Inf2C - Computer Systems
Lecture 14-15
Virtual Memory

Boris Grot

School of Informatics

University of Edinburgh

Previous lecture: Memory hierarchy

- Main idea: exploit locality in memory references to create the illusion of a fast & large memory
  – Temporal vs spatial locality
- Memory hierarchy levels: registers, cache (≥1 levels), main memory, disk
- Cache: hardware-managed storage
  – Exploits temporal & spatial locality
  – Fully-associative vs direct mapped

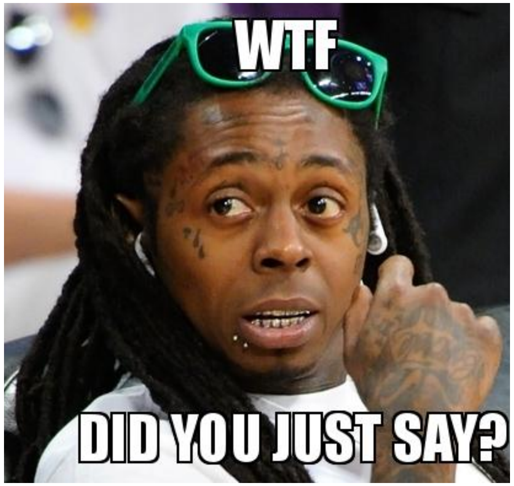
Coursework 2: cache simulator

Explore the impact of the following design choices:

- Cache size
- Block size
- Fully-associative vs direct-mapped
- Replacement policy

Due: Wed, Nov 23, 4pm

Coursework 2: it's only a simulator!



- The caches will not store data
- Cache size refers only to the size of the data portion
- All addresses are physical
  – No virtual-to-physical address translation
- You are building a simulator
  – Not the same as real hardware
  – E.g., use a built-in C data type for the valid "bit"

Coursework 2: other issues

- Your code additions are <u>not</u> restricted to the two places we've indicated
  - You must have a modular design (i.e., functions)
- Start the coursework by understanding Tutorial 4 questions 2 & 3
  - Read the book, notes, and slides
  - Read others' questions on Piazza
    - Only if you can't find the answer in any of the above should you post a new question
- Make sure your code compiles & runs on DICE

Lecture 14-15: Virtual memory

- Motivation
- Overview
- Address translation
- Page replacement
- Fast translation – TLB

Motivation

Virtual memory addresses two main problems:

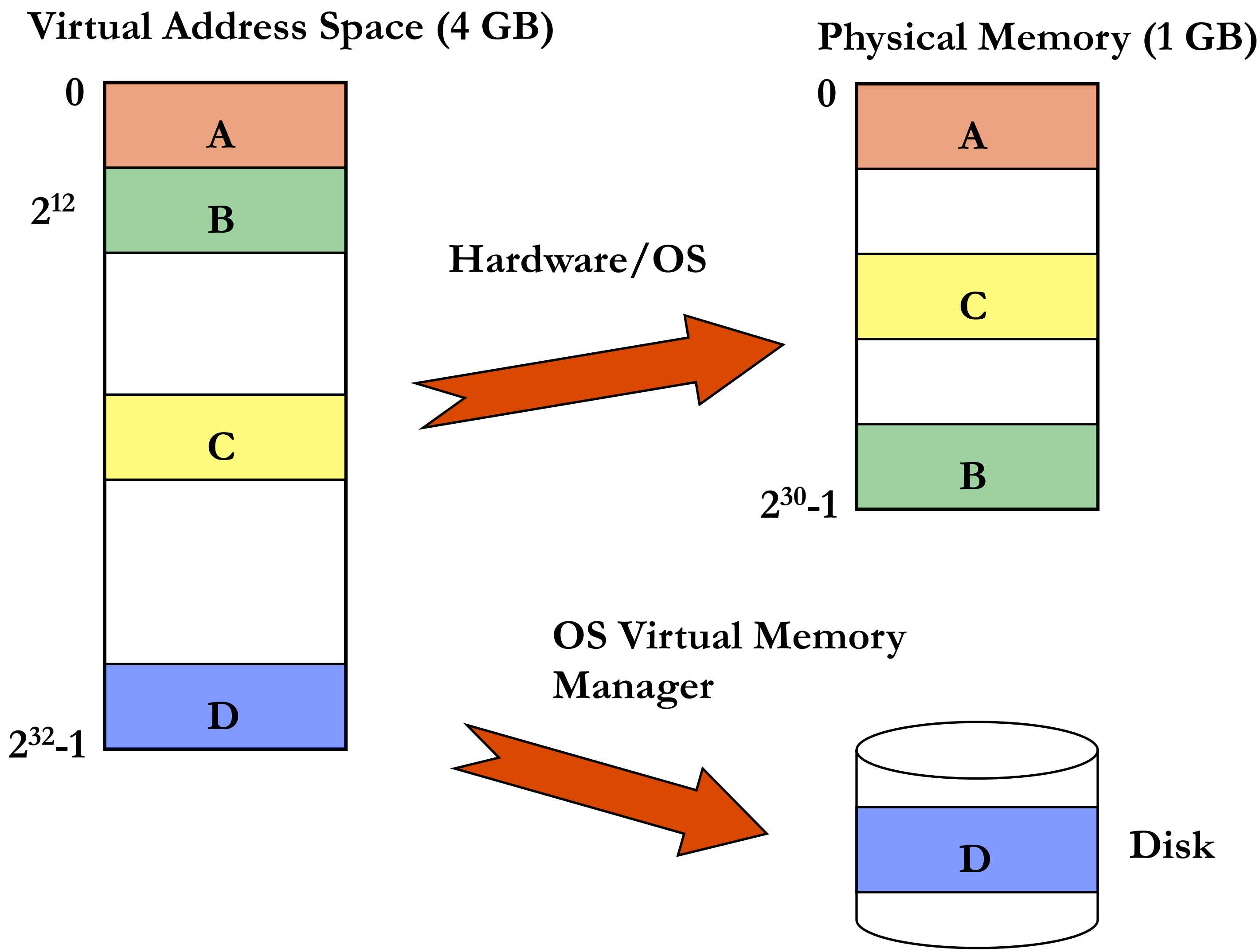
1) Capacity: how do we remove burden of programmers dealing with limited main memory?
   - Want to allow for the physical memory to be smaller than the program's address space (e.g., 32 bits → 4GB)
   - Want to allow multiple programs to share the limited physical memory with no human intervention

2) Safety: how do we allow for safe and efficient sharing of memory among multiple programs?
   - Want to prevent user programs from accessing the memory used by the OS
   - Want strict control of access by each user program to memory of other user programs

Virtual Memory

- Basic idea: each program thinks it owns the entire memory → the **virtual address space**
  - PC and load/store addresses are **virtual addresses**
- Actual main memory: **physical address space**
  - Virtual addresses are **translated** on-the-fly to physical addresses
  - Parts of virtual address space not recently used are stored on disk
- Address translation is done jointly by the OS and hardware

Address translation for 1 program



Physical memory as cache for VM

- Virtual memory space can be larger than physical memory
  - Programmer always sees the full address space (MIPS: 2^{32} bytes)
- Physical memory used as a cache for the virtual memory
  - Physical memory holds the currently used portions of a program's code and data (exploits locality!)
- Secondary storage (disk or flash) "backs" the physical memory
  - OS reserves a portion of the disk for **swap space**
  - OS swaps portions of each process' code and data areas in & out of physical memory on demand (process called **paging**)
  - Swapping is transparent to the programmer

Paging

- A "cache line" or "block" of VM is called a page
  - Plain "page" or "virtual page" for virtual memory
  - "Page frame" or "physical page" for physical memory
- Typical sizes are 4-8 KB (MB or GB in servers)
  - Large enough for efficient disk use and to keep translation tables small
- Mapping is done through a per-program **page table**
  - Allows control of which pages each program can access
  - Different programs can use same virtual addresses

Typical Virtual Memory Parameters

parameter	Cache	Physical Memory
size	1KB – 1MB	128MB - 128GB
block/page	16 - 128 bytes	4KB (up to 4GB)
hit time	2-10 cycles	100 - 200 cycles
miss penalty	8 - 200 cycles	1M - 10M cycles
miss rate	0.1 - 10%	0.00001 - 0.001%

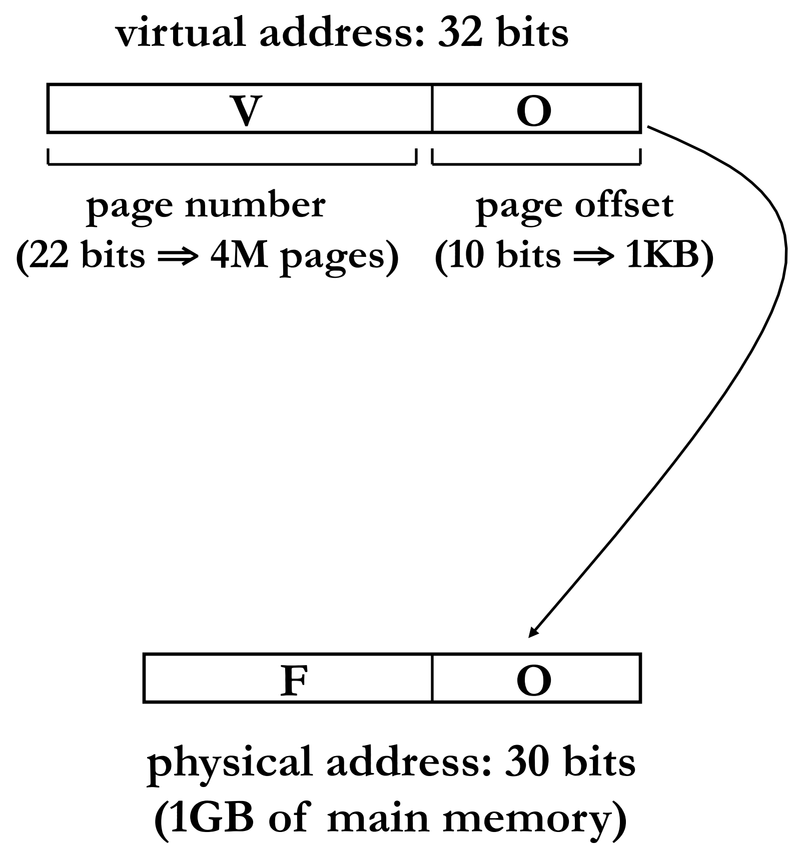
Modified from H&P 5/e Fig. 5.35

- Virtual Memory miss is called a **page fault**

Address Translation

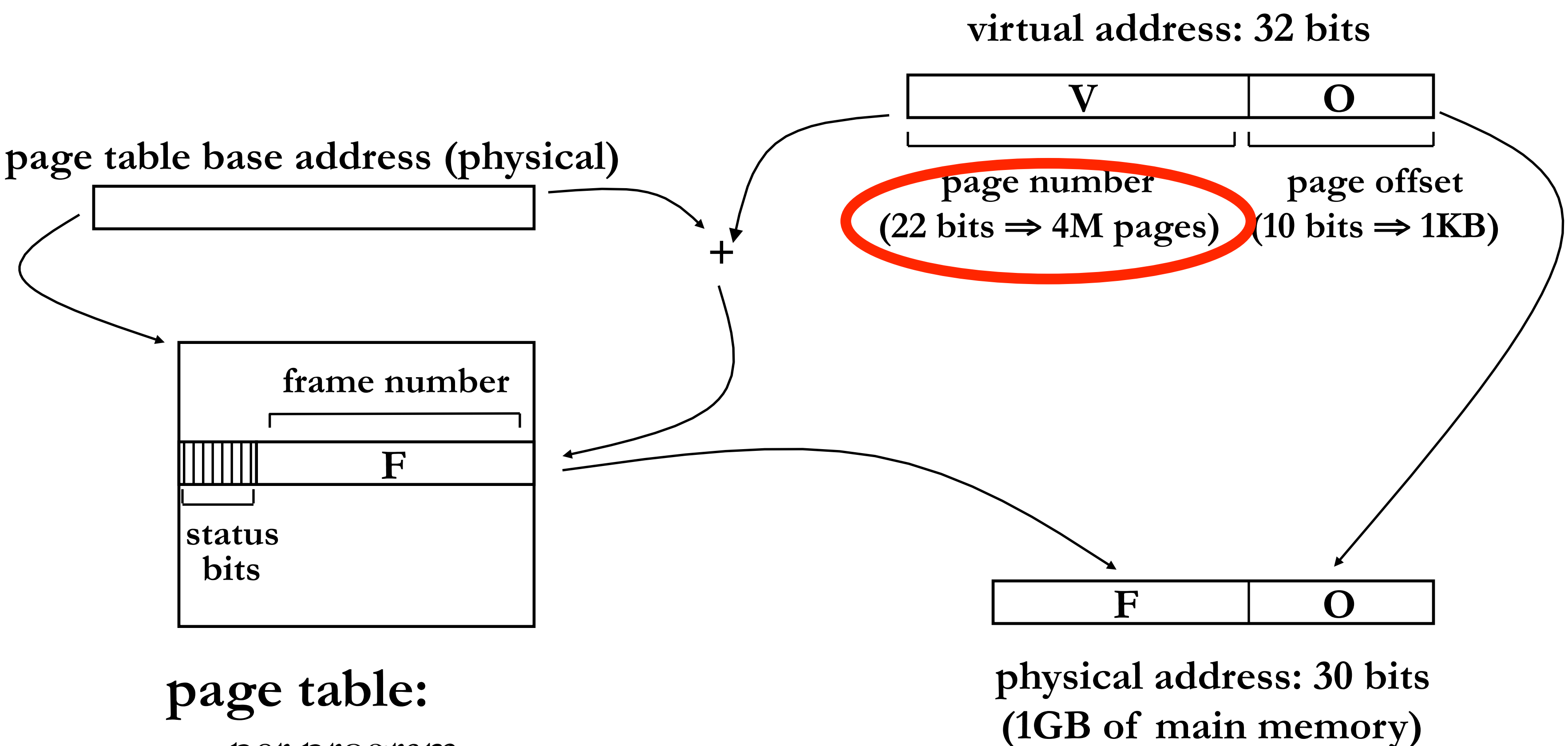
Need:

- A mapping from virtual (V) to physical (F) page numbers
- Page offset not translated
- Must be efficient (in time and space)



Solution: **Page Table!**

Address Translation



page table:

- per program
- one entry per page (e.g. 4M entries)
- located in the system portion of main memory

Practice problem

What is the size of the page table given a 32-bit virtual address space, 4 KB physical pages, and 1 GB of main memory?

Moving pages to/from memory

- Pages are allocated on demand
  - E.g., program launch (results in pages allocated for code, data, and stack); malloc (heap space)
- Pages are replaced and swapped to disk when system runs out of free page frames
  - Aim to replace pages not recently used (principle of locality). **A**(ccess) bit for a page is set whenever page is accessed and is reset periodically
  - If any data in page has been modified, the page must be written back to disk: **M**(odified) bit in status bits is set
- Access to a swapped-out page causes a page-fault which invokes the OS through the interrupt mechanism
  - **R**(esidence) bit in page table status bits is zero

Page replacement

- Least Recently Used (on previous slide)
  – Use past behaviour to predict future
- FIFO – replace in same order as filled
  – Simpler to implement
- Example: page references: 0 2 6 0 7 8
  – Physical memory 4 frames

LRU	FIFO
0	8
8	2
6	6
7	7

Providing Protection

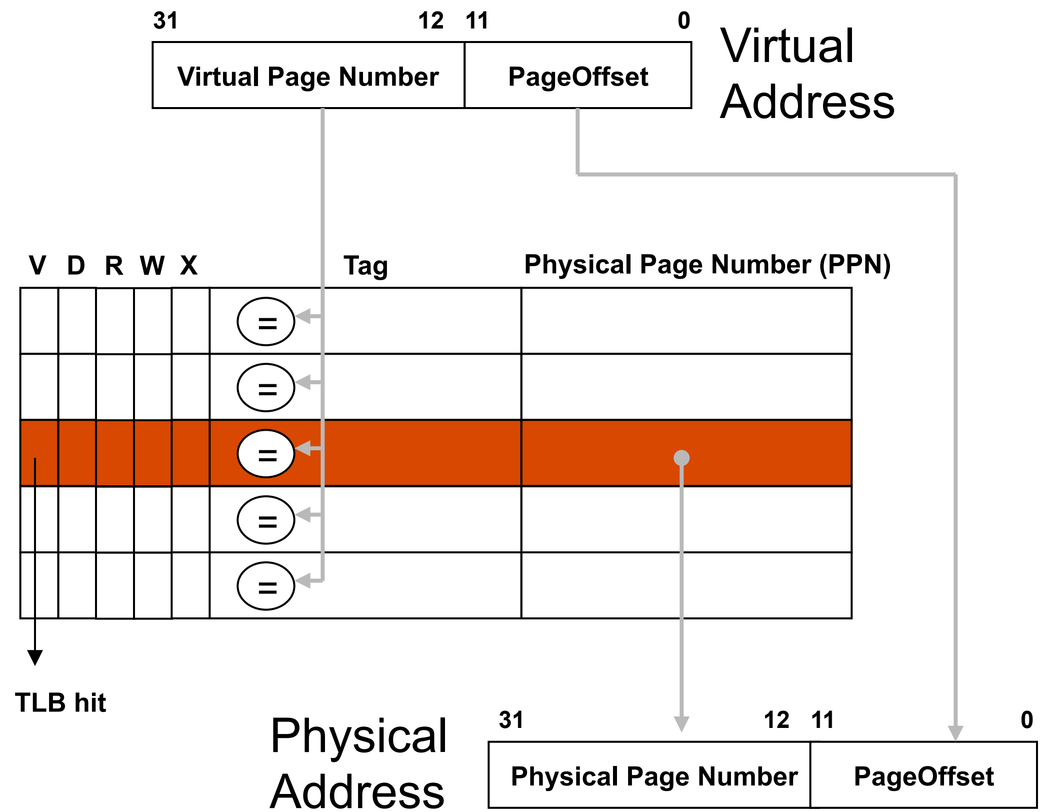
- Each page table entry can have permission bits that control whether
  – the process is allowed to access a page
  – read & write, read-only or execute-only access is allowed
- This enables per-process memory protection
  – E.g. can set up private and shared areas
- Important that only OS can change page tables
  – How? Next lecture!

Fast address translation

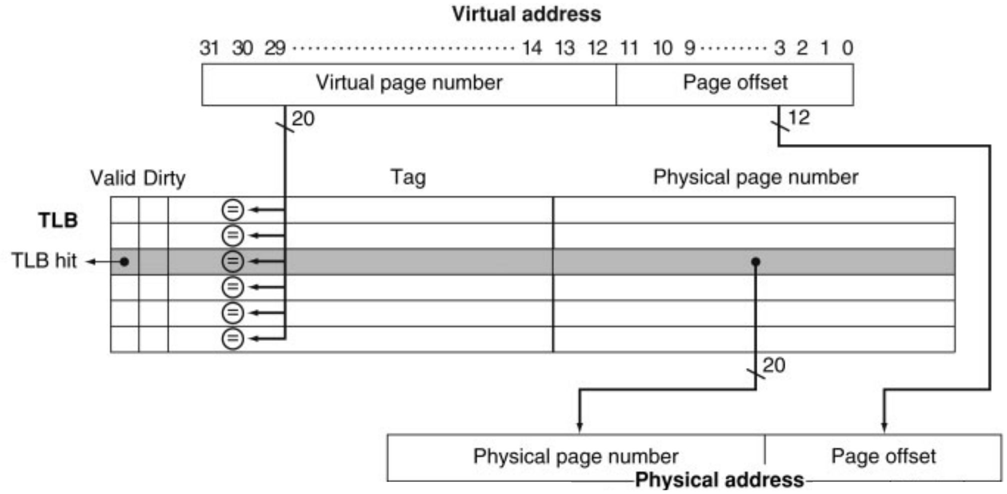
- Problem: page table accesses add latency to each memory access
  – Two memory accesses per load and store (1 to get the page table entry + 1 to get the data)
- Fast address translation: **Translation Lookaside Buffer (TLB)** contained in the MMU
  – Is a cache of page table entries
    - Each TLB entry holds translation information, not program data
    - Tag: virtual address. Entry: physical frame address
  – Small and fast table in hardware, located close to processor
  – Can capture most translations due to principle of locality
  – When page not in TLB: access the page table, and save the translation entry in TLB

Translation Look-aside Buffer (TLB)

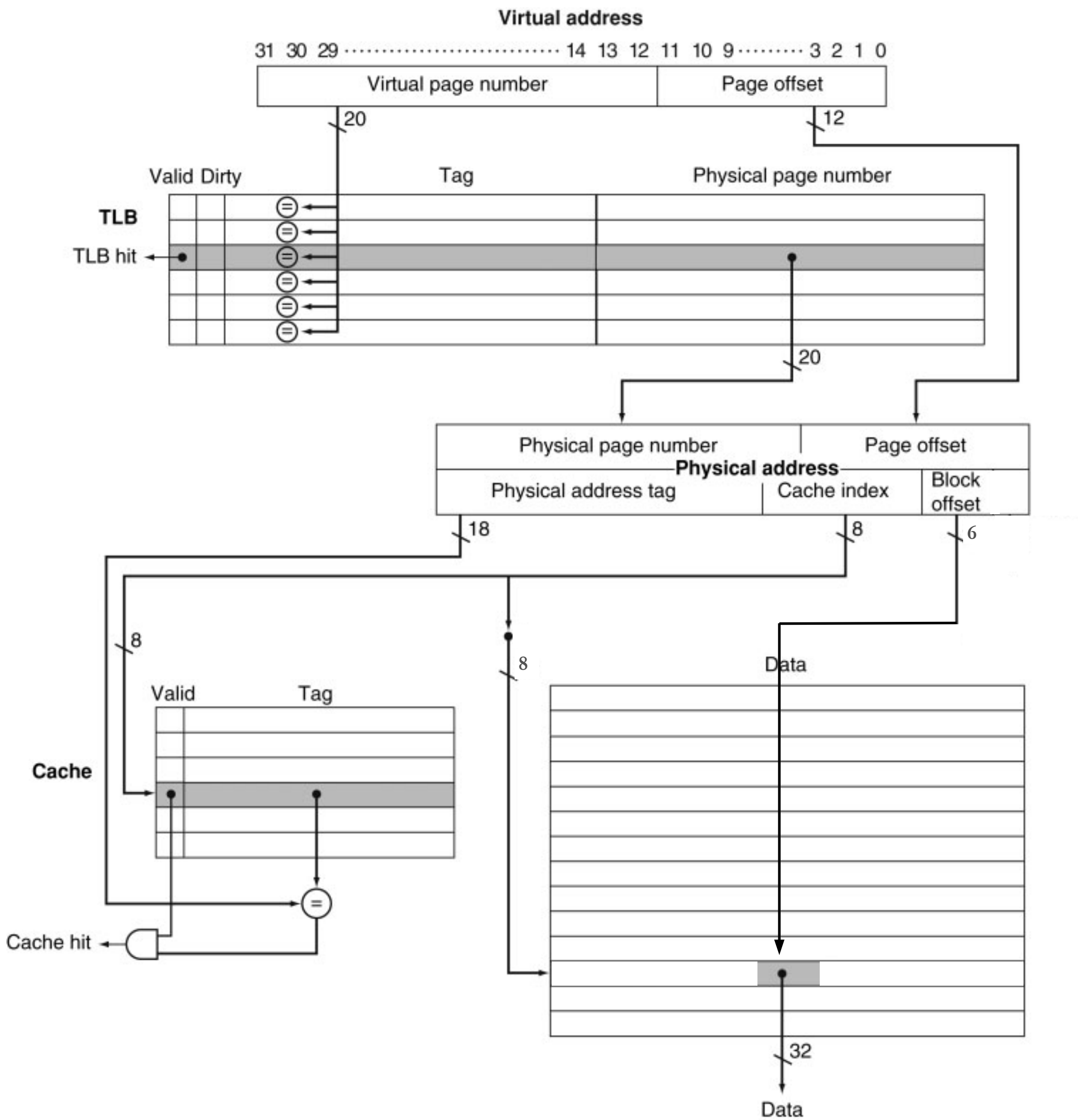
- TLB: a small, fully-associative cache of page table entries
- V (valid) bit indicates a valid entry
- D (dirty) bit indicates whether page has been modified
- R, W, X permission bits
  – Permissions checked on every memory access
- Physical address formed from PPN and Page Offset
- Page table accessed on a TLB miss



Integrating a TLB with the Cache



Integrating a TLB with the Cache



Virtual Memory: full picture