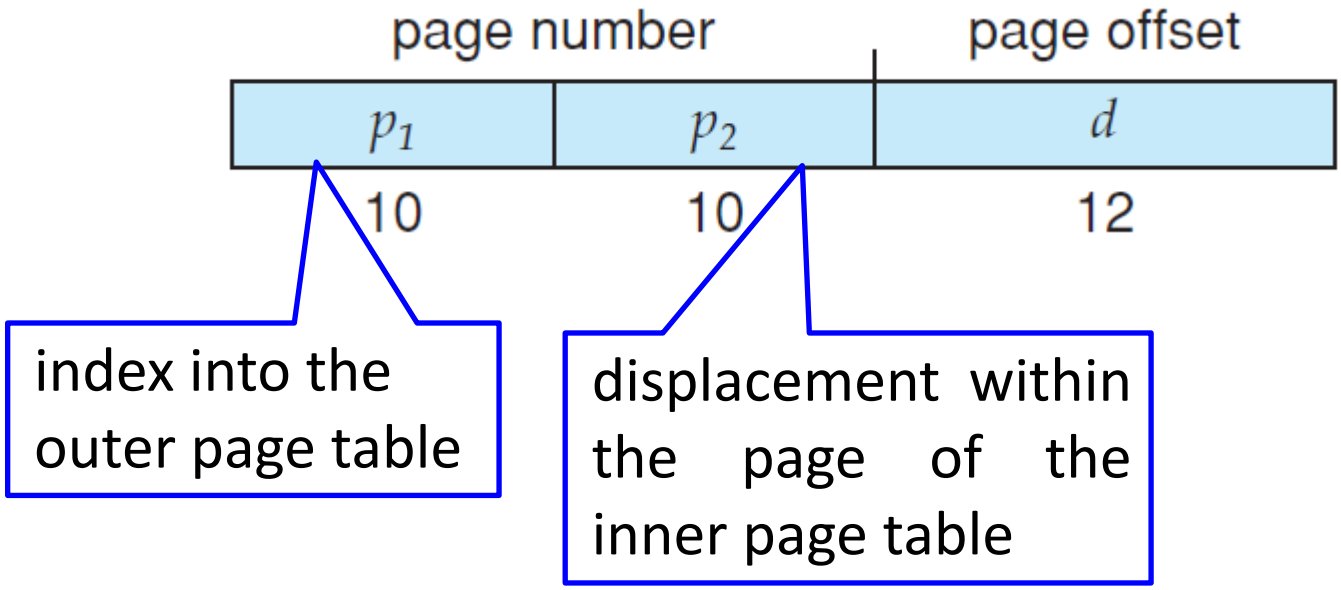
Example: In 32-bit address with 4KB (2^{12}) page size, page table may have $2^{32}/2^{12} = 1\text{M}$ entries, 4byte each, yielding 4MB physical space for a page table. ■

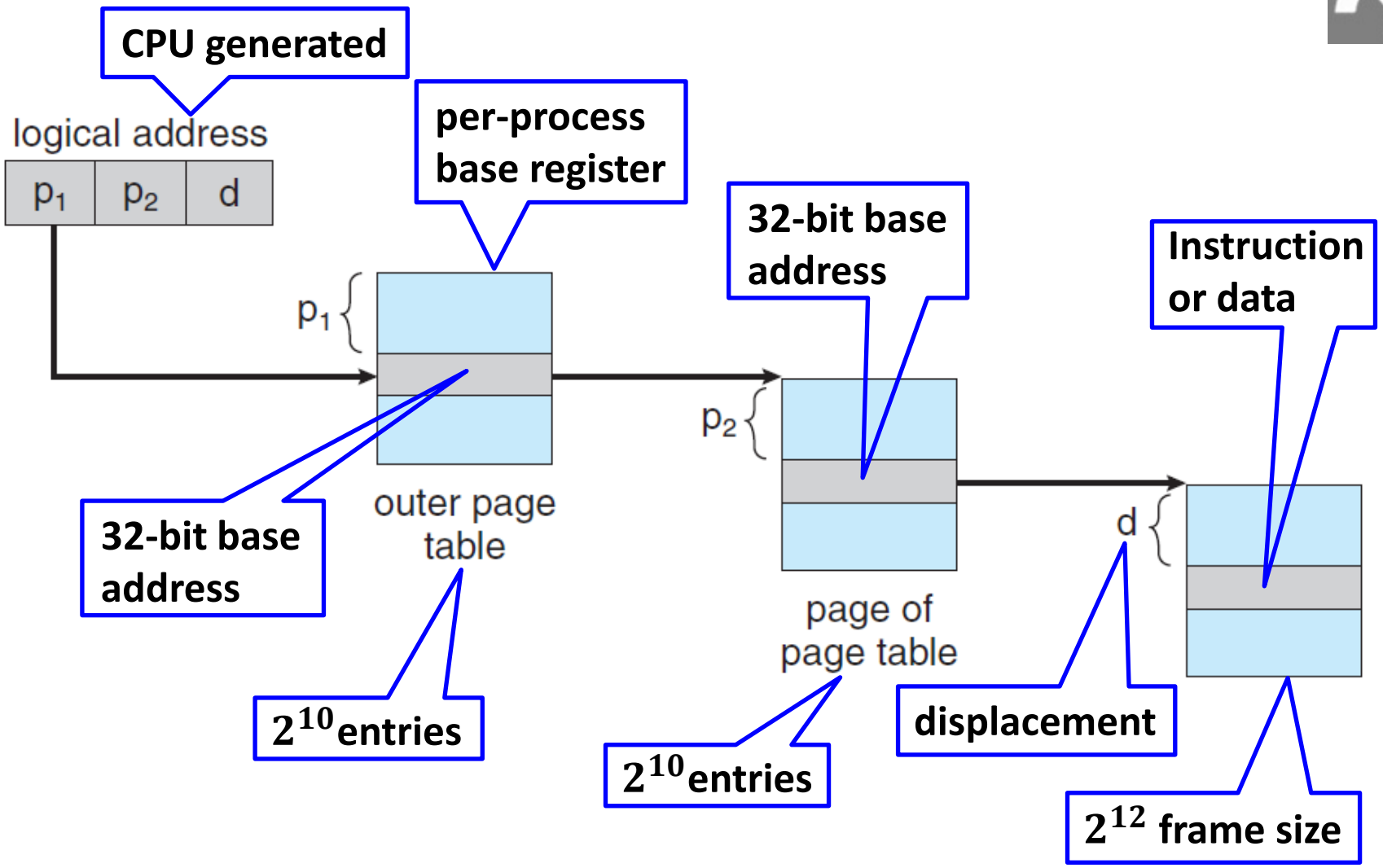
We do not want to allocate the page table **contiguously** in main memory.

A solution is to use a two-level paging algorithm, in which the page table itself is also paged.

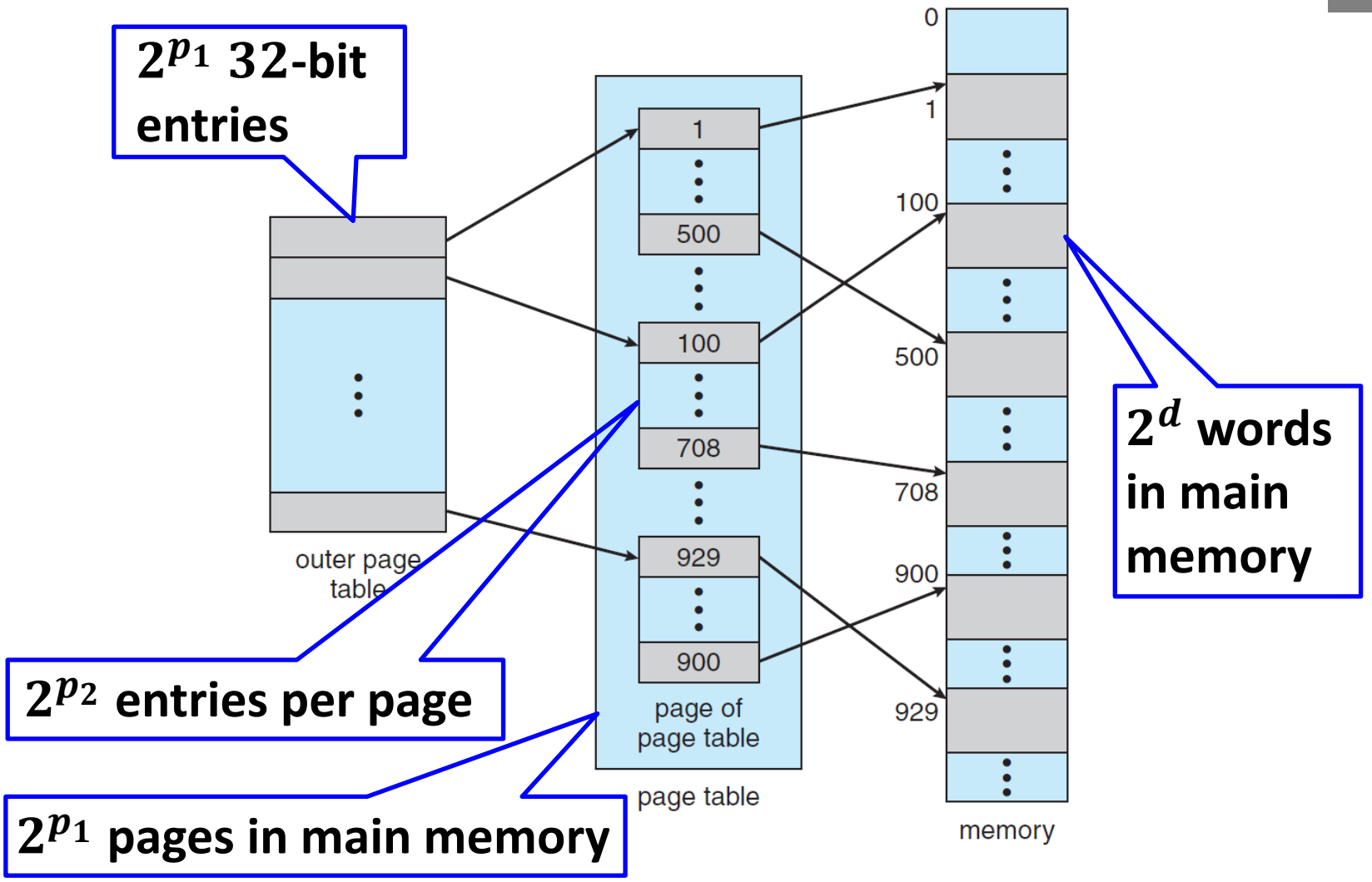
Example: 32-bit logical address and 4KB page size. Logical address is divided into 20-bit page # and 12-bit page offset.

The page number is further divided into 10-bit **page number** and 10-bit **page offset**. ■





Address translation for a two-level 32-bit paging.



Two-level page-table (forward-mapped page table).

Fast Address Translation: The **TLB**

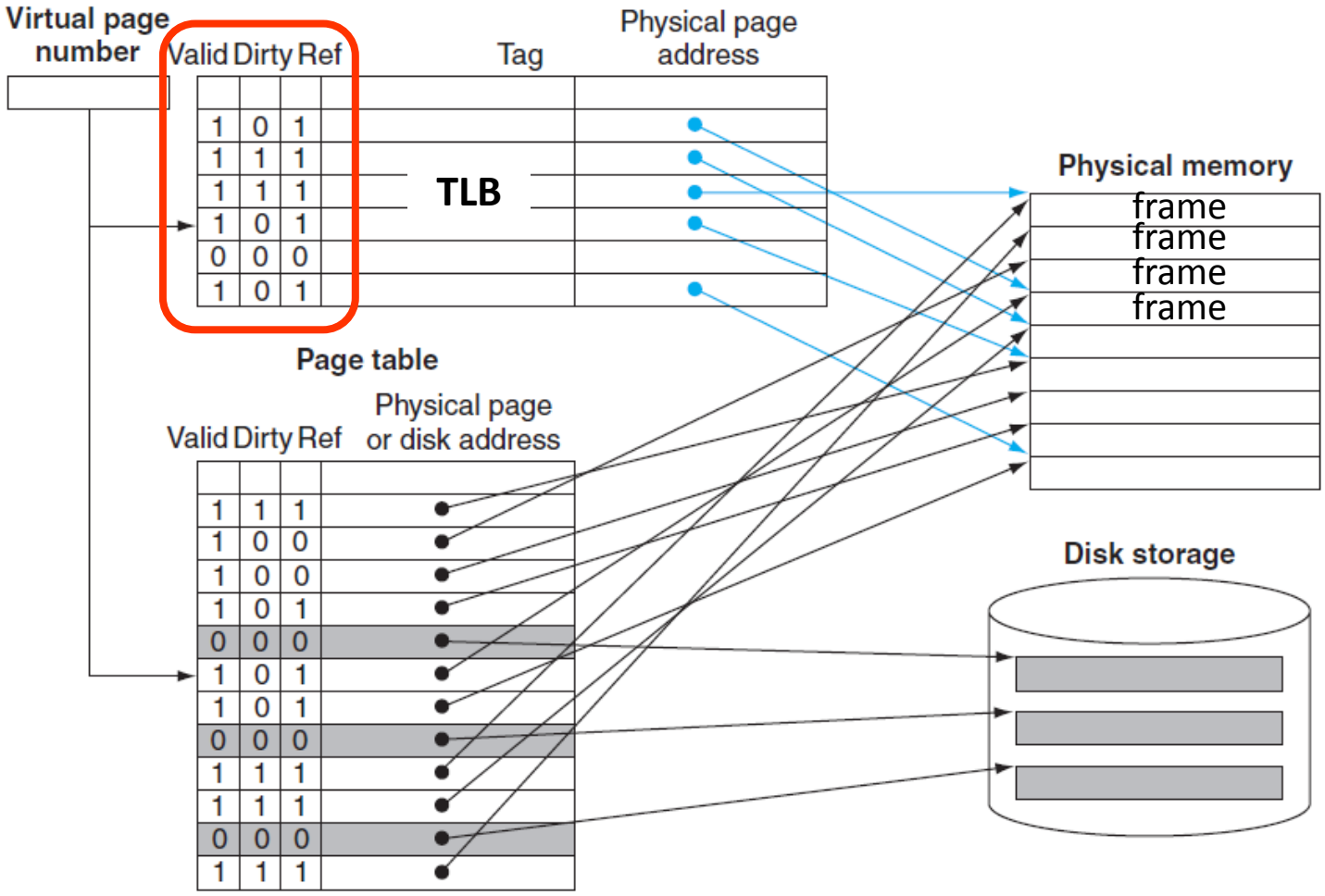
The page tables are stored in main memory. Every memory access by a program is thus twice long:

- One access to obtain the physical address (from page table),
- a second access to get the data (elsewhere in memory).

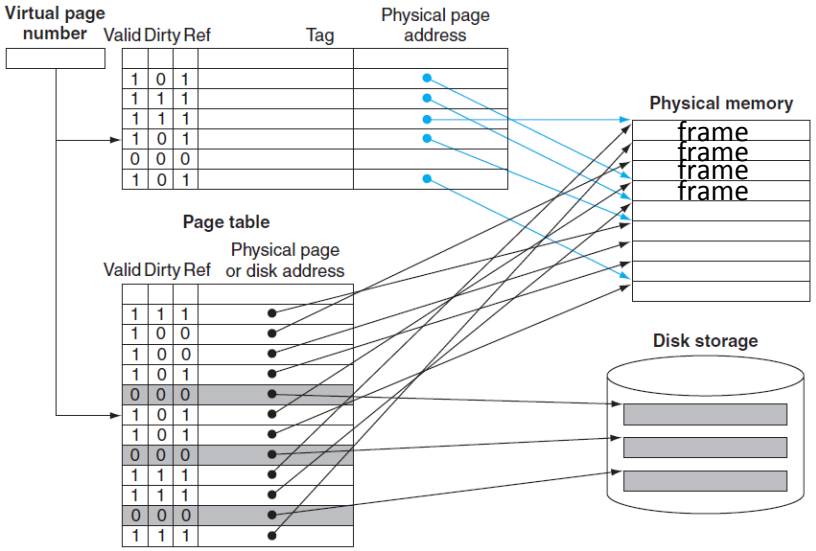
Locality of reference to the page table can help:

- A translation for a virtual page number will probably be needed again soon, because references to the words on that page have both **temporal** and **spatial** locality.

A special **cache**, called **Translation Look-aside Buffer** (**TLB**), keeps track of recently used translations.



We access the TLB on every reference instead of the page table. TLB must therefore include the **valid**, **dirty** and the **reference** bits.



Every reference looks up the virtual page # in TLB.

TLB hit uses the physical page # to form the address and turns on the reference bit. Write turns on the dirty bit too.

A TLB miss can be either a **true** page fault or just a TLB miss. If the page exists in memory, the processor loads the translation from the page table into the TLB and tries the reference again.

In a **true** page fault the processor invokes the OS (exception).

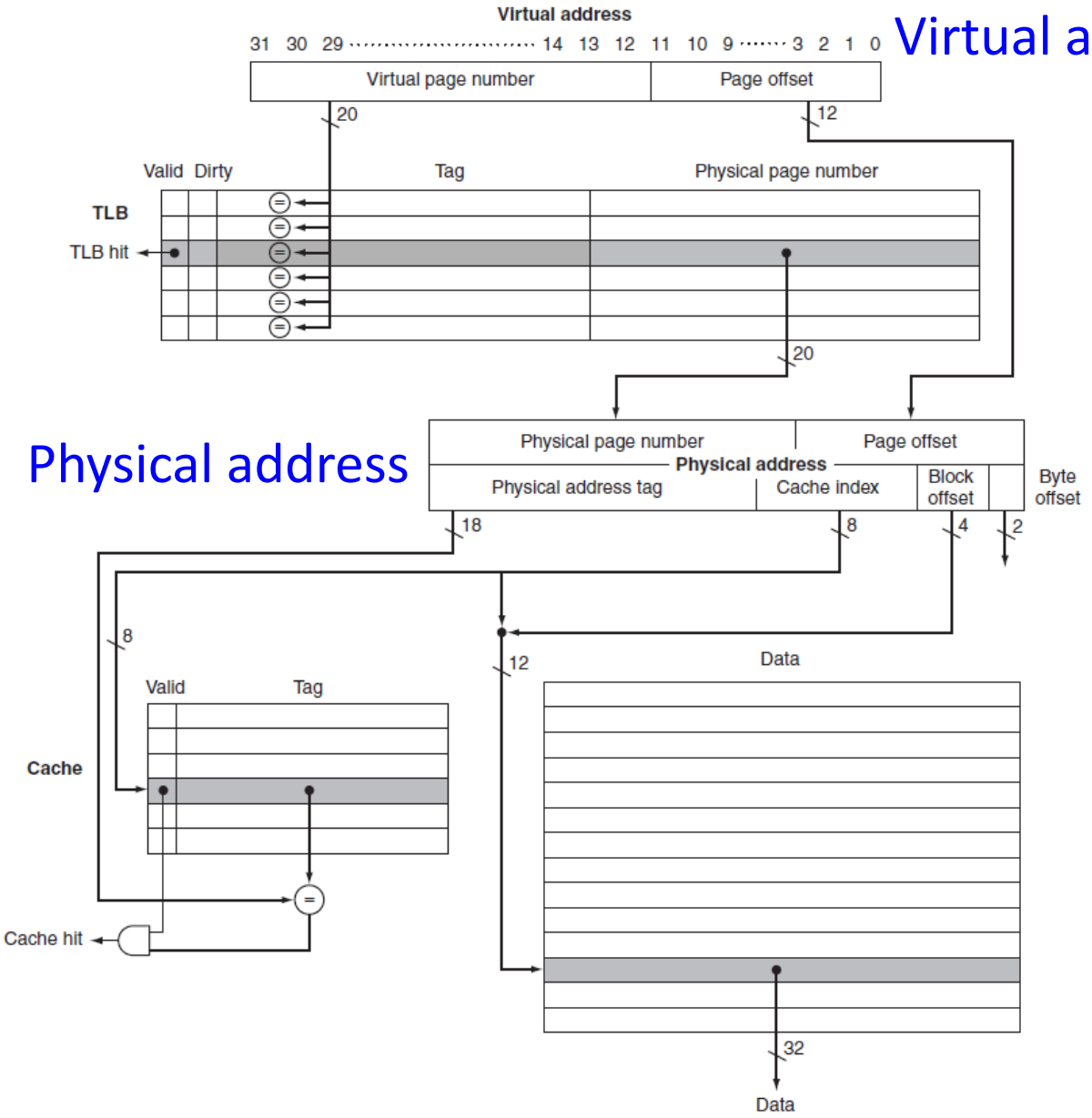
TLB has much fewer entries than pages in main memory. TLB misses are therefore much more frequent than page faults.

At TLB miss we need to select a TLB entry to replace and copy the **reference** and **dirty** bits back to the page table entry. Those are the only TLB entry portion that can be changed.

Write-back copies entries back from TLB at miss time rather than when they are written (**dirty**, **reference**). It is very efficient since TLB miss rate is small.

Typical values for a TLB

- TLB size: 16–512 entries
- Block size: 1–2 page table entries (typically 4–8 bytes each)
- Hit time: 0.5–1 clock cycle
- Miss penalty: 10s–100s clock cycles
- Miss rate: 0.01%–1%



Virtual address

TLB

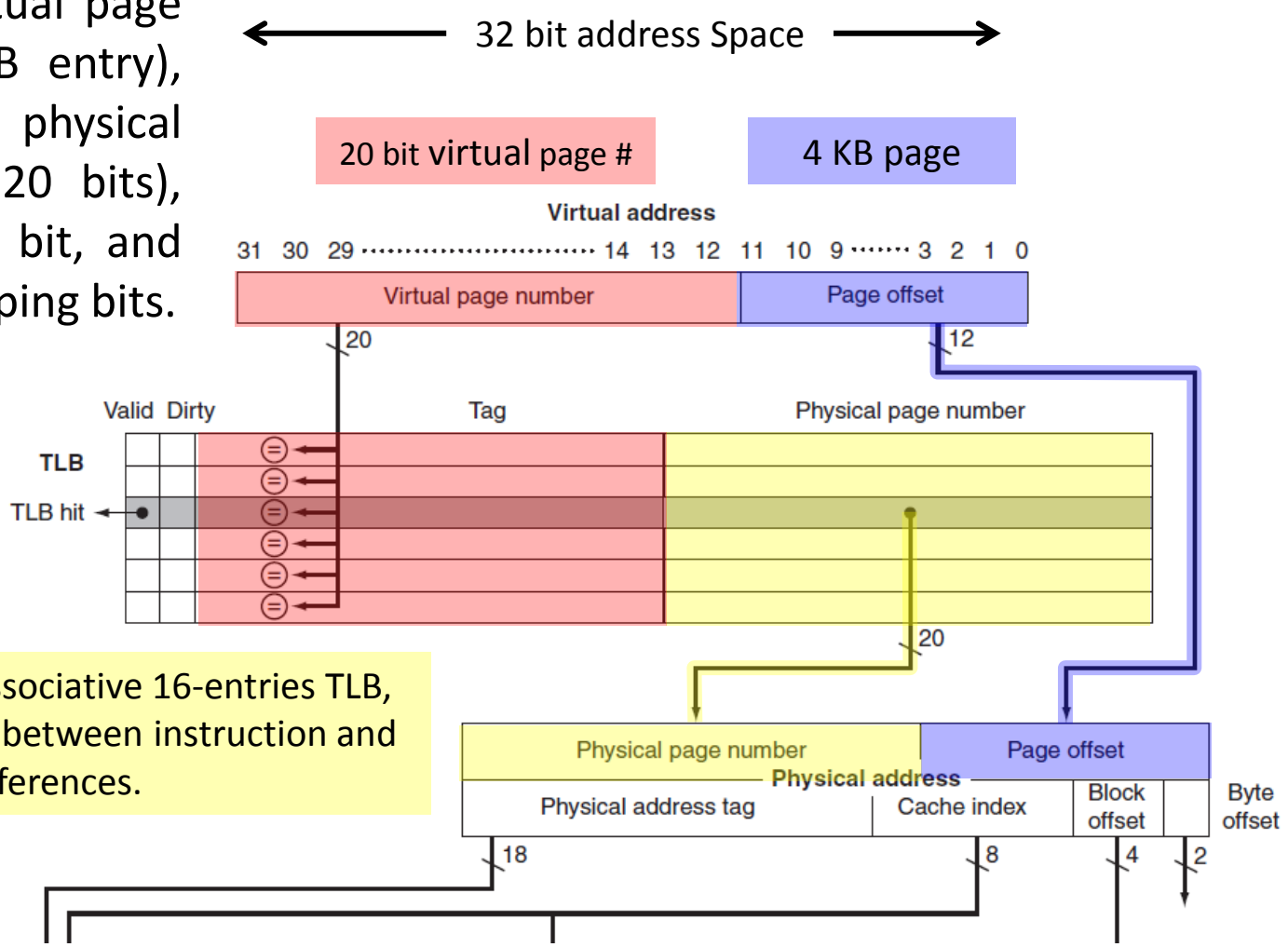
Physical address

Real TLB Read

cache

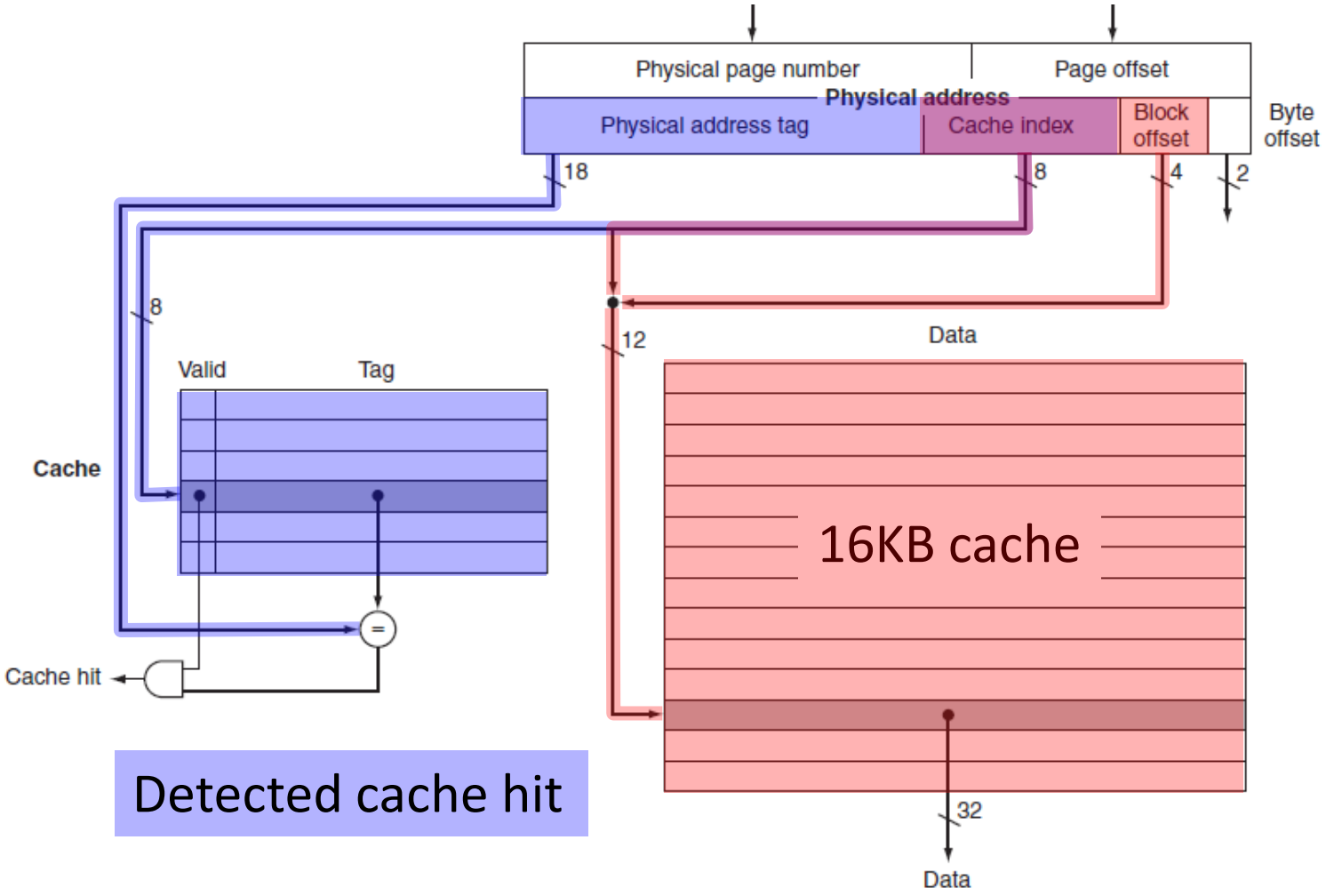
TLB entry is 64 bits:

20-bit tag (virtual page # for that TLB entry), corresponding physical page # (also 20 bits), valid bit, dirty bit, and other bookkeeping bits.



Fully associative 16-entries TLB, shared between instruction and data references.

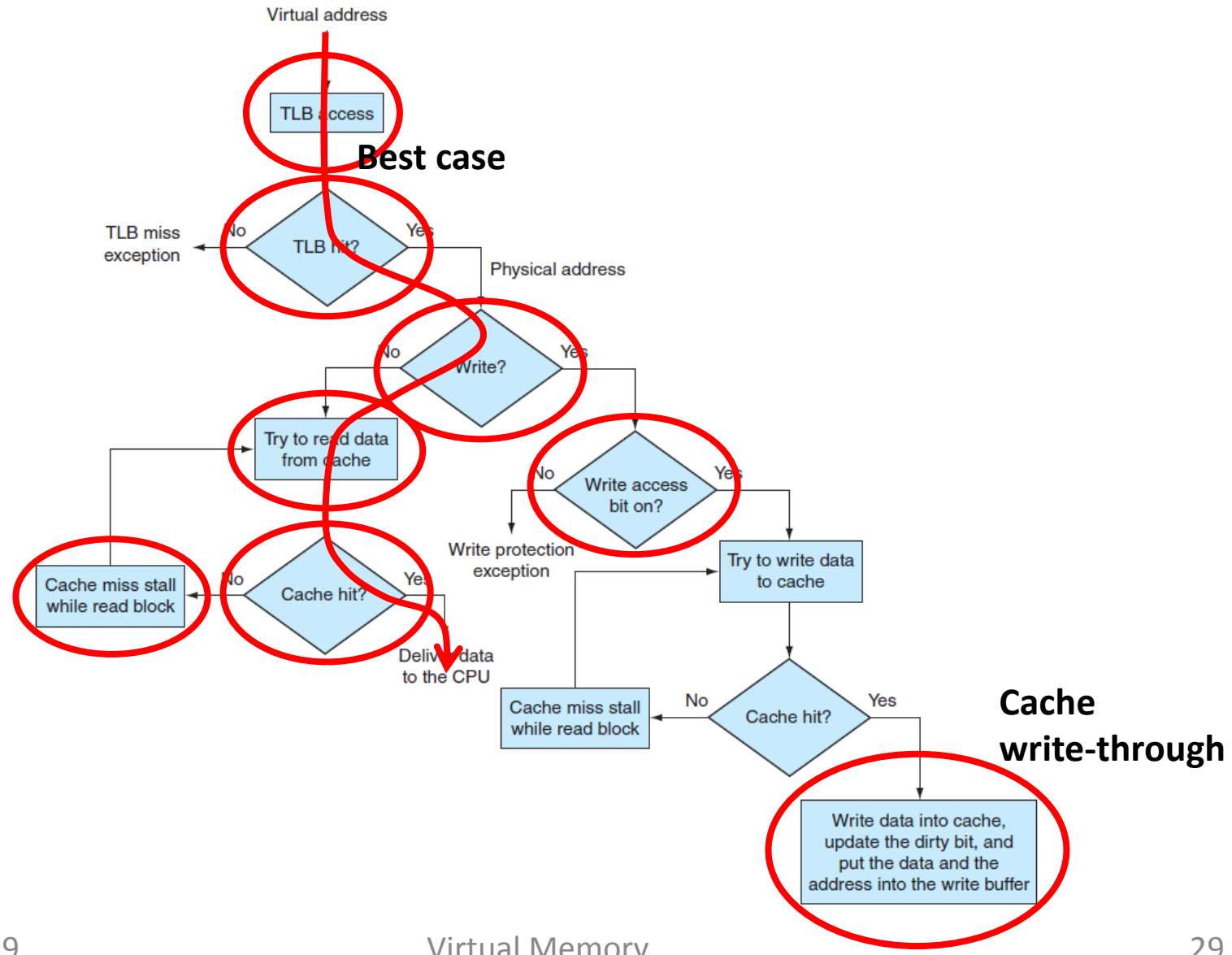
physical address generation



Detected cache hit

read from cache

Read and Cache Write-Through



What happens at MIPS TLB miss? First, the **virtual page #** of the reference is saved in a special register (for the OS).

The OS handles the miss in SW (exception). The routine indexes the page table using the **virtual** page # and the **page table register** (stores the starting address of the active process page table).

Using a special set of system instructions that can update the TLB, the OS places the physical address from the page table into the TLB.

TLB miss takes 13 clock cycles, assuming the OS code and the page table entry are in the instruction and data caches.

A **true** page fault occurs if the page table entry does not have a valid physical address.

VM, TLBs and Caches Integration

VM and cache work together as a hierarchy. Data must be in main memory if it is in the cache.

OS maintains this hierarchy by flushing the contents of any page from the cache upon deciding to migrate that page to disk.

The OS modifies the page tables and TLB accordingly. An attempt to access any data on the page generates a page fault.

In **best case**, a virtual address is translated by the TLB and sent to the cache where the appropriate data is found, retrieved, and sent back to the processor.

In the **worst case**, a reference can miss all: the TLB, the page table, and the cache.

Consider all the seven combinations of the three events. State for each whether it can actually occur and under what circumstances.

TLB	Page table	Cache	Possible? If so, under what circumstance?
hit	hit	miss	Possible, although the page table is never really checked if TLB hits.
miss	hit	hit	TLB misses, but entry found in page table; after retry, data is found in cache.
miss	hit	miss	TLB misses, but entry found in page table; after retry, data misses in cache.
miss	miss	miss	TLB misses and is followed by a page fault; after retry, data must miss in cache.
hit	miss	miss	Impossible: cannot have a translation in TLB if page is not present in memory.
hit	miss	hit	Impossible: cannot have a translation in TLB if page is not present in memory.
miss	miss	hit	Impossible: data cannot be allowed in cache if the page is not in memory.

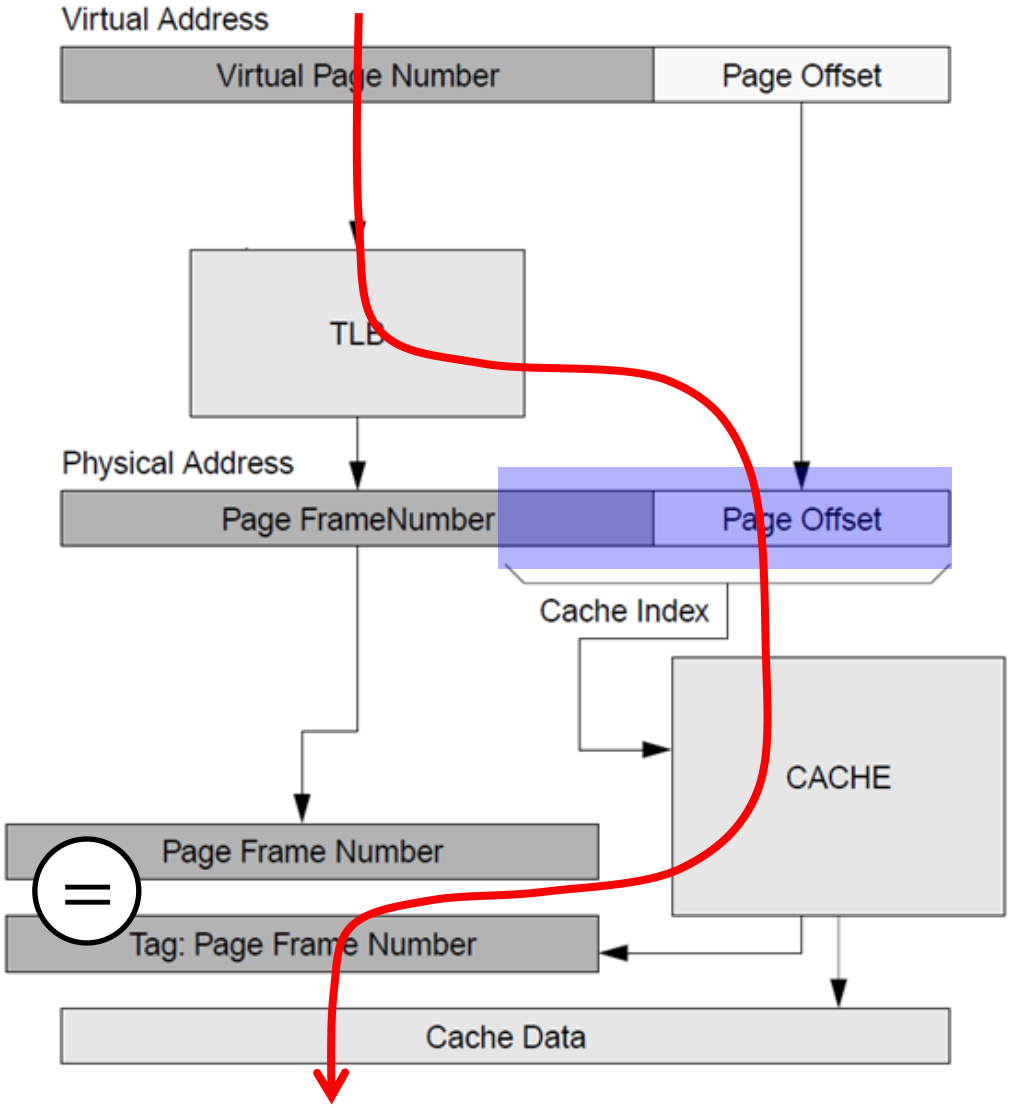
Virtual Caches

All memory addresses were translated to physical addresses before the cache was accessed.

The cache is thus **physically indexed** and **physically tagged**, called also **transparent** cache.

Pro: Since the cache and the physical memory use the same namespace, it can be entirely controlled by HW, and the OS needs no intervene.

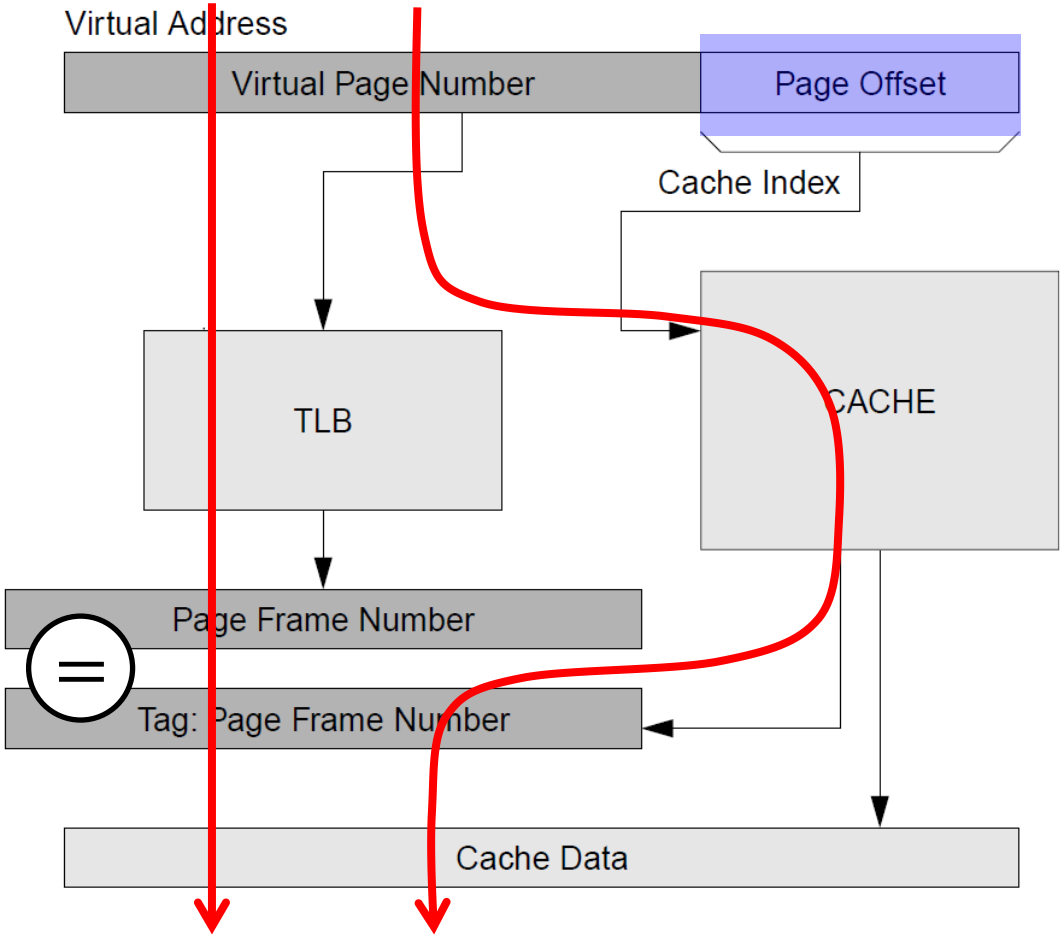
Con: Address translation is on critical path. Problem for high clock speed, large application data (memory size), larger TLB needed.



At hit, time to access memory includes TLB and cache accesses. The two can be **pipelined**.

Can the TLB be eliminated from the critical path?

Note that the cache index involves both page offset and frame number bits, but page bits are not translated.

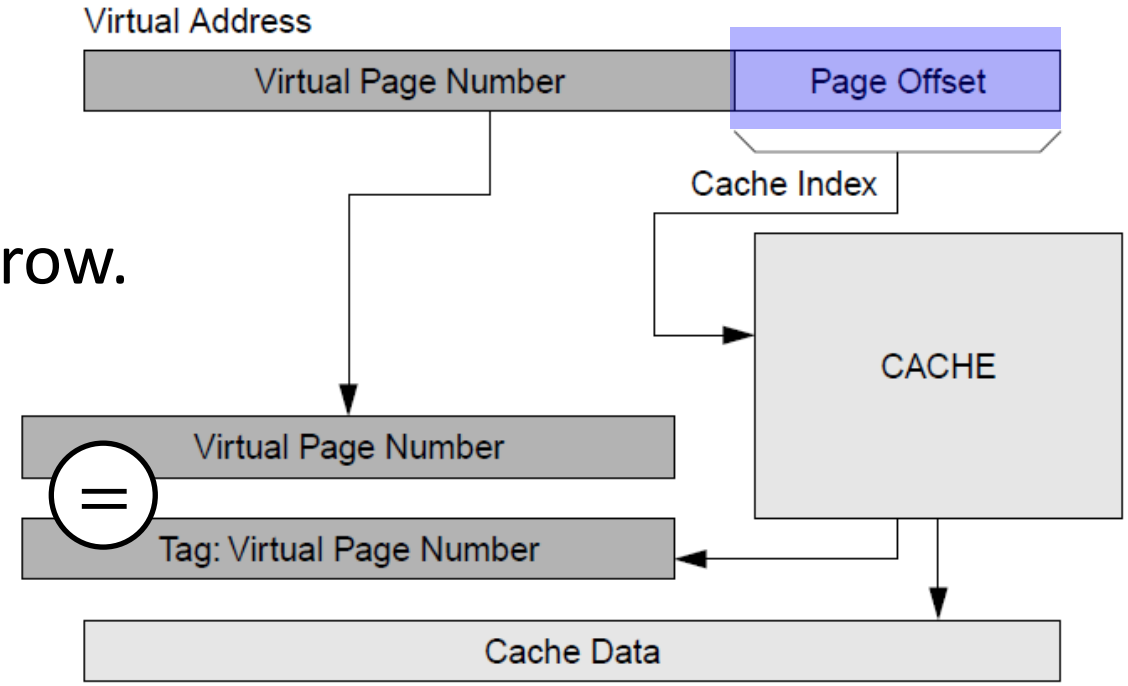


If cache index uses only page offset bits, TLB is eliminated from the critical path.

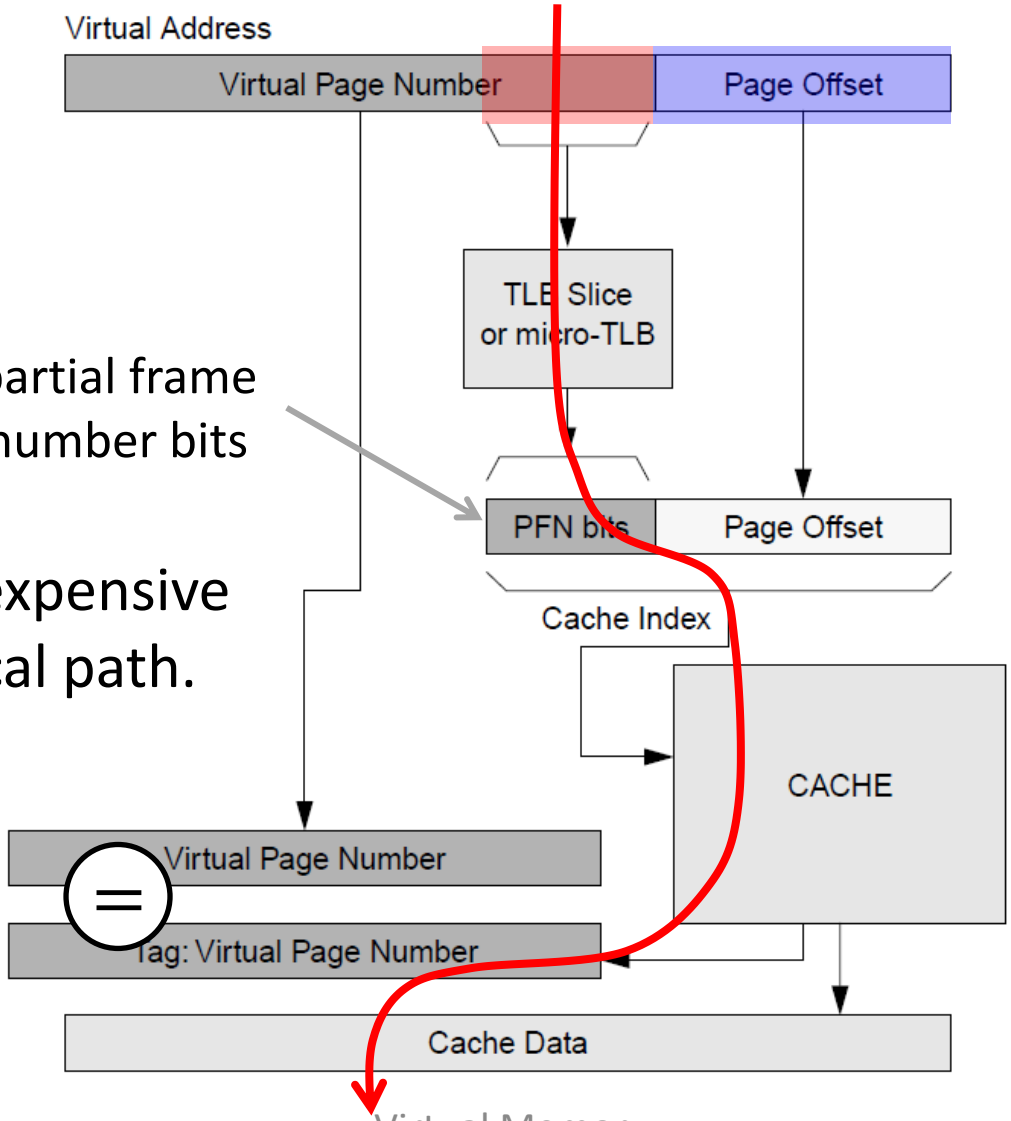
Con: Cache index cannot grow, same cache and frame size. Cache capacity can grow only by increasing block size and/or associativity.

Physically indexed virtually tagged eliminate TLB.

Index still cannot grow.

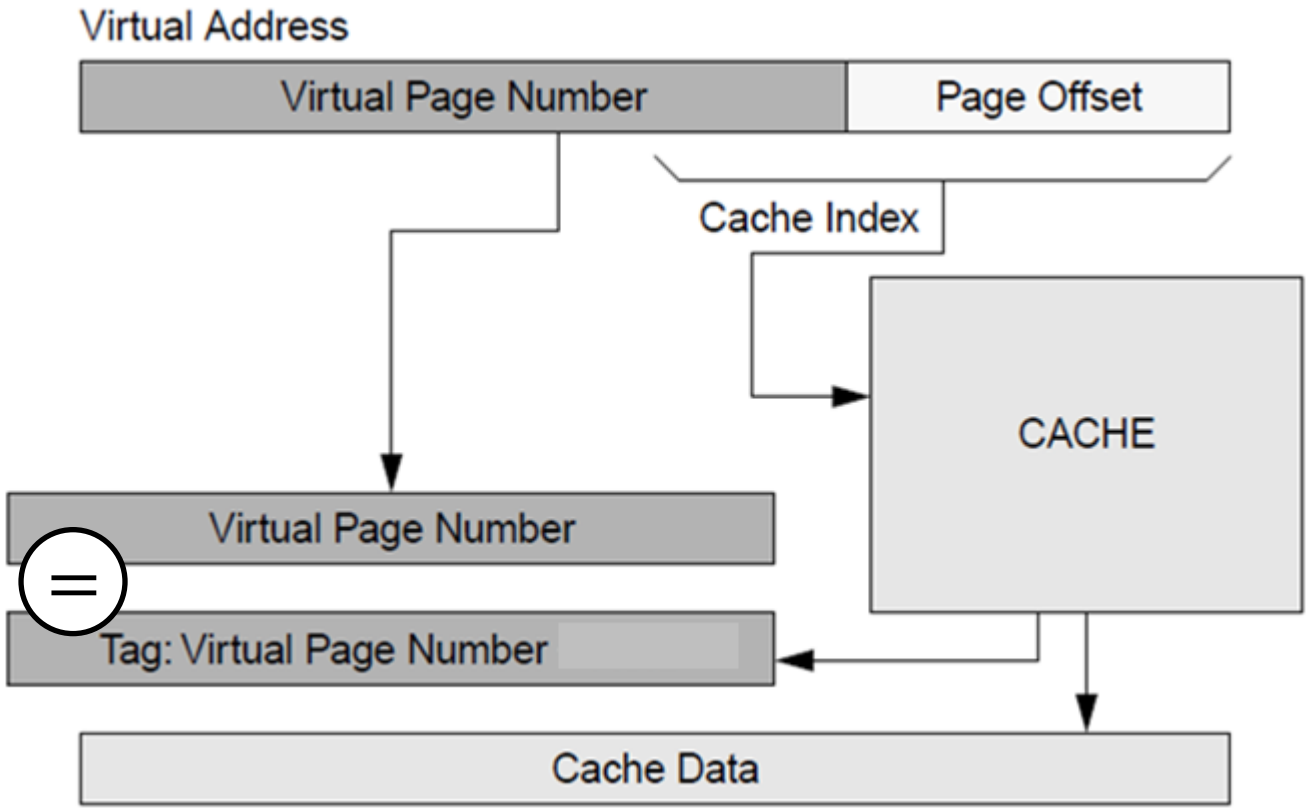


Solved by translating a TLB slice (MIPS R6000).



TLB less expensive but on critical path.

In **virtually indexed virtually tagged** translation (TLB) is not needed anywhere.



A TLB is needed on a **cache miss**. OS must translate the virtual address and load the datum from the main (physical) memory.

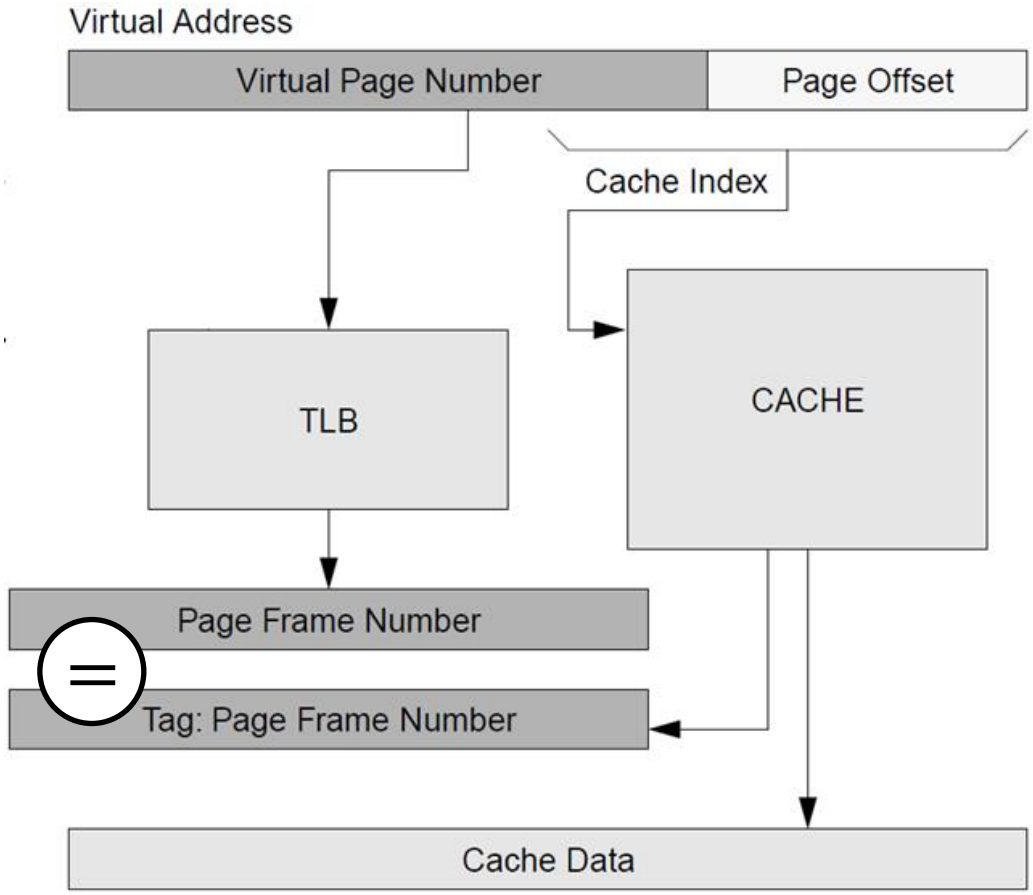
On a cache miss the TLB is not on the critical path and hence it could be large.

Virtual tagging may cause a page to be shared between programs. **Aliasing** may occur by two virtual addresses for the same frame.

A word may be cached by two different programs, allowing one to write data without the other being aware of it.

A compromise is by **virtually indexed physically tagged** cache.

Tag extraction by cache indexing and tag translation of virtual address are done in parallel.



It attempts to achieve the performance of virtually indexed caches with the architecturally simpler physically tagged cache.

Tag extraction by cache indexing and tag translation of the virtual address are done in parallel.

Unlike physically indexed caches, management is still necessary, because the cache is virtually indexed.