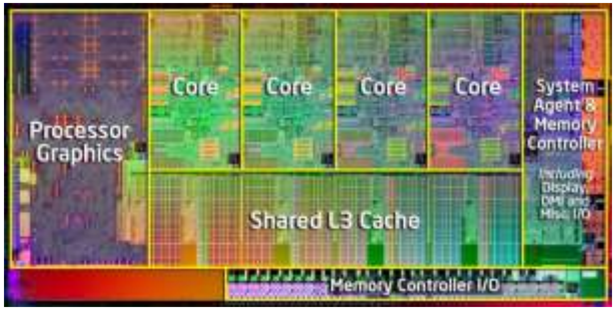


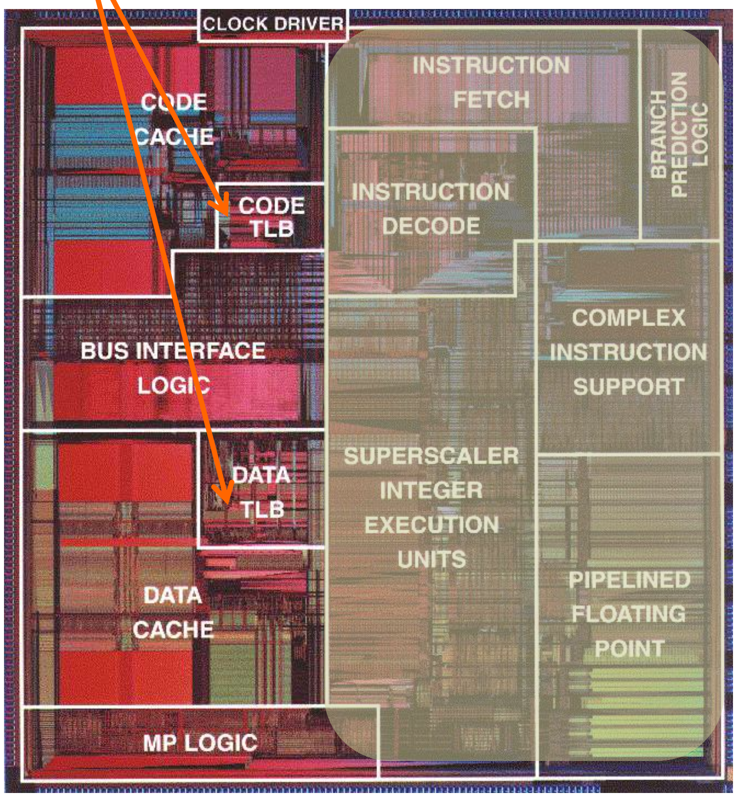
COSE222 Computer Architecture

Lecture 7. TLB for Virtual Memory

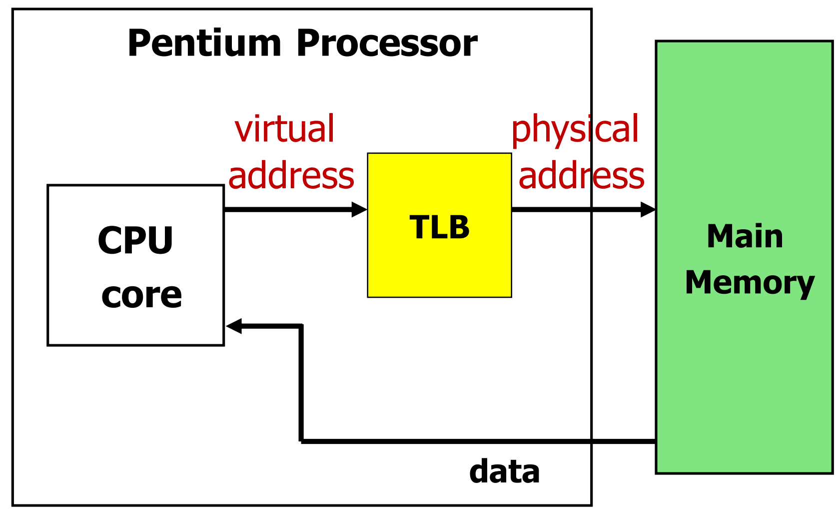
Prof. Taeweon Suh
Computer Science & Engineering
Korea University

Why TLB in Processor?

- Translation Lookaside Buffer (TLB)
  - TLB is there for Virtual Memory



Intel Pentium Processor (1993)

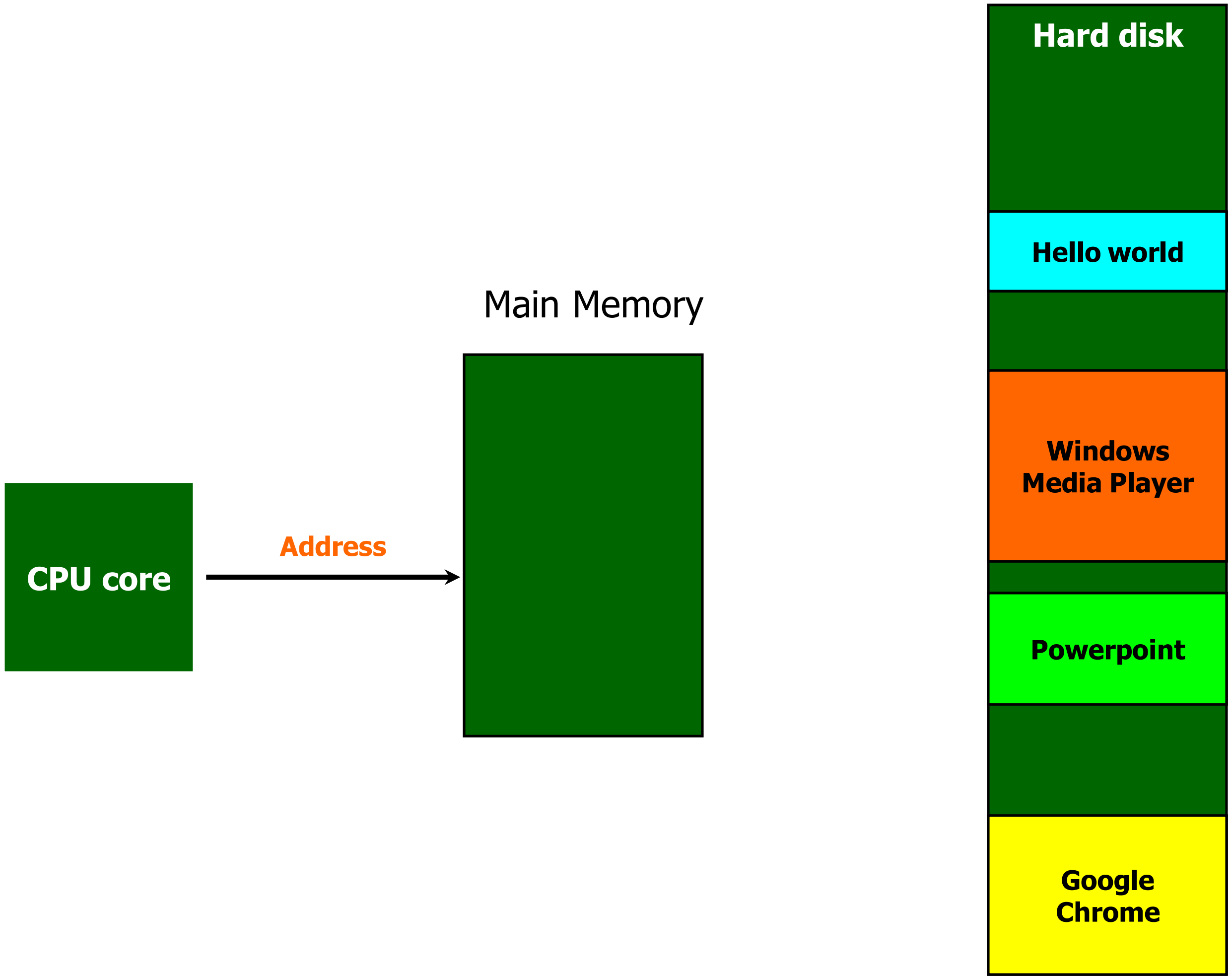


Pentium (1993)
60MHz
3.1M Transistors
(800nm process)

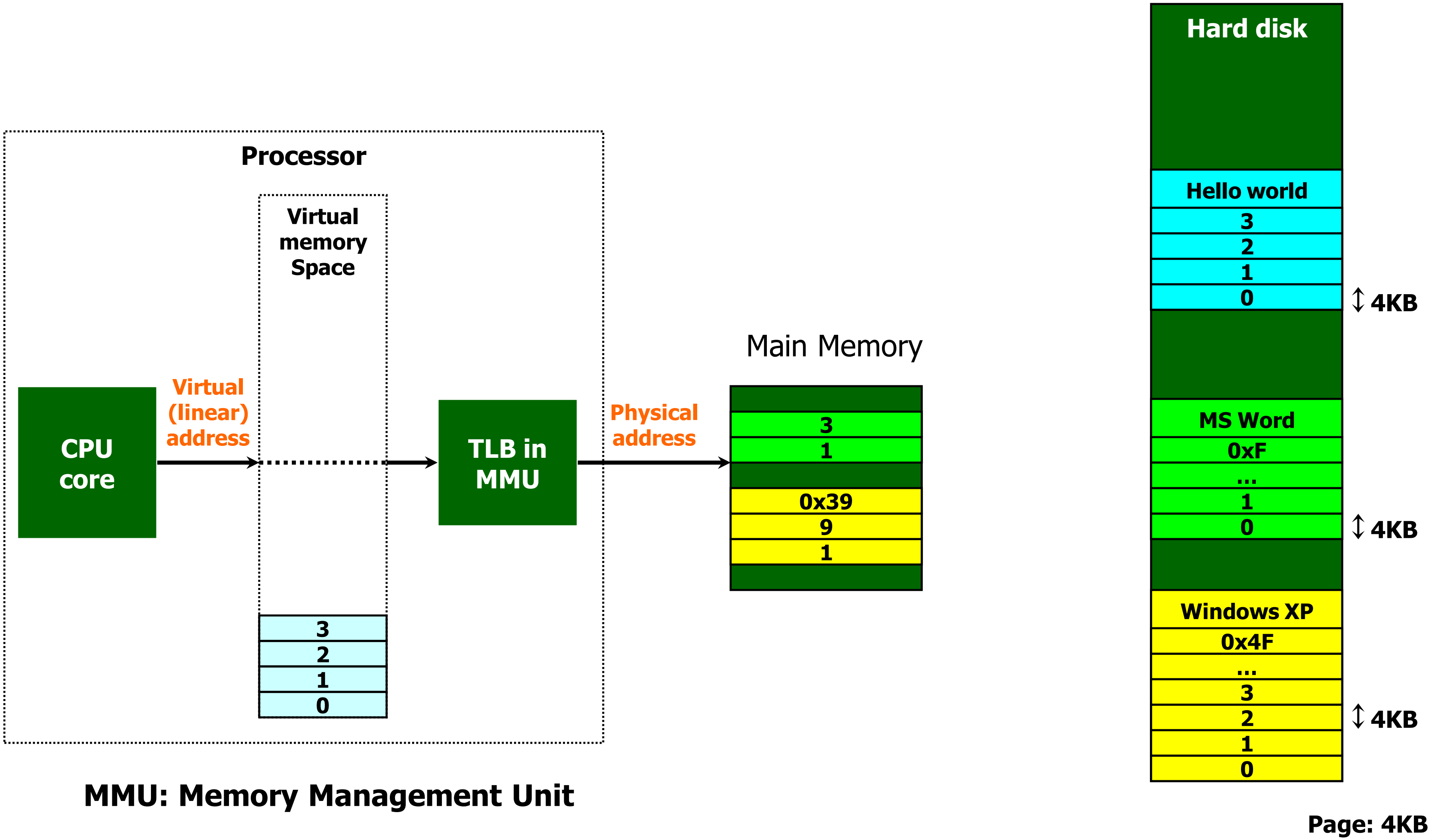
Motivation of Virtual Memory

- Virtual memory (VM) was invented to relieve programmers from burdens
  1. VM allows efficient and safe sharing of main memory among multiple programs
     - Consider a collection of programs running all at once on a computer
     - We don't know which programs will share main memory with other programs when we compile them
       - In fact, the programs sharing main memory change dynamically while the programs are running
       - Because of this dynamic interaction, we would like to compile each program into its own address space (virtual address space)
     - VM (implemented in Operating System) dynamically manages the translation of the program's address space (virtual address space) to the physical address space
  2. VM provides the ability to easily run programs larger than the size of physical memory
     - In old days, if a program is too large for memory, it was the programmers' responsibility to make it fit
     - Programmers divided programs into pieces and then load and unload pieces into main memory under user's program control

Complication in Multiprocessing

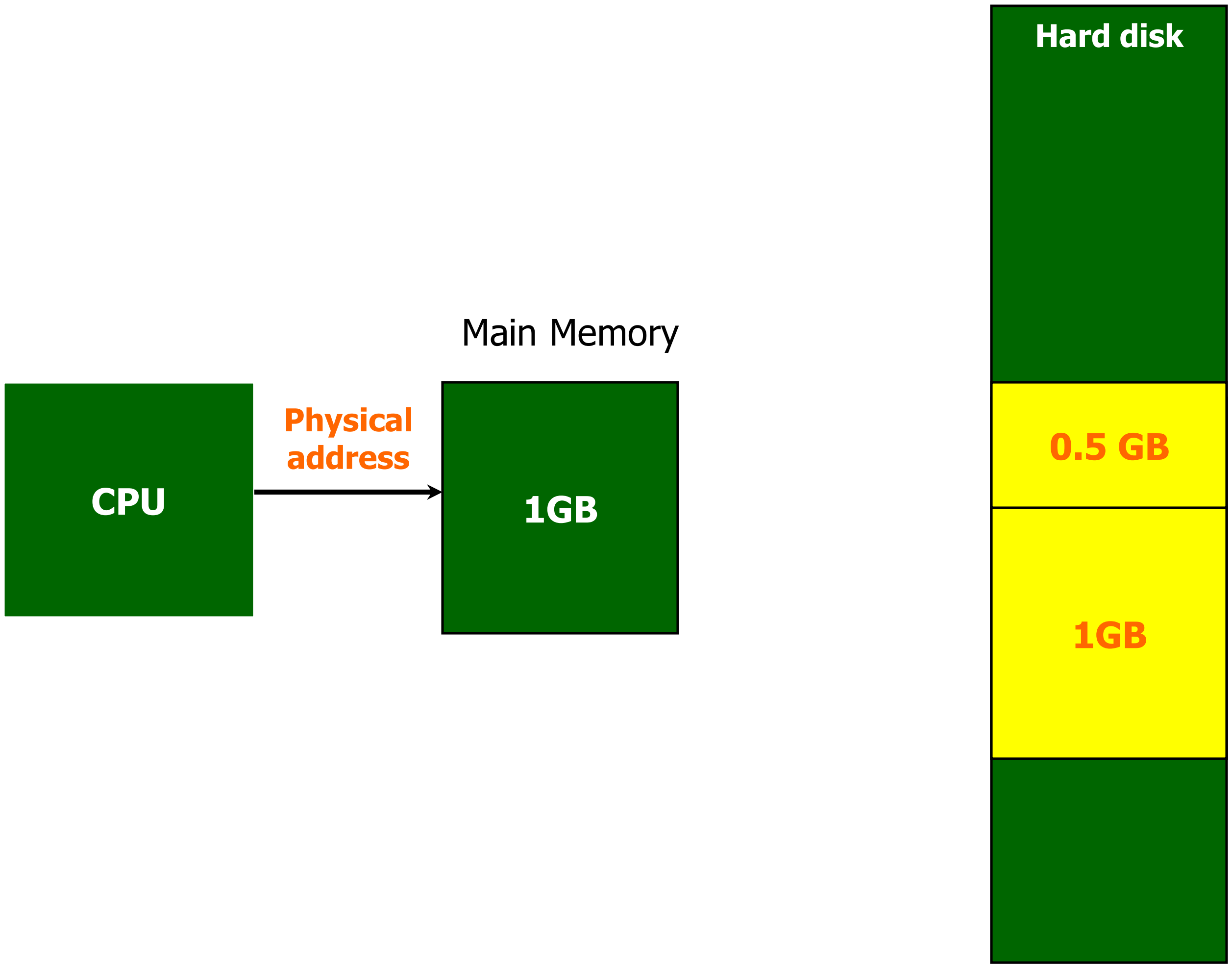


Motivation #1 of Virtual Memory



MMU: Memory Management Unit

Motivation #2 of Virtual Memory

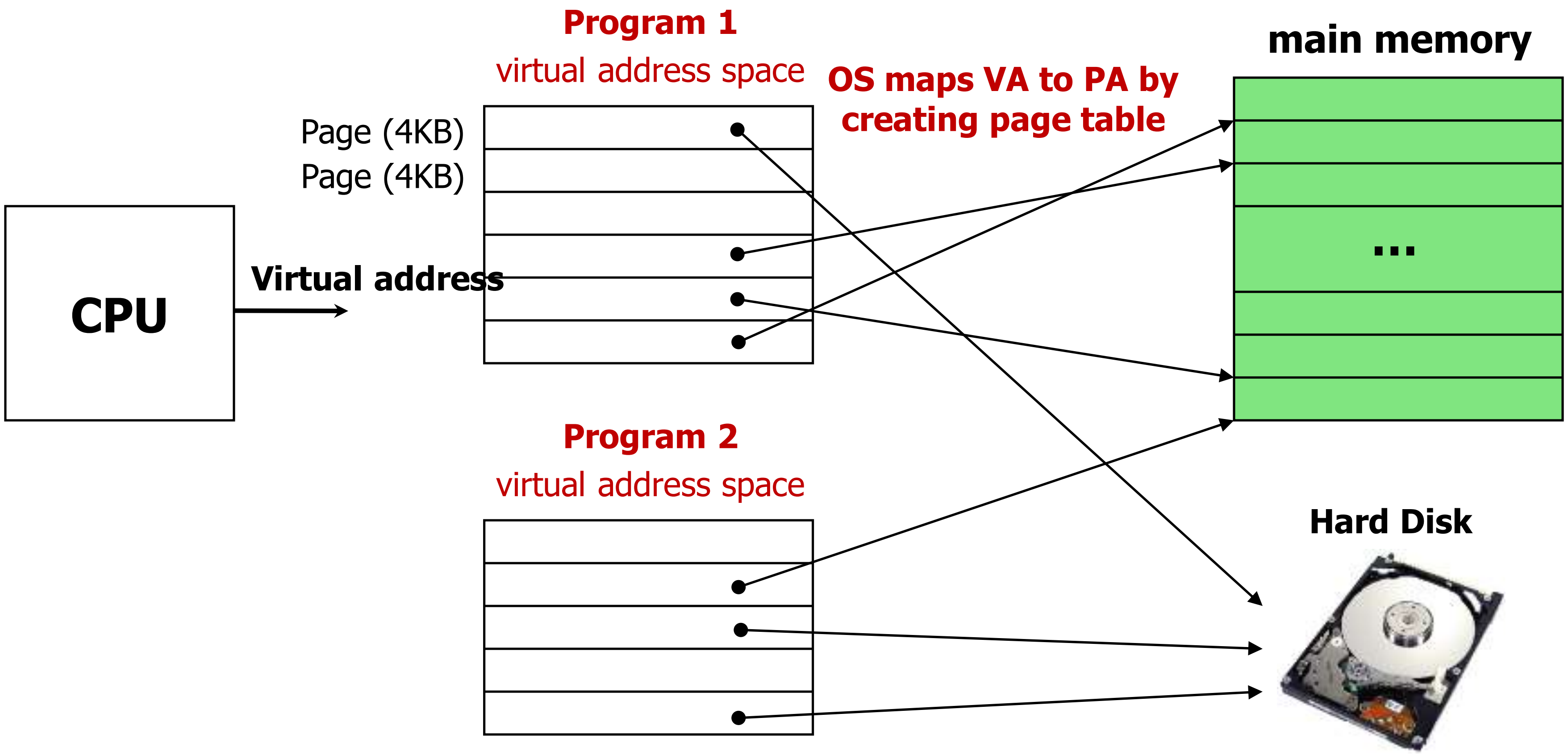


Virtual Memory

- Virtual memory is a technique provided by operating systems such as Windows and Linux
- Virtual memory uses main memory as a "cache" for secondary storage
  - Virtual memory automatically manages the 2 levels of memory hierarchy: main memory and secondary storage (HDD)
  - Virtual space is split into fixed-sized blocks, which are called pages (typically 4KB)
  - Load only required pages for execution to physical memory
  - Operating systems create page tables, which contain the translation information from virtual page to physical page

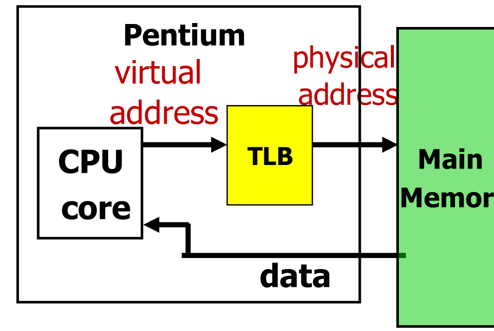
Illustration: Two Programs Sharing Physical Memory

- A program's address space is divided into pages
  - The start location of each page (either in main memory or in secondary storage) is contained in the page table



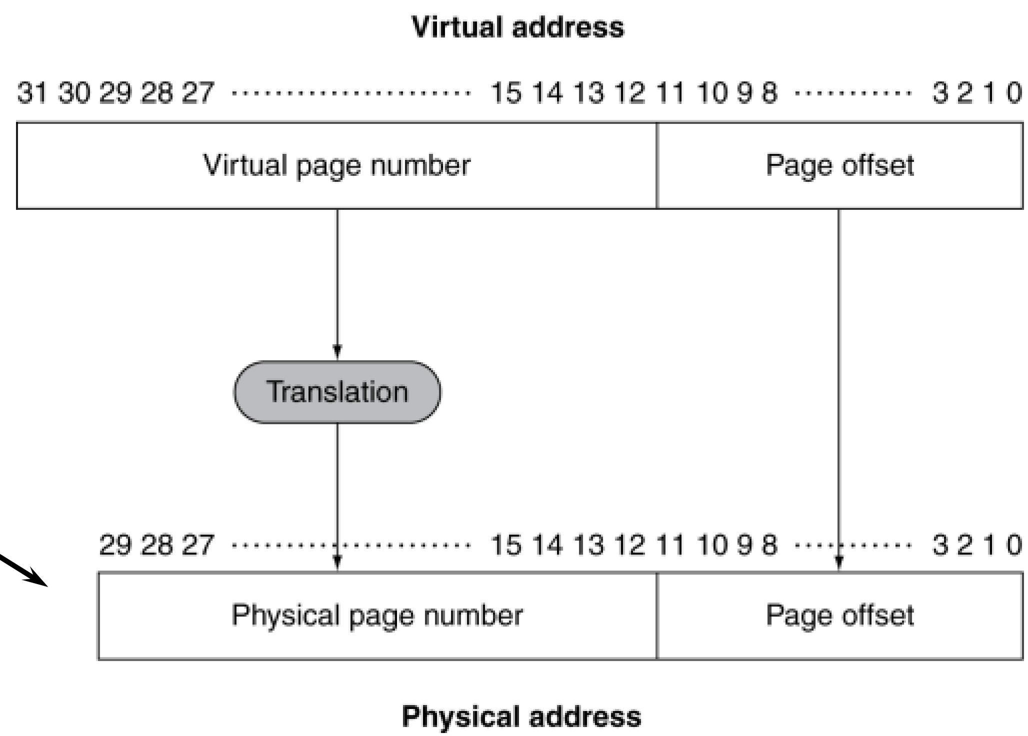
Address Translation

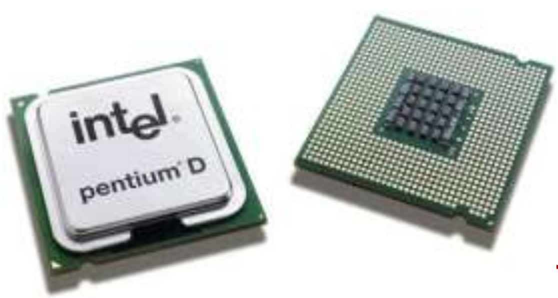
- So, each memory request first requires an address translation from the virtual space to the physical space
  - A virtual address is translated to a physical address by a combination of hardware and software



The width of the virtual address is the address width that CPU core generates. (Usually 32-bit in 32-bit CPU since PC is 32-bit wide)

The width of the physical address is the number of address pins that the processor provides





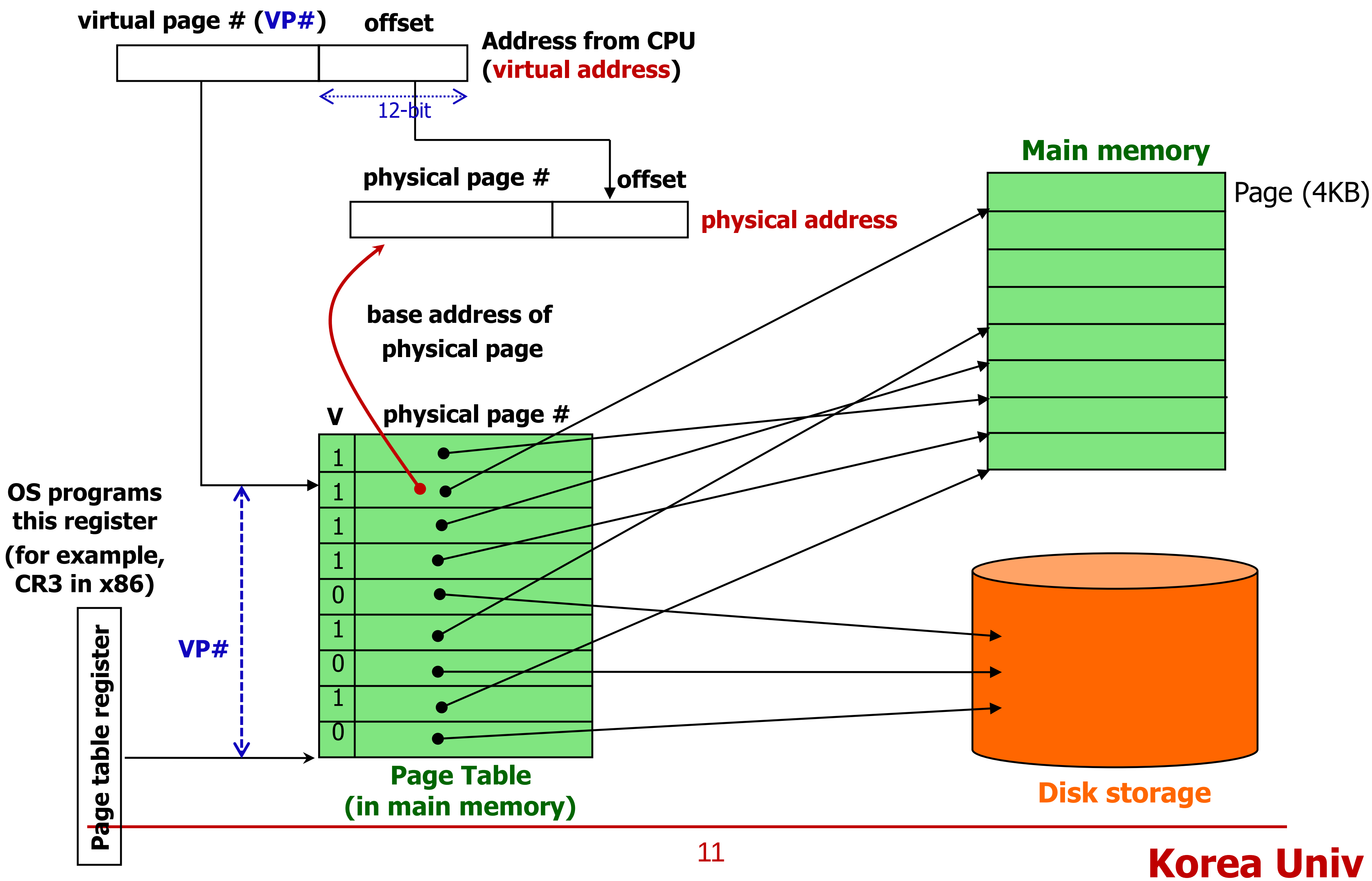
Page Tables

- Again, OS creates a page table for each process (that is, each program)

- Page table contains information about virtual to physical address translations
  - Page table is located in main memory
  - Page table is composed of page table entries (PTEs)
  - PTE is indexed by virtual page number
  - Page table register (for example, CR3 in x86) in CPU points to page table in main memory

- There are 2 cases
  - If the requested page is present in main memory, PTE stores the physical page number plus other status bits (referenced, dirty, …)
  - If the requested page is not present in main memory, PTE can refer to a location in swap space on disk
    - **Swap space** is the space on the disk reserved for the full virtual memory space of a process (program)

Address Translation Mechanism



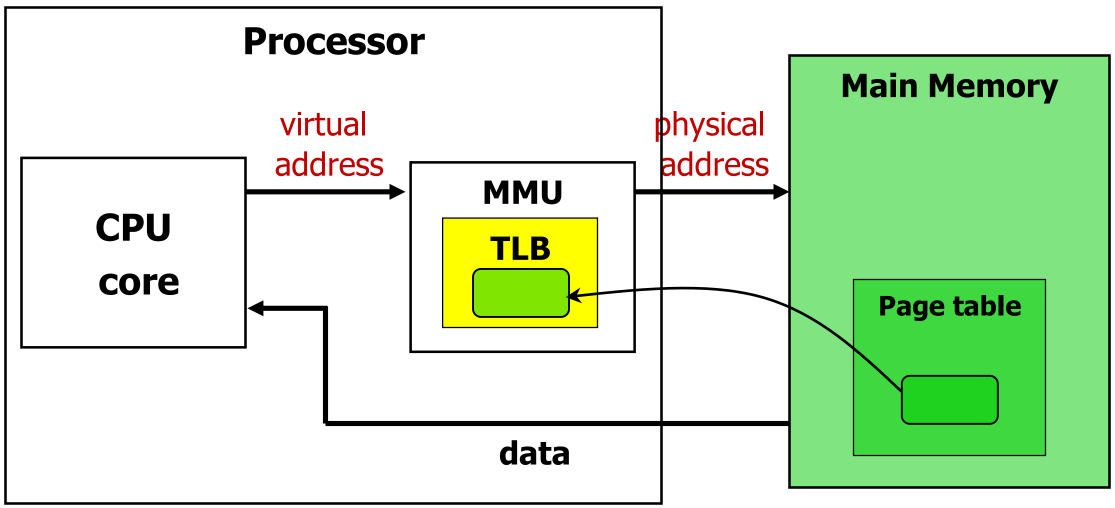
Translation Lookaside Buffer (TLB)

- Since the page tables are stored in main memory, every memory access by a program takes at least 2 transactions
  - 1^{st} one is for translating a VA to a PA
  - 2^{nd} one is for accessing data (or instruction)
  - It makes programs run very slow
  - So, there should be hardware support

- Access to page tables has a good locality
  - Once the page (4KB) is accessed, the same page would be accessed again by the program in the near future
  - So, use a fast cache for PTEs, called a Translation Look-aside Buffer (TLB), within the CPU to store recently-accessed physical page numbers

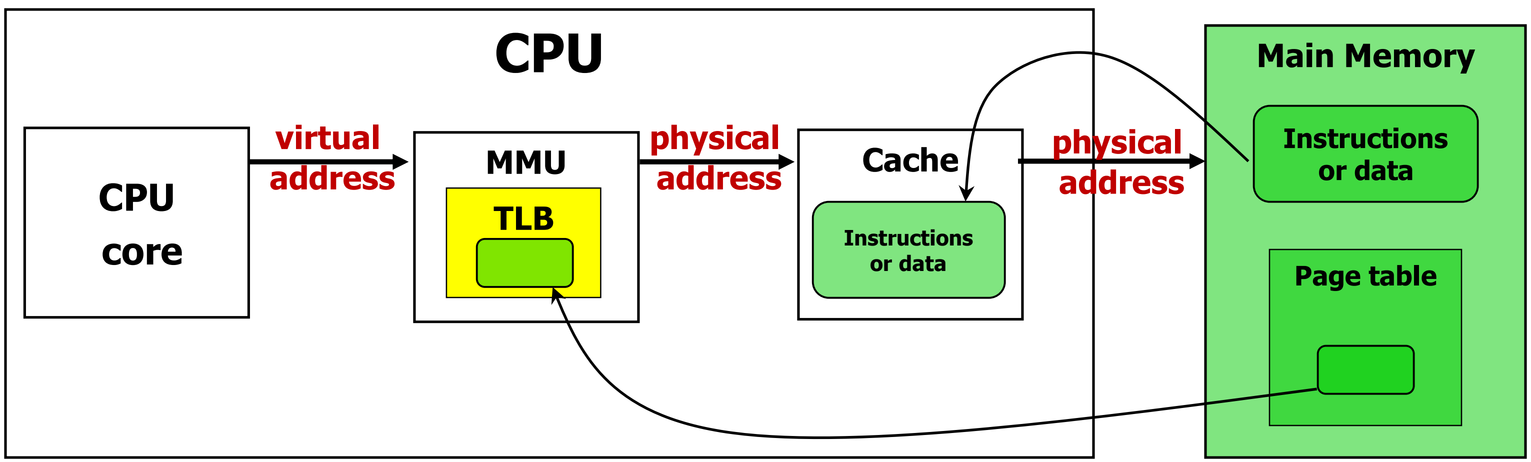
Translation Lookaside Buffer (TLB)

- TLB caches recently used PTEs from page table



TLB with a Cache

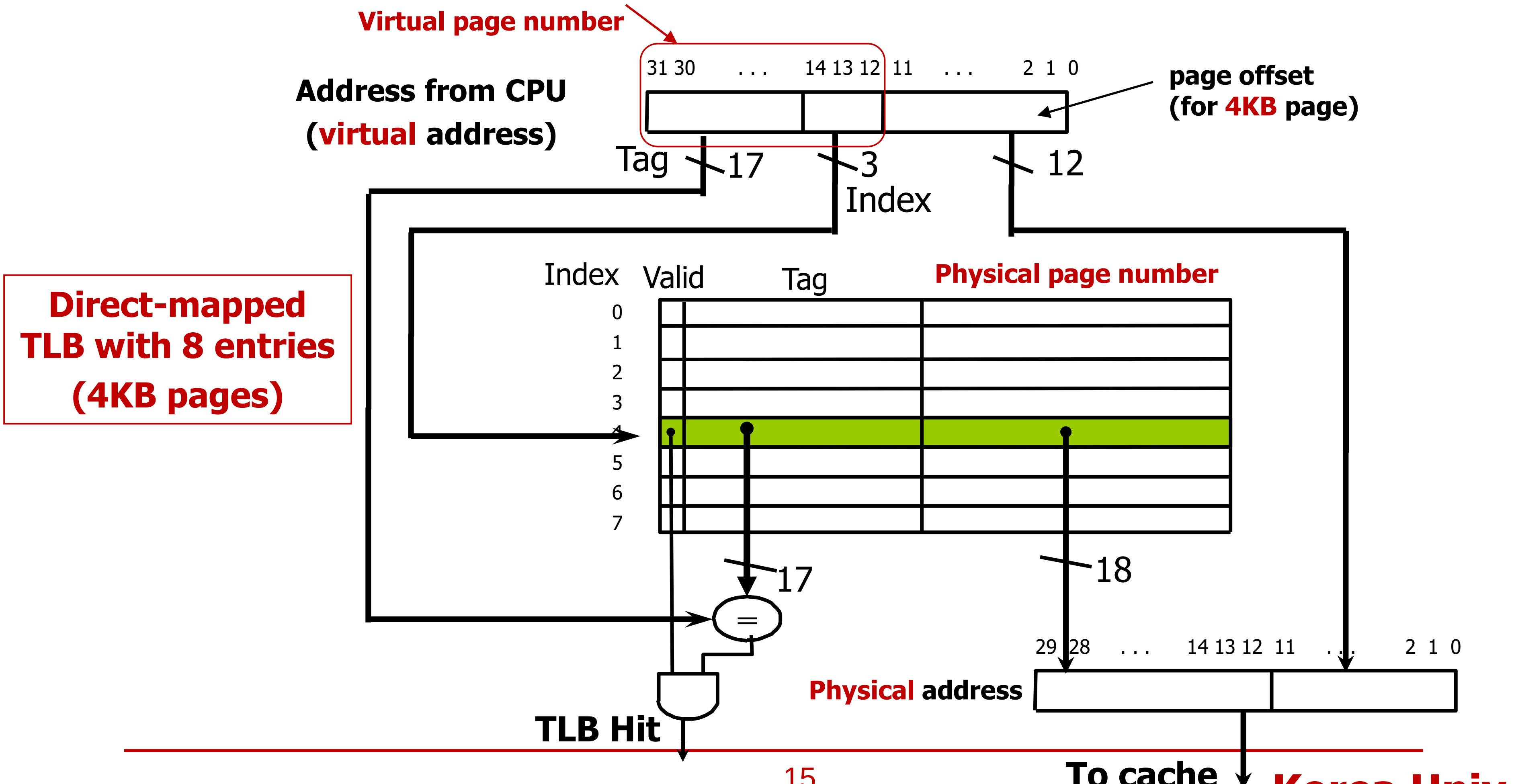
- TLB is a cache for page table
- Cache is a cache (?) for instruction and data
  - L1 cache indexing is performed with virtual address or physical address
  - L2 and L3 are typically indexed with physical address



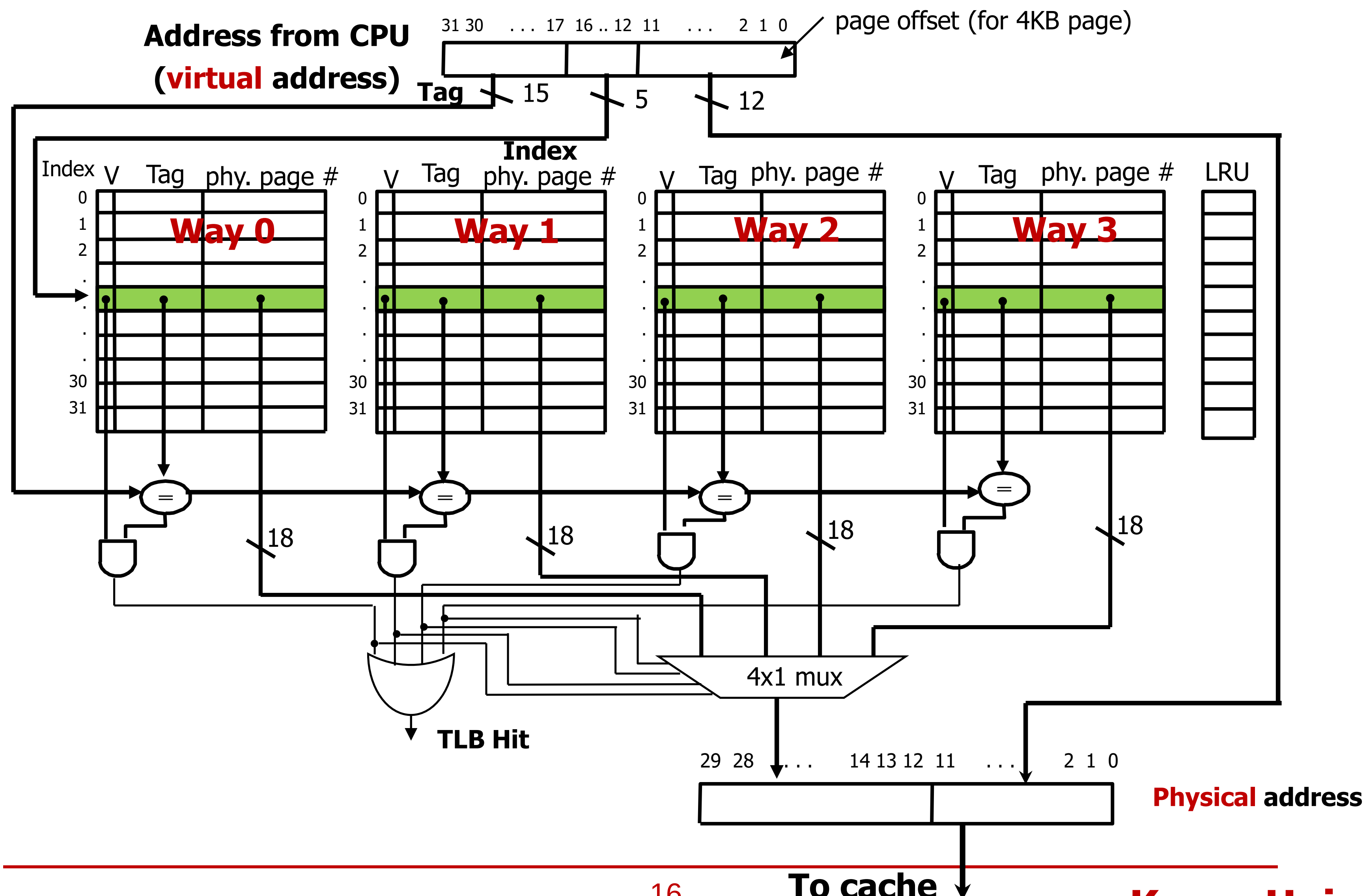
- Let's assume that, for the examples in the next 2 slides,
  - Virtual address is 32-bit wide
  - Physical address is 30-bit wide
    - What is the max. size of main memory that can be attached to the processor?

TLB Structure

- Just like caches, the TLB can be organized as direct mapped, set associative, or fully associative



TLB Example: 4-way set associative with 128 entries



TLB Terminology

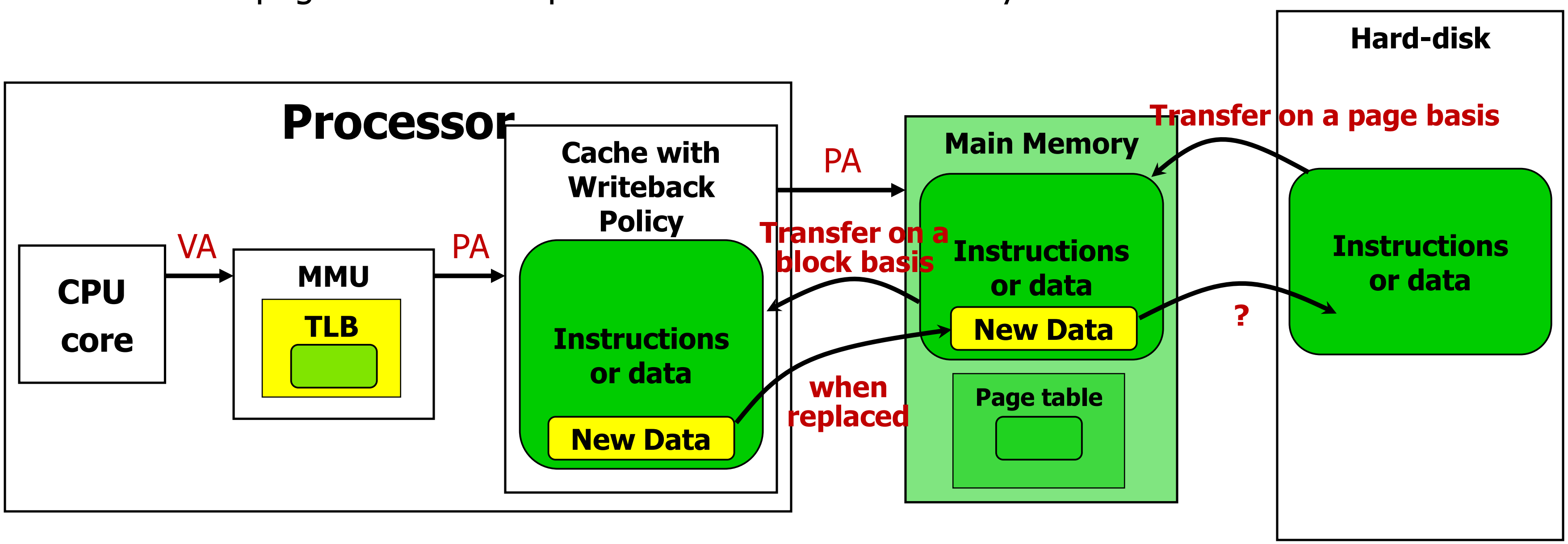
- **TLB hit**
  - An accessed page by CPU is present in TLB
- **TLB miss**
  - An accessed page by CPU is not present in TLB
  - There are 2 possible cases
    - Merely **TLB miss**
      - If the page is already loaded into main memory, then the TLB miss can be handled (in hardware or software) by loading the translation information from the page table into the TLB
      - Takes 10's of cycles to find and load the translation info into the TLB
    - **Page fault**
      - If the page is not in main memory, it's called **page fault**
      - The requested page is transferred from hard-disk to main memory via DMA (Direct Memory Access)
      - Takes 1,000,000's of cycles to service a page fault because hard-disk is extremely slow
  - TLB misses are much more frequent than page faults

TLB Misses

- 2 cases as mentioned
  - **TLB miss** if the requested page is in main memory
    - Load the corresponding PTE to TLB from page table in memory and retry
    - Could be handled in hardware
      - Can get complex for more complicated page table structures
    - Or in software
      - Raise a special exception
      - Exception handler accesses a page table in main memory and brings a page table entry to TLB
  - **Page fault** if the requested page is not in main memory (handled by OS)
    - Transfer the control to OS by interrupting the current process
    - Use faulting virtual address to find PTE
    - Locate the corresponding page on hard-disk
    - Choose a page to replace in main memory
      - If dirty, write the page to hard-disk first
    - Read the requested page from hard-disk into memory and update page table
    - Make the interrupted process runnable again
    - Then restart the faulting instruction

TLB Structure

- Write-back cache updates only a block in cache w/o updating main memory
  - So there is a dirty bit in each cache line
  - When the dirty cache line is replaced, main memory will have the up-to-date copy, but hard-disk will have stale information
- Then, how do you know which pages in main memory need to be updated to hard-disk?
  - How do you keep track of which pages in main memory have been changed?
  - Which page does OS swap out when the main memory is in full use?



TLB Structure

- TLB should keep track of which pages have been updated (written)
  - So, TLB needs a dirty bit
  - When entries with dirty bits set are replaced in TLB, the page table in main memory is also updated to indicate that the pages should be written back to hard-disk
- Some CPUs have a reference bit or use bit, which is set whenever a page is accessed

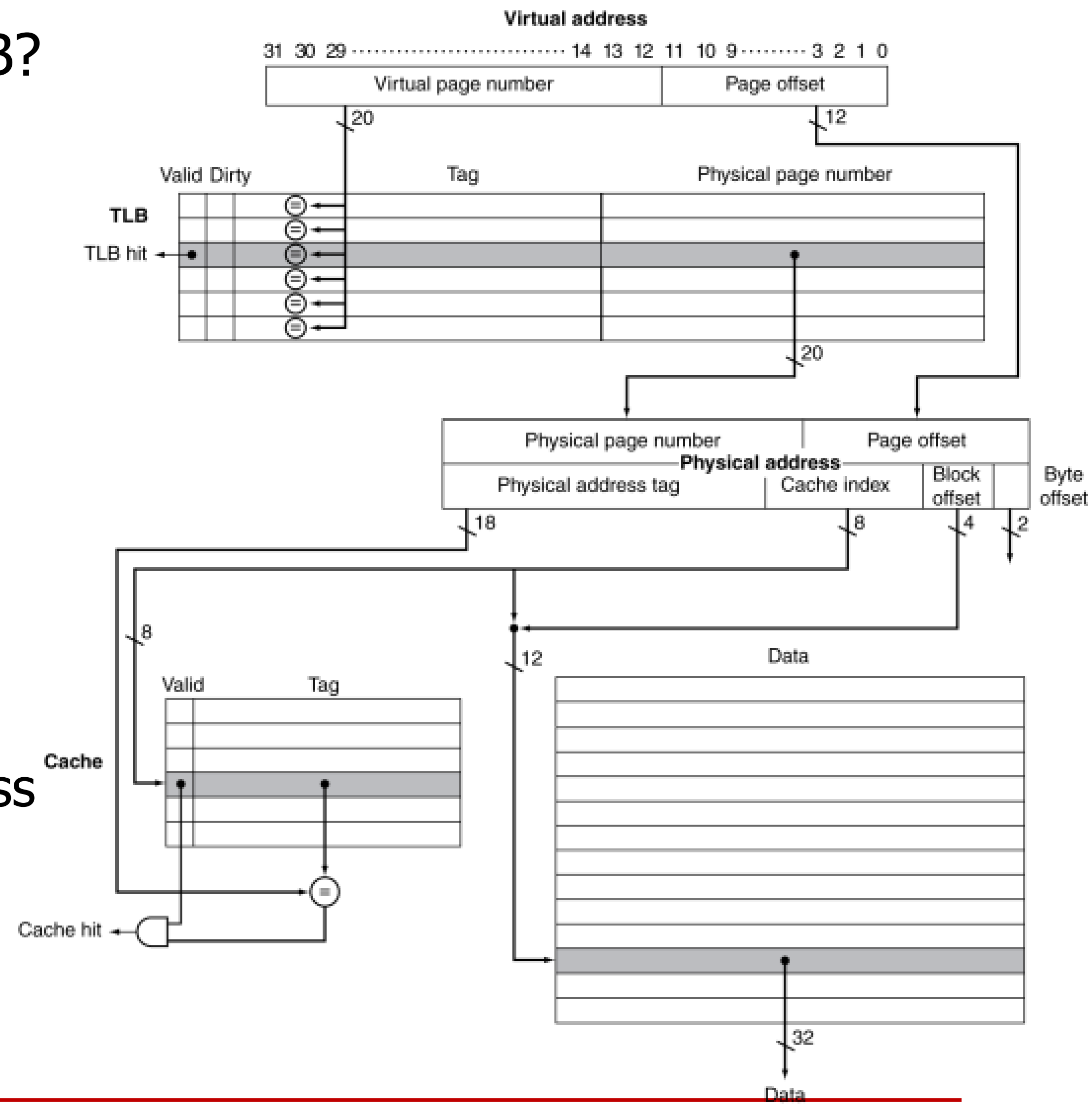
Valid	Tag	Physical page #	Dirty	Ref

Replacement and Writes

- To reduce page fault rate, operating systems prefer least-recently used (LRU) replacement
  - Reference bit (aka use bit) in PTE set to 1 on access to page by hardware
  - Periodically cleared to 0 by OS (software)
  - A page with reference bit = 0 has not been used recently
  - So, OS may swap out a page with the reference bit = 0 from main memory
- Writes to hard-disk take millions of CPU clock cycles
  - Thus, writes occur on a block (usually 4KB) basis
  - Use write-back (Write-through is impractical)
    - Perform the individual writes into the page only in main memory
  - Dirty bit in PTE is set when page is written
    - Write the page back to disk when it is displaced from main memory

TLB and Cache Interaction

- Which structure is this TLB?
  - Fully-associative
- Physically-indexed and physically-tagged cache (PIPT)
  - Need to translate before cache lookup
  - So, even a cache hit requires both a TLB access and a cache access
  - It may be pipelined to enhance throughput

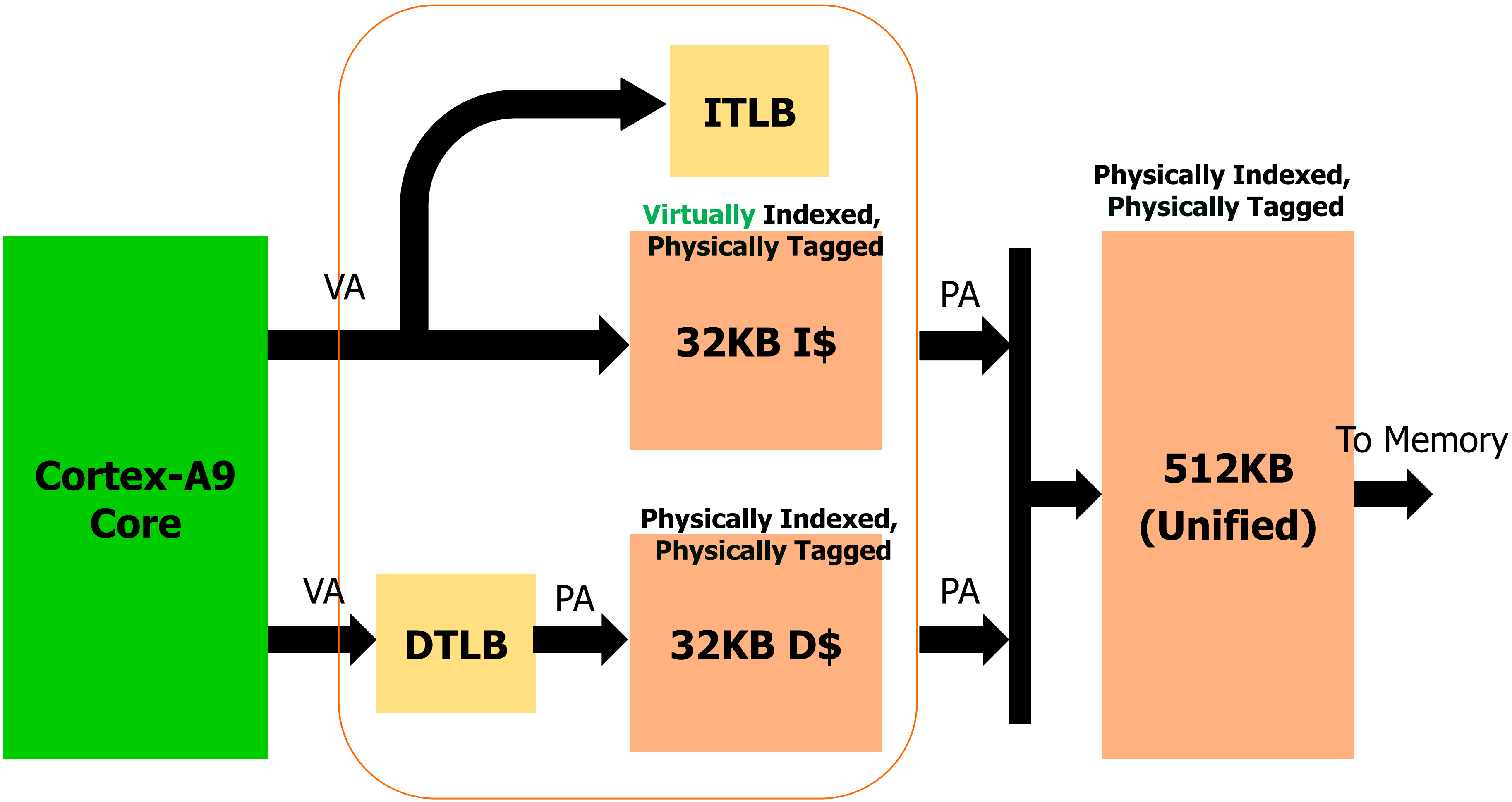


Two Machines' TLB Parameters

	Intel Nehalem	AMD Barcelona
Address sizes	48 bits (vir); 44 bits (phy)	48 bits (vir); 48 bits (phy)
Page size	4KB	4KB
TLB organization	L1 TLB for instructions and L1 TLB for data per core; both are **4-way set assoc**.; **LRU** L1 ITLB has **128 entries**, L1 DTLB has **64 entries** L2 TLB (unified) is **4-way set assoc**.; **LRU** L2 TLB has **512 entries** TLB misses handled in hardware	L1 TLB for instructions and L1 TLB for data per core; both are **fully assoc**.; **LRU** L1 ITLB and DTLB each have **48 entries** L2 TLB for instructions and L2 TLB for data per core; each are **4-way set assoc**.; round robin LRU Both L2 TLBs have **512 entries** TLB misses handled in hardware

Backup Slides

Caches in Cortex-A9



TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	Hit	Hit	Yes – what we want!
Hit	Hit	Miss	Yes – although the page table is not checked if the TLB hits
Miss	Hit	Hit	Yes – merely a TLB miss
Miss	Hit	Miss	Yes – merely a TLB miss. After retry, a miss in cache
Miss	Miss	Miss	Yes – page fault. After retry, a miss in cache
Hit	Miss	Miss/ Hit	Impossible – TLB translation not possible if page is not present in memory
Miss	Miss	Hit	Impossible – data not allowed in cache if page is not in memory

Memory Protection

- Why protect memory?
  - The most important function of virtual memory is to allow sharing of a single main memory by multiple processes
  - Different processes (tasks) can share parts of their virtual address spaces
    - To allow another process (say P1) to read a page owned by process P2, P2 would ask OS to create a page table entry for a virtual page in P1's address space that points to the same physical page that P2 wants to share
  - So, there should be memory protection among these processes
  - Any bits that determine the access rights for a page must be included in both the page table and the TLB, because the page table is accessed only on a TLB miss
    - Write access bit can be used to restrict the sharing to just read sharing
    - The write access bit should be in both the page table and the TLB

Handling a TLB Miss

- Consider a TLB miss for a page that is present in memory (i.e., the Valid bit in the page table is set)
  - A TLB miss (or a page fault exception) must be asserted by the end of the same clock cycle that the memory access occurs so that the next clock cycle will begin exception processing

Register	CP0 Reg #	Description
`EPC`	14	Where to restart after exception
`Cause`	13	Cause of exception
`BadVAddr`	8	Address that caused exception
`Index`	0	Location in TLB to be read/written
`Random`	1	Pseudorandom location in TLB
`EntryLo`	2	Physical page address and flags
`EntryHi`	10	Virtual page address
`Context`	4	Page table address & page number

A MIPS Software TLB Miss Handler

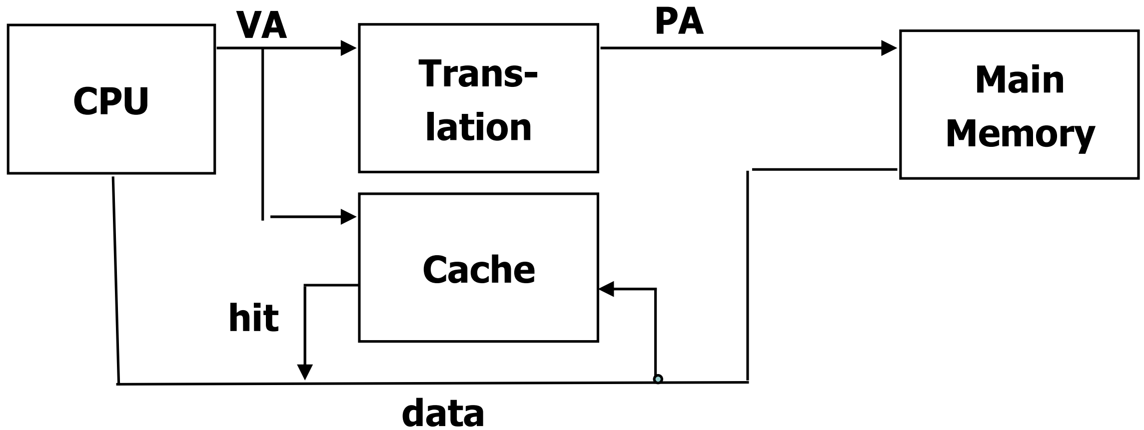
- When a TLB miss occurs, the hardware saves the address that caused the miss in `BadVAddr` and transfers control to 8000 0000_{hex}, the location of the TLB miss handler

```
TLBmiss:
 mfc0  $k1, Context #copy addr of PTE into $k1
 lw    $k1, 0($k1)  #put PTE into $k1
 mtc0  $k1, EntryLo #put PTE into EntryLo
 tlbwr              #put EntryLo into TLB at random
 eret               #return from exception
```

- `tlbwr` copies from `EntryLo` into the TLB entry selected by the control register `Random`
- A TLB miss takes about a dozen clock cycles to handle

Why Not a Virtually Indexed and Virtually Tagged (VIVT) Cache?

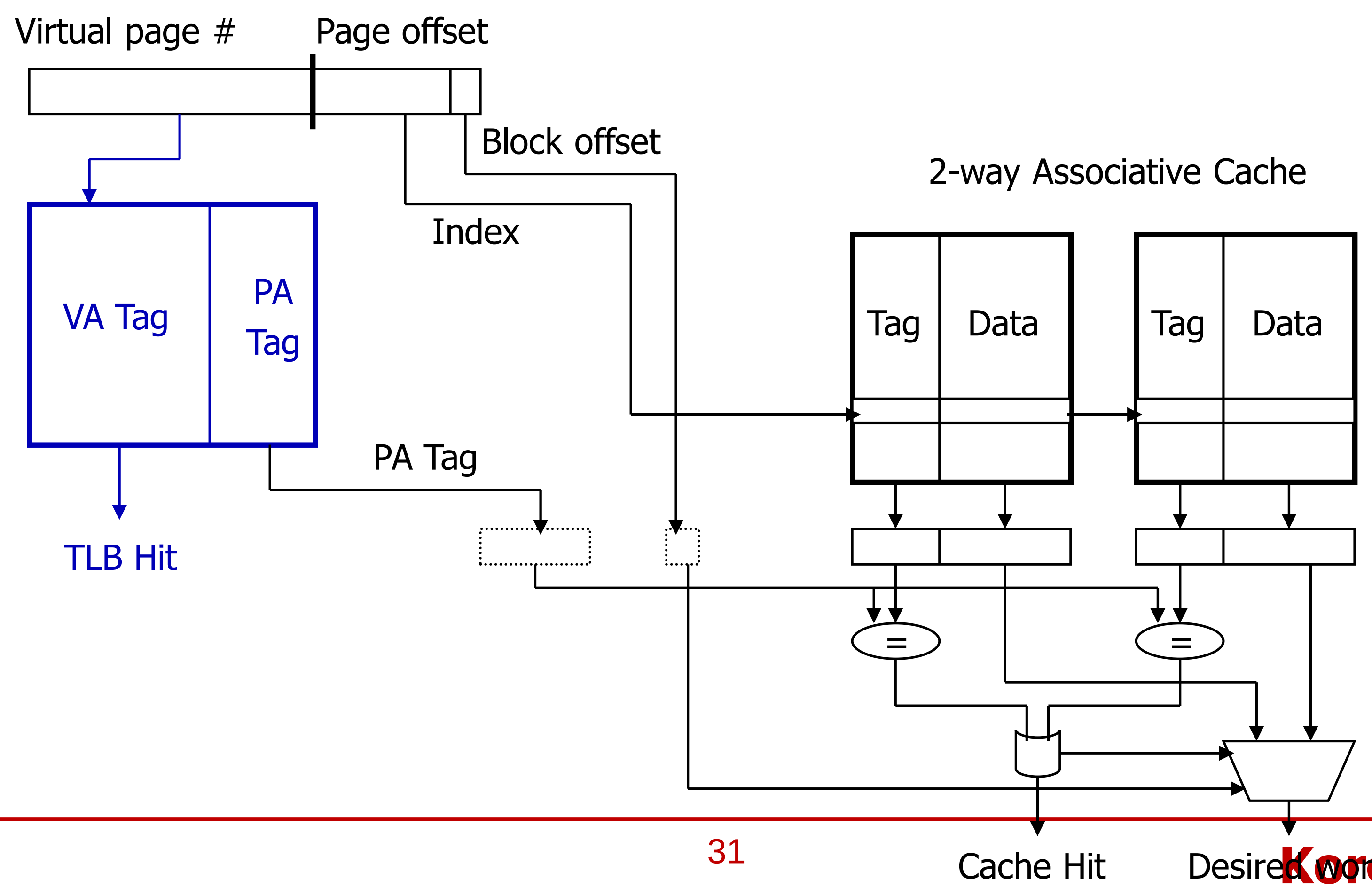
- A virtually addressed cache would only require address translation on cache misses



- **Aliasing** problem
  - 2 programs that are sharing data will have 2 different virtual addresses for the same physical address
  - So, cache could have 2 copies of the shared data and there are 2 entries in the TLB
    - It would lead to coherence issues
    - Must update all cache entries with the same physical address. Otherwise the memory becomes inconsistent

Reducing Translation Time in PIPT

- Can overlap the cache access with the TLB access
  - Works when the high order bits of the VA are used to access the TLB while the low order bits are used as index into cache



Motivation #1