Virtual memory

1. Accommodate multi-programming with limited RAM!!!
2. Give all programs unlimited memory space !!!

A few questions to think about

- can a program written for 8051, run on 8085 or 8086 or PowerPC or any other processor? (your question should be??)
- What about the memory size of the program; Can it fit in the given address space?
- should I think about the available RAM capacity on the target system before I write a program? (available RAM can be less than the max capacity)
- can I run multiple programs on the same processor?
  - Since program must be loaded onto RAM for it to run, what about the memory requirement of each of the programs; how can I serve all the programs with the available RAM (fixed capacity)?
  - will the programs share the available RAM?
- can I load my program at any location; is there a specific location demand from the program or am I free to chose any location that is available?
  - What if I use direct addresses for my data addressing and code addressing?
  - Relocatable code

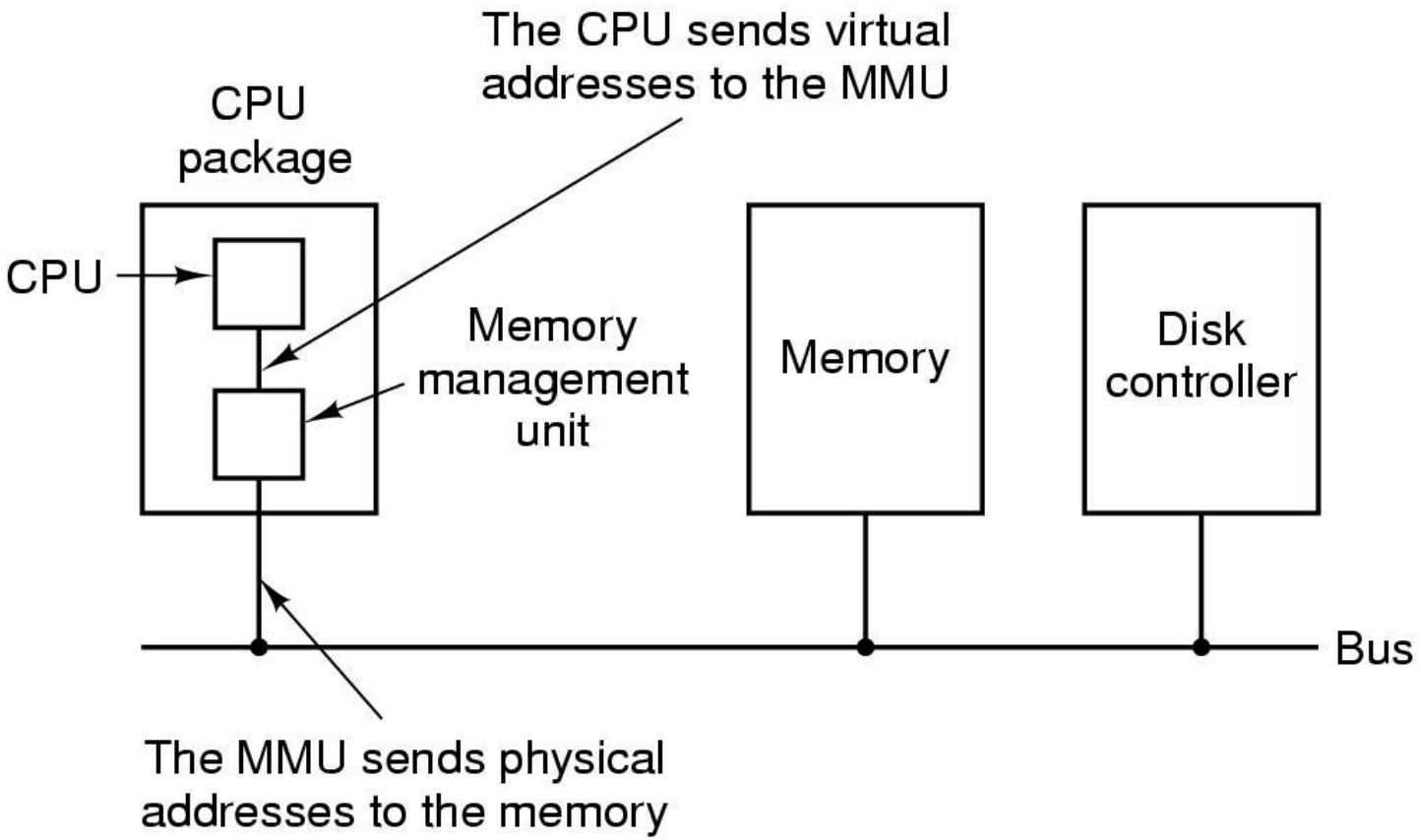
Virtual Memory

- Virtual memory achieves a complete separation of logical and physical address-spaces
- Today, typically a virtual address is 32 bits, this allows a process to have "*virtually*" 4GB of addressable memory
  - Physical memory is much smaller than this, and varies from machine to machine

Terminology in Virtual Memory

- Each process has its own private "virtual address space" (e.g., 2^{32} Bytes); program uses "virtual addresses" for locating data and code
- Each computer has a "physical address space" (e.g., 128 MegaBytes DRAM); also called "real memory"
- Address translation: mapping virtual addresses to physical addresses
  - Allows multiple programs to use (different chunks of physical) main memory at the same time
  - Also allows some chunks of virtual memory to be represented on disk, not in main memory (to exploit memory hierarchy)

Virtual Memory

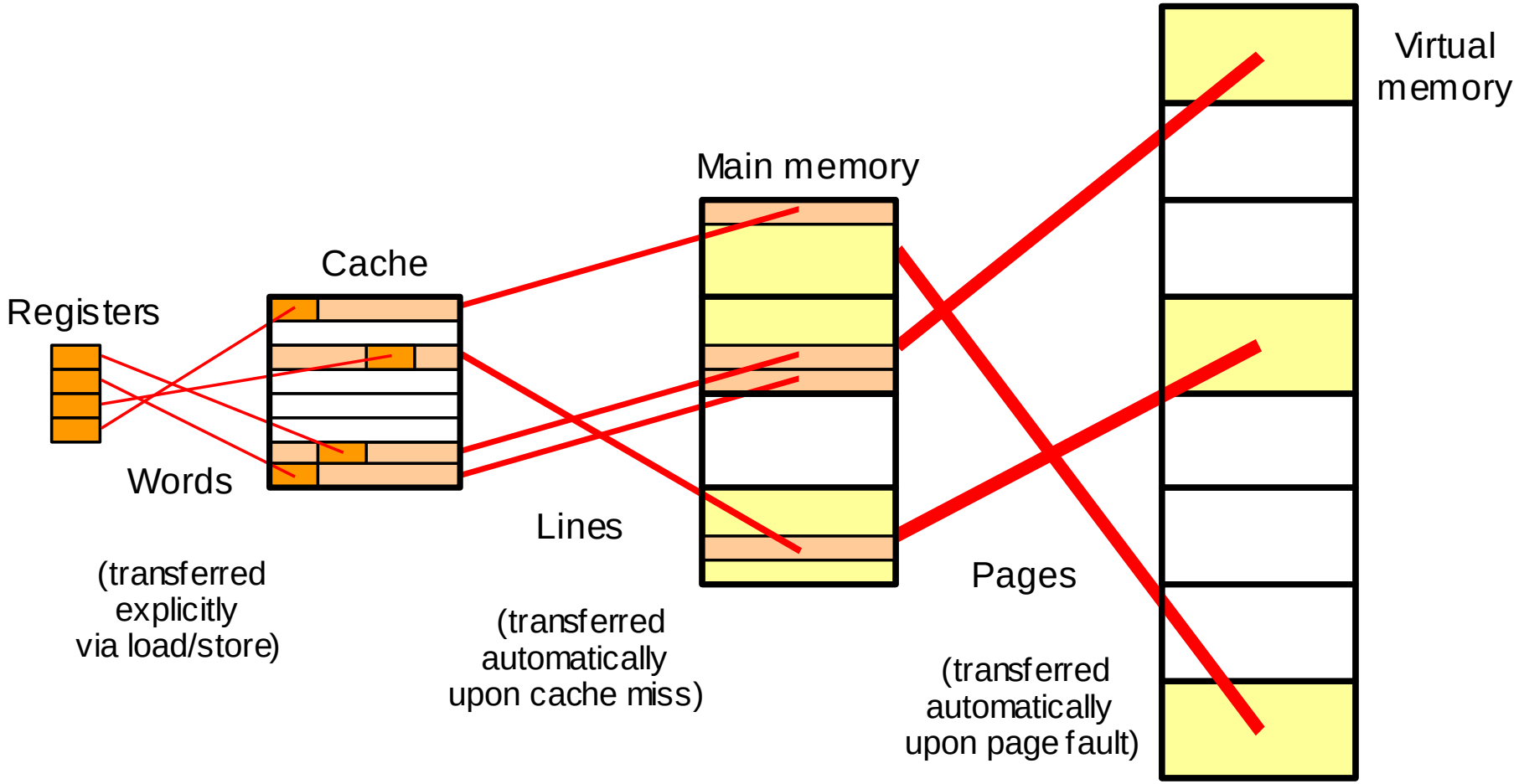


Position and function of MMU

Virtual Memory

- Uses main memory as a “cache” for secondary memory
  - Allows efficient and safe sharing of memory among multiple programs
  - Provides the ability to easily run programs larger than the size of physical memory
  - Simplifies loading a program for execution by supporting code relocation (i.e., the code can be loaded anywhere in main memory)
- What makes it work? – again the Principle of Locality
  - A program is likely to access a relatively small portion of its address space during any period of time
- Each program is compiled into its own address space – a “virtual” address space
  - During run-time, each virtual address must be translated to a physical address (an address in main memory)

Memory Hierarchy: The Big Picture



Data movement in a memory hierarchy.

Virtual Memory

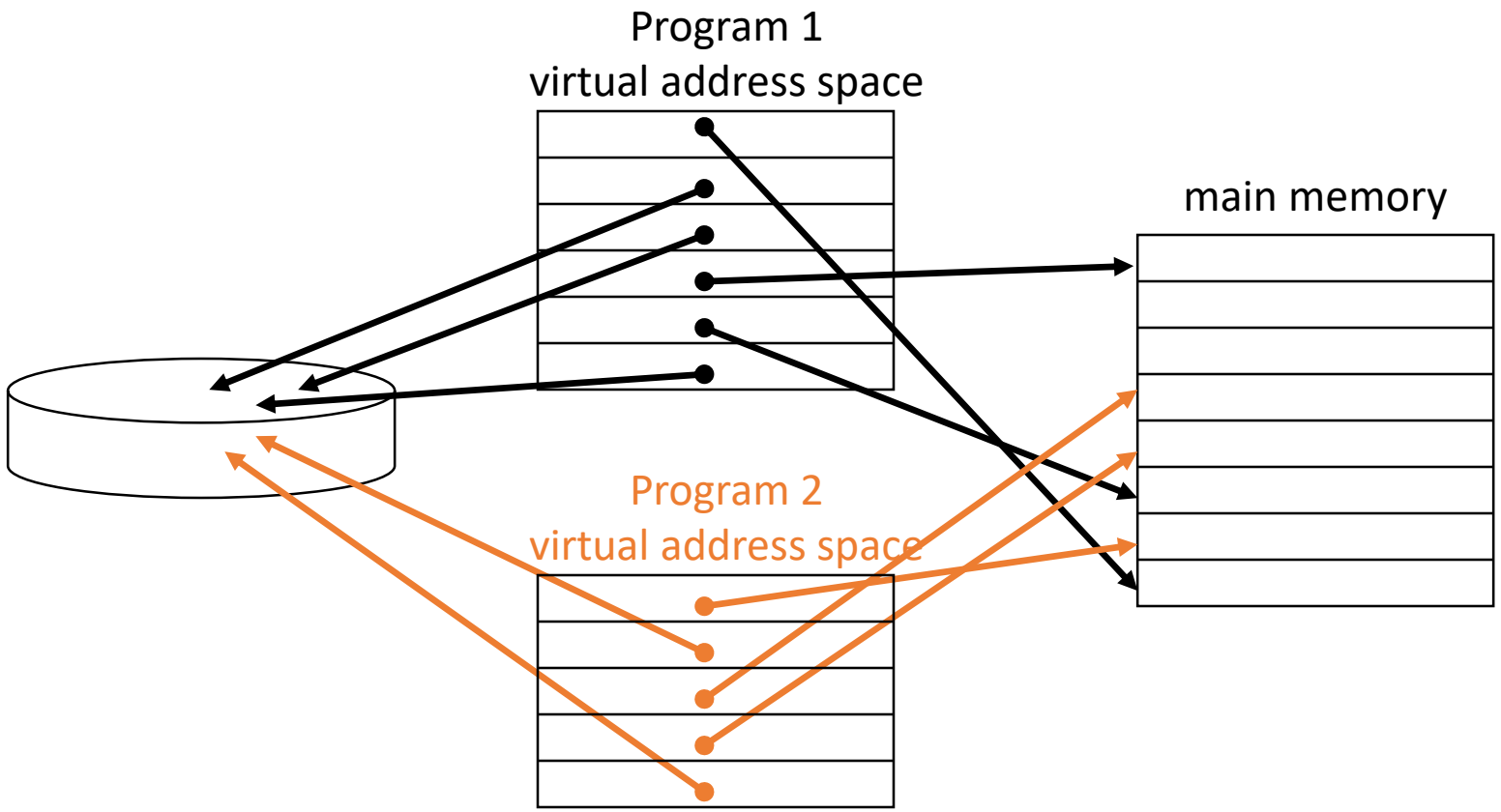
- A process may be broken up into chunks that do not need to be located contiguously in main memory
- No need to load all chunks of a process in main memory
- A process may be swapped in and out of main memory such that it occupies different regions at different points of time
- **Advantages**:
  - More processes can be maintained in main memory
  - With so many processes in main memory, it is very likely that a process will be in the Ready state at any particular time (*CPU efficiency*)
  - A process may be larger than all of main memory

Comparing cache and VM

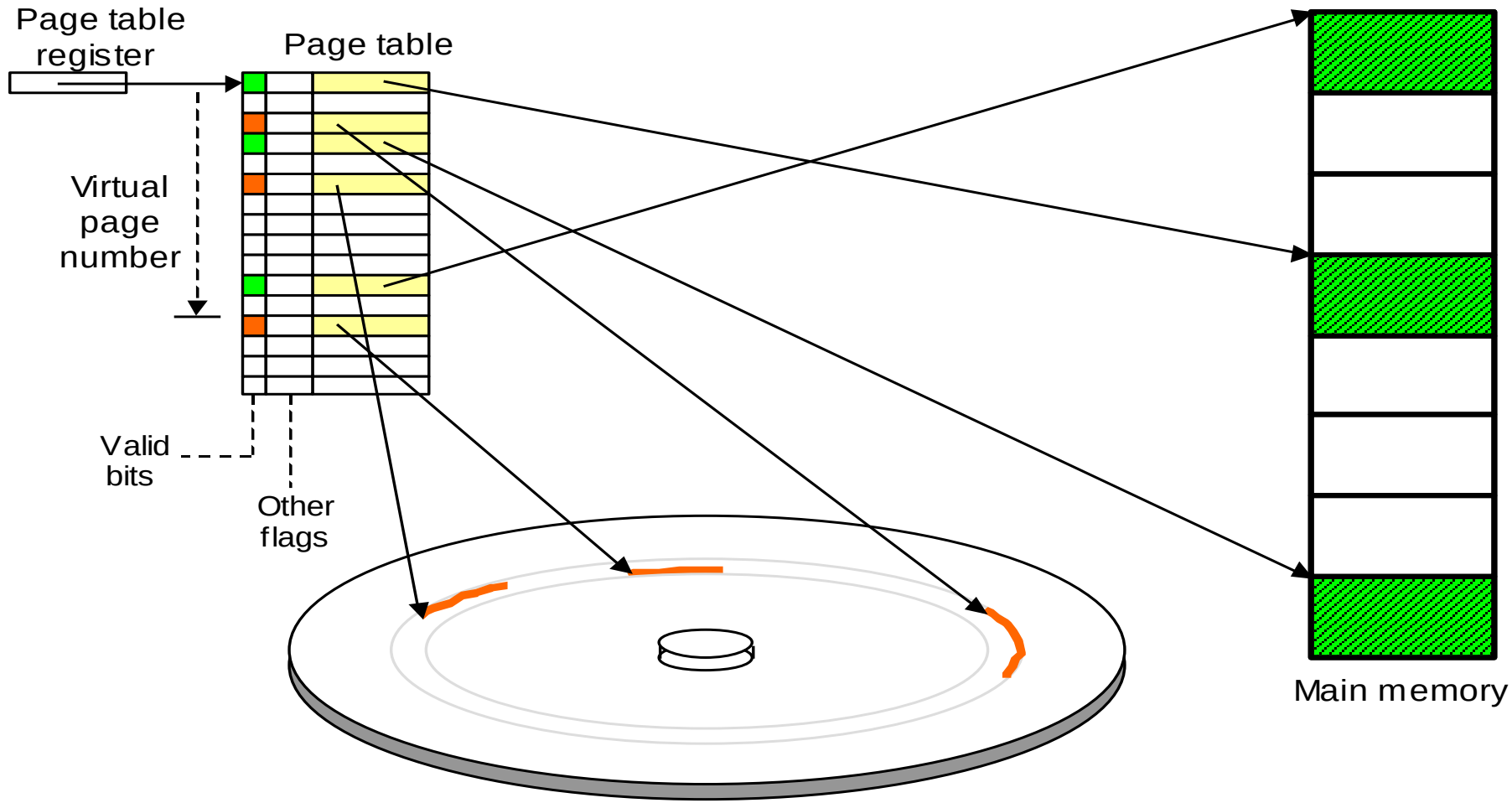
- Caching
  - **To address bottleneck between CPU and Memory**
  - **Direct**
  - **Associative**
  - **Set Associate**
- Virtual Memory Systems
  - **To make programs to think that they have whole of memory that they need**
  - **To address size constraint between Memory & Disk**
  - **Pages**
  - **Translation Lookaside Buffer**

Two Programs Sharing Physical Memory

- A program's address space is divided into pages (page has a fixed size)
- The starting location of each page (either in main memory or in secondary memory) is contained in the program's page table (page table not shown in the diagram below)



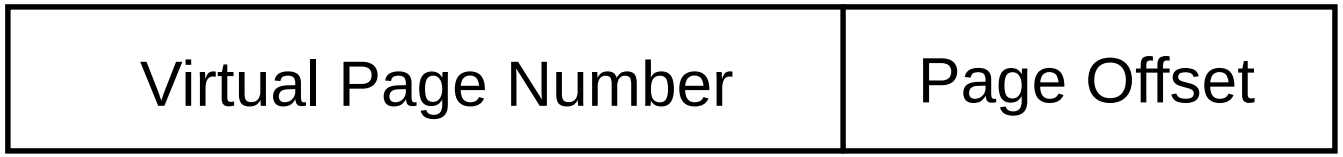
Page Tables and Address Translation



The role of page table in the virtual-to-physical address translation process.

How to Perform Address Translation?

- Virtual Memory divides memory into equal sized pages
- Virtual Memory address is divided into two fields;
  - Virtual page number
  - Page offset
- offsets within the pages do not change

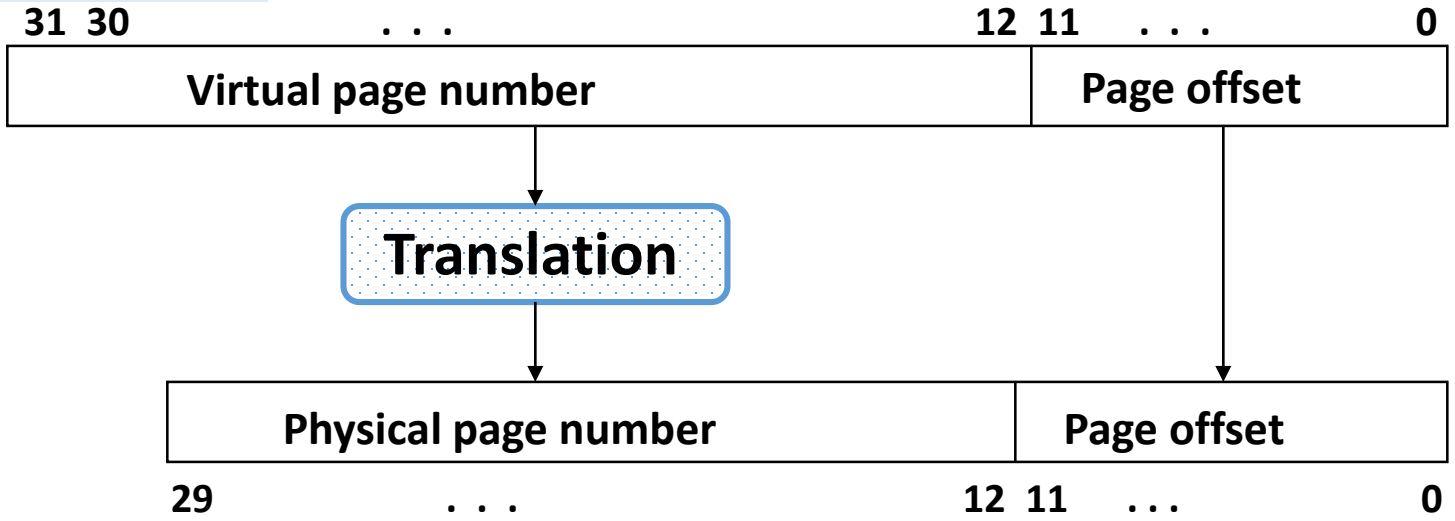


virtual address

Address Translation

- A virtual address (logical address or programmer's address) is translated to a physical address (main memory address) by a combination of hardware and software

Virtual Address (VA)



Physical address space is 1 GB

Page size is 4K

Virtual address space is 4 GB

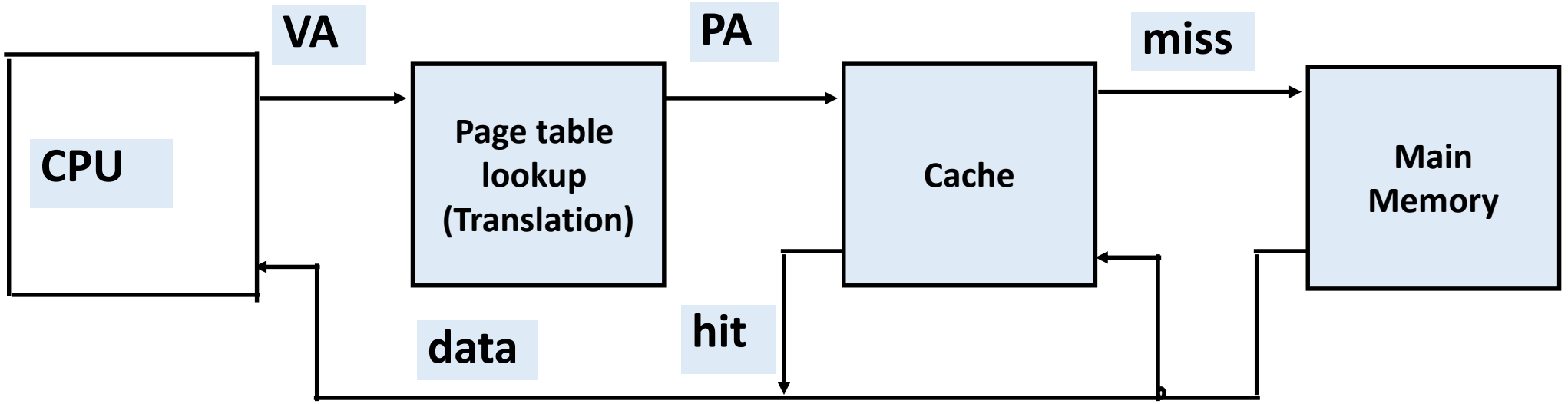
Physical Address (PA)

- So each memory request *first* requires an address translation from the virtual space to the physical space
- A virtual memory miss (i.e., when the page is not in main memory) is called a ***page fault***

How to Translate Fast?

- Problem: Virtual Memory requires two memory accesses!
  - Page Table is in physical memory! => 2 main memory accesses!
  - one to translate Virtual Address into Physical Address (page table lookup)
  - one to transfer the actual data (cache hit)
- Observation: since there is locality in pages of data, there must be locality in virtual addresses of those pages!
- Why not create a cache of virtual to physical address translations to make translation fast? (smaller is faster)
- such a “page table cache” is called a Translation Lookaside Buffer, or TLB

Virtual Addressing with a Cache



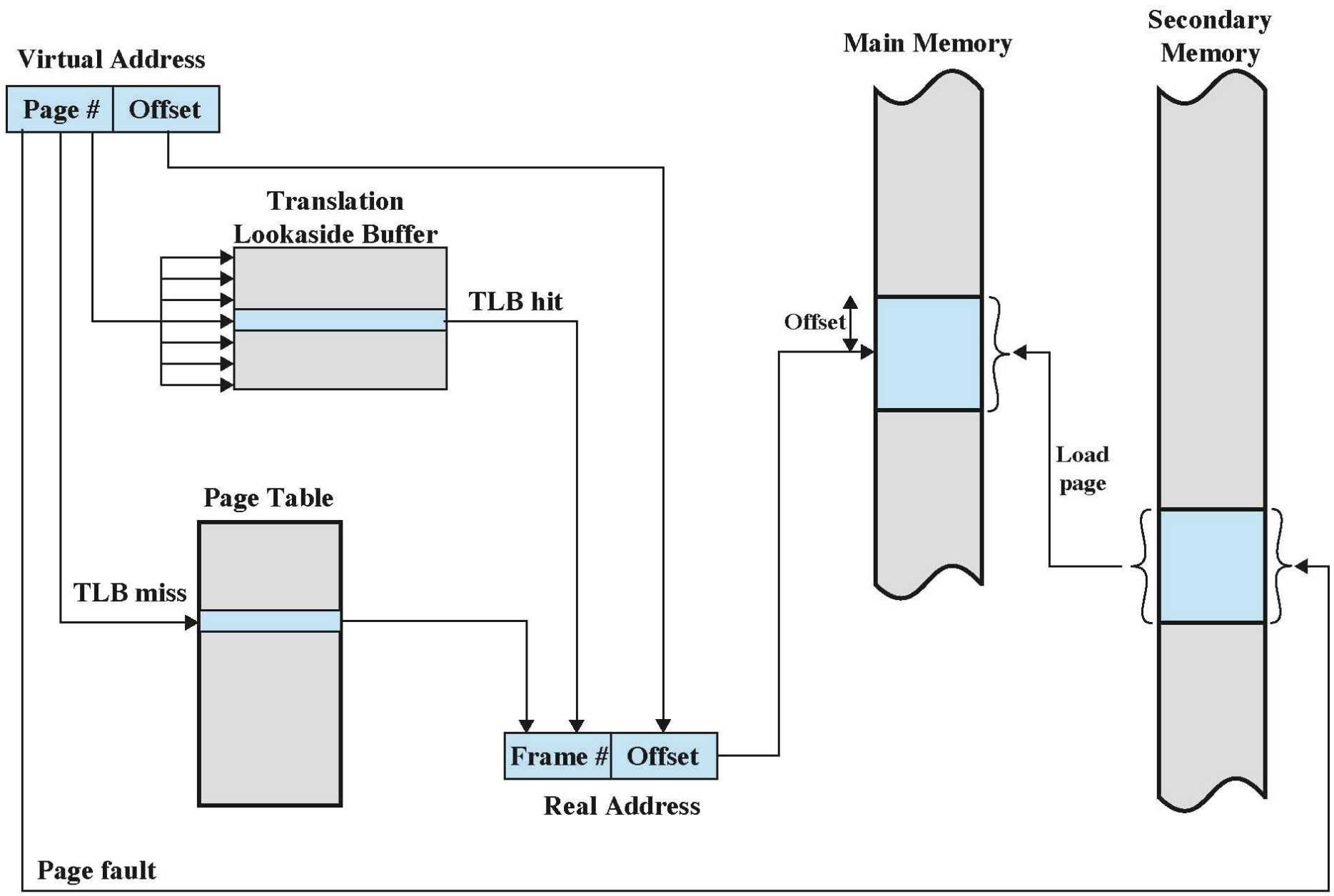
- Translation Lookaside Buffer (TLB) –a small cache that keeps track of recently used address mappings to avoid accessing page table

Translation Lookaside Buffer

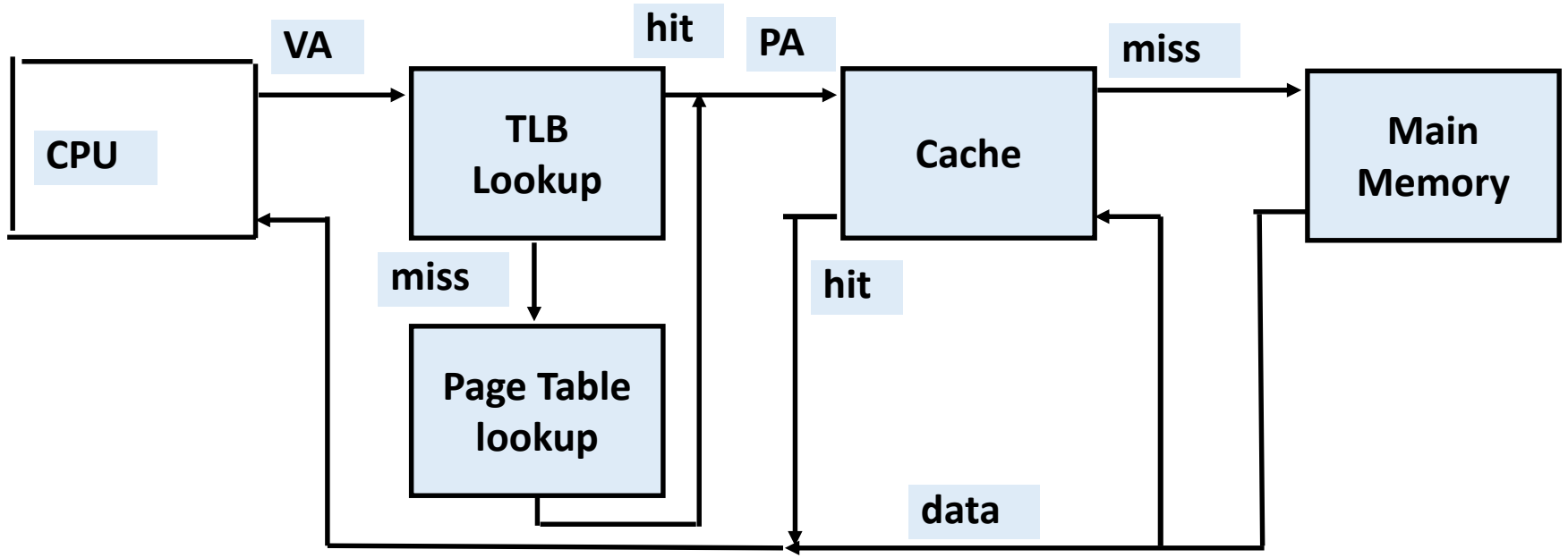
- Functions same way as a memory cache. TLB stores most recently used page table entries. Stores virtual page numbers and the mapping for it. Uses associative mapping (i.e. any virtual page number maps to any TLB index)

- Given a virtual address, processor searches the TLB first.
- If page table entry is present (a ***hit***), the frame number is retrieved and the real address is formed.
- If page table entry is not found in the TLB (a ***miss***), the virtual page number is used to index the page table, and TLB is updated to include the new page entry.

TLB working showing TLB hit, TLB miss and page fault



A TLB in the Memory Hierarchy



- A TLB miss – is it a page fault or merely a TLB miss?
  - If the page is 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
  - If the page is not in main memory, then it's a true page fault
    - Takes 1,000,000's of cycles to service a page fault
- TLB misses are much more frequent than true page faults

TLB Event Combinations

TLB	Page Table	Cache	Possible? Under what circumstances?
Hit	(Hit) DC	Hit	Yes – this is what we want! although the page table is not checked if the TLB hits
Hit	Hit	Miss	Yes – TLB hits, but data is not in cache; cache miss
Miss	Hit	Hit	Yes – TLB miss, PA is in page table
Miss	Hit	Miss	Yes – TLB miss, PA is in page table, but data not in cache
Miss	Miss	Miss	Yes – page fault
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