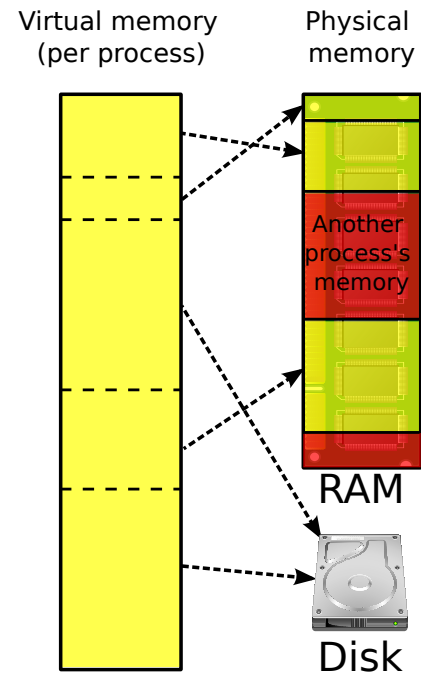


# CS 261 Fall 2021

Mike Lam, Professor



## Virtual Memory and Operating Systems

# Topics

- Operating systems
- Address spaces
- Virtual memory
- Address translation
- Memory allocation

# Lingering questions

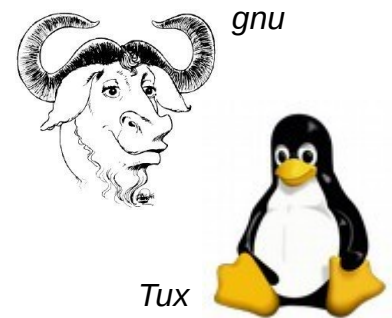
- What happens when you call `malloc()`?
  - How exactly is memory “allocated”?
- What is the correspondence between addresses in machine code and physical memory cells?
  - Are Y86 operand addresses used by the hardware?
- *There's a gap here ...*
  - In early machines, there was no gap; the machine ran one program at a time and every program had complete control of the machine – there was no need for `malloc()`
  - Modern machines support multi-tasking, so this is not sufficient
  - What we need is some *systems software* to mediate between user programs and the hardware

# Operating systems

- An **operating system** (OS) is systems software that provides essential / fundamental system services
  - Manages initialization (**booting**) and cleanup (**shutdown**)
  - Manages hardware/software interactions (**I/O**)
  - Manages running programs (**scheduling**)
  - Manages memory (**virtual memory**)
  - Manages data (**file systems**)
  - Manages external devices (**drivers & interrupts**)
  - Manages communication (**networking**)
  - Manages security (**permissions**)

# Kernel

- The OS **kernel** is the core piece of software that has complete control over the system
  - Direct access to all hardware (“**kernel mode**”)
    - All other software runs in **user mode**
  - Design philosophies: **monolithic** kernels vs. **microkernels**
    - Classic debate: **Tanenbaum vs. Torvalds**
  - Often designed to be small but extensible
    - Plugins are called **drivers**
  - Technically, “**Linux**” is a kernel
    - Some call the operating system “**GNU/Linux**”
    - Combination of **Linux** kernel and **GNU** userspace utilities

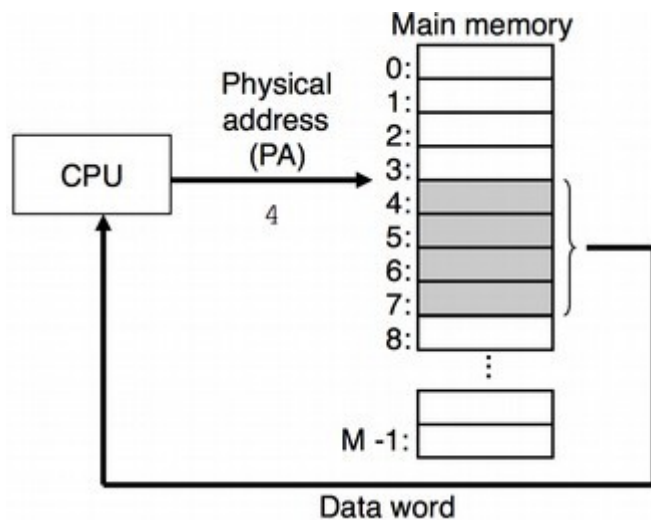


# OS abstractions

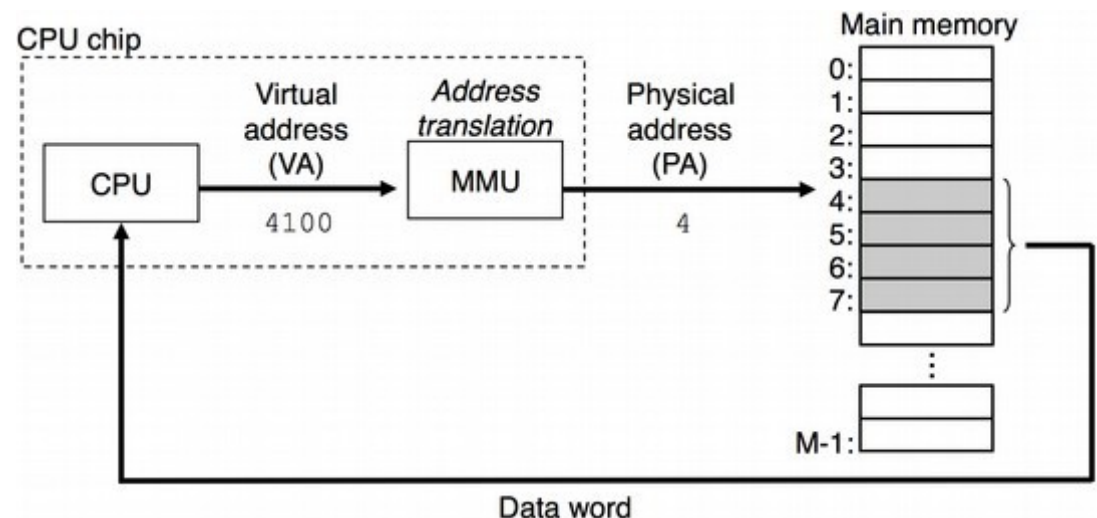
- The OS provides many useful abstractions so that programs don't need to handle hardware details
  - CS 450 covers operating systems in detail
- In this class:
  - **Virtual memory**: logical view of memory hierarchy
  - **Process**: logical view of a program running on a CPU
  - **File**: logical view of data on a disk
  - **Thread**: *logical flow of execution in a program*

# Virtual memory

- Kernel translates between **virtual** and **physical** addresses
- Goals:
  - Use main memory as a cache for disks
  - Provide every process with a uniform view of memory
  - Protect processes from interference



No virtual memory



With virtual memory

# Address spaces

- An **address space** is an ordered set of non-negative integer addresses
  - Ex: { 0, 1, 2, 3, ... , 499, 500 }
  - **Linear** address spaces don't skip any addresses
  - Two address spaces: **virtual** and **physical**
  - Every byte has two addresses (virtual and physical)

**Example:** Y86 programs have a virtual address space with addresses that range from 0x0 to 0x1000, which is large enough to store 4K bytes

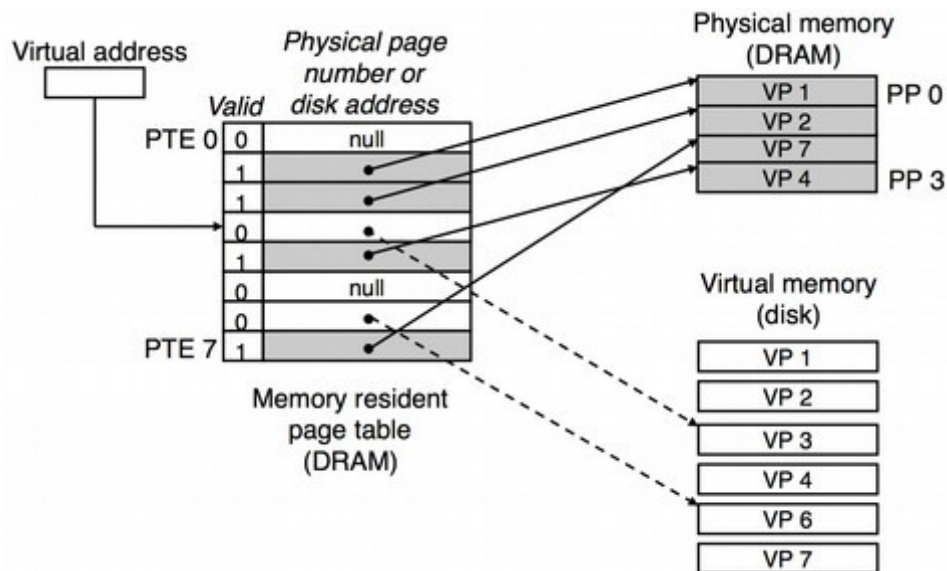


# Virtual memory

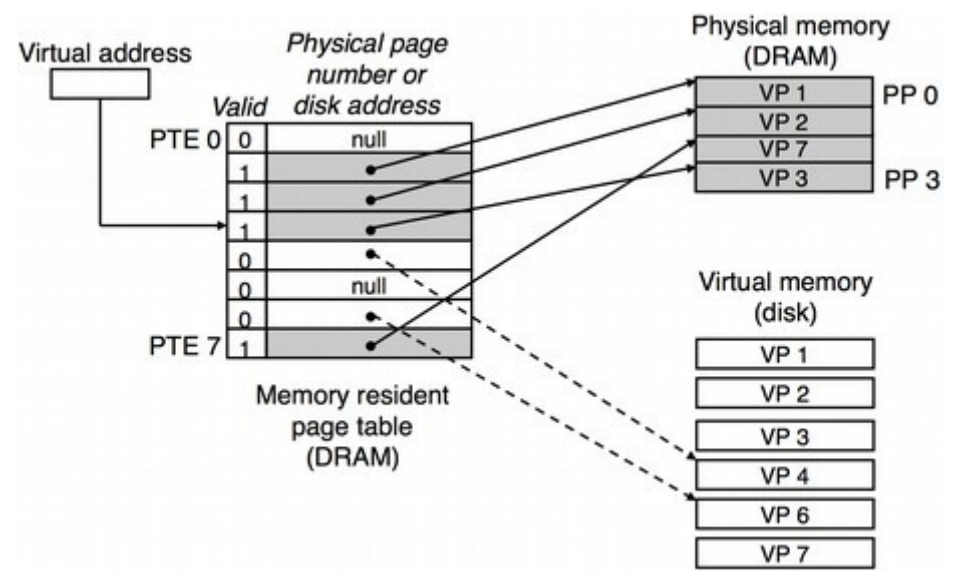
- Fixed-sized memory partitioning
  - Virtual address space into **virtual pages**
  - Physical address space into **physical pages** (or **frames**)
  - Pages are usually relatively large (4 KB to 2 MB)
- Virtual memory uses RAM as a cache for pages
  - Process uses consistent virtual / logical addresses
  - OS translates these to physical addresses as necessary
    - Use a table for fast lookups!
  - We will assume hardware handles L1, L2, & L3 SRAM caches

# Page tables

- **Page table**: OS data structure for page lookups (array of page table entries)
- DRAM cache misses (called **page faults**) are very expensive
  - Disks are MUCH slower than DRAM
  - Transferring pages back and forth is called **paging** or **swapping**



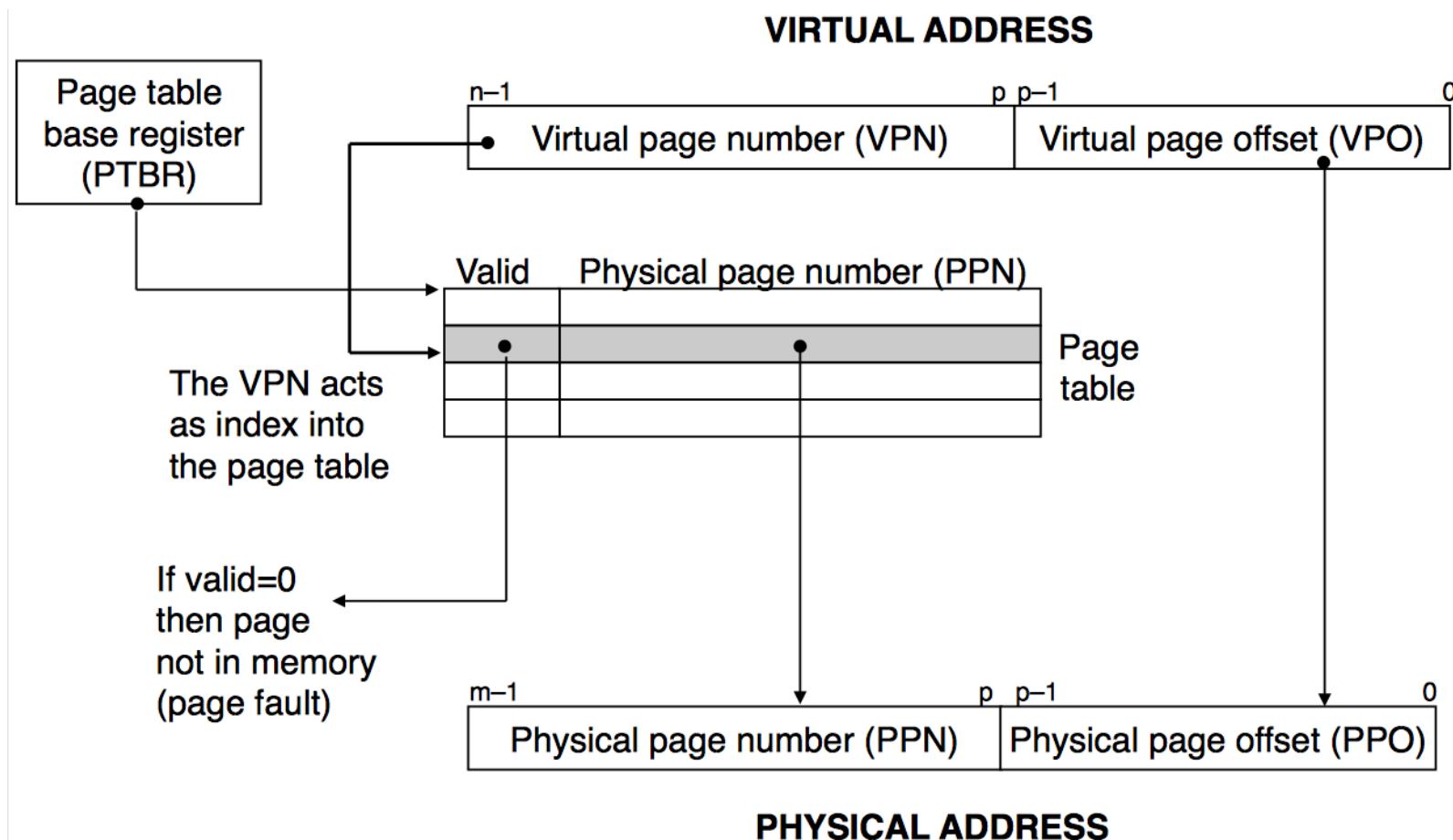
before page fault on VP 3



after page fault on VP 3

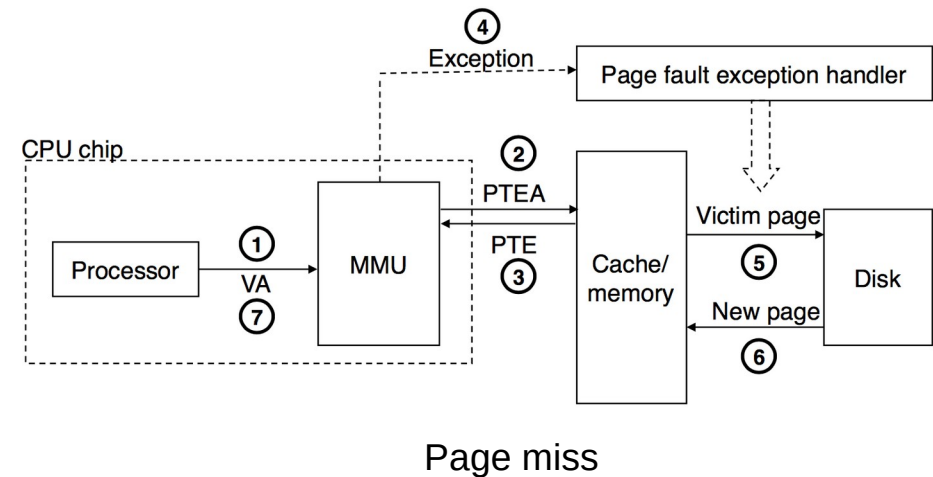
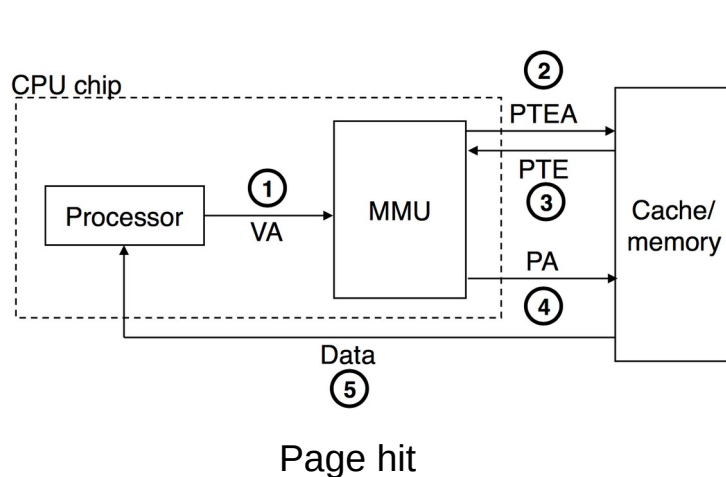
# Address translation

- n-bit virtual address space  $\Rightarrow$  m-bit physical address space
- p-bit page offsets (page size is  $2^p$ )



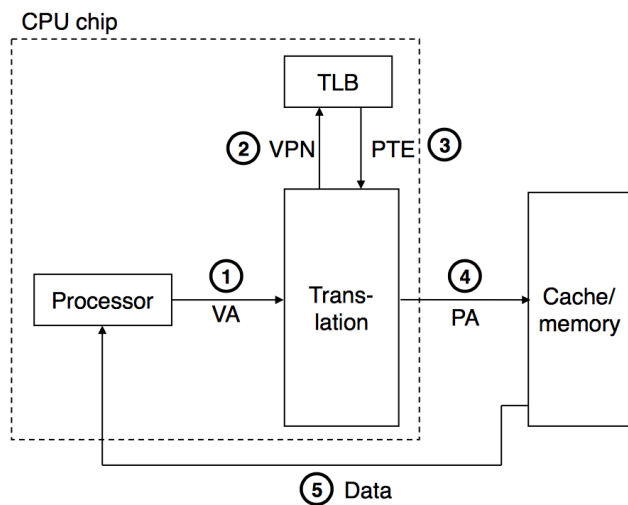
# Address translation

- **Memory management unit (MMU)**
  - On-chip CPU component for address translation
  - Goal: perform translation as quickly as possible

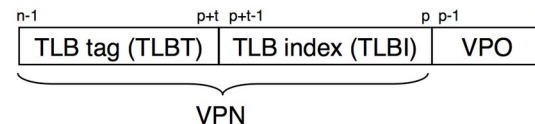


# Address translation

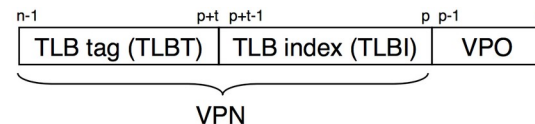
- **Translation lookaside buffer (TLB)**
  - Small cache of **page table entries** (PTEs) in MMUs
  - Provides faster address translations (in most cases)
  - *It's caches all the way down ...*



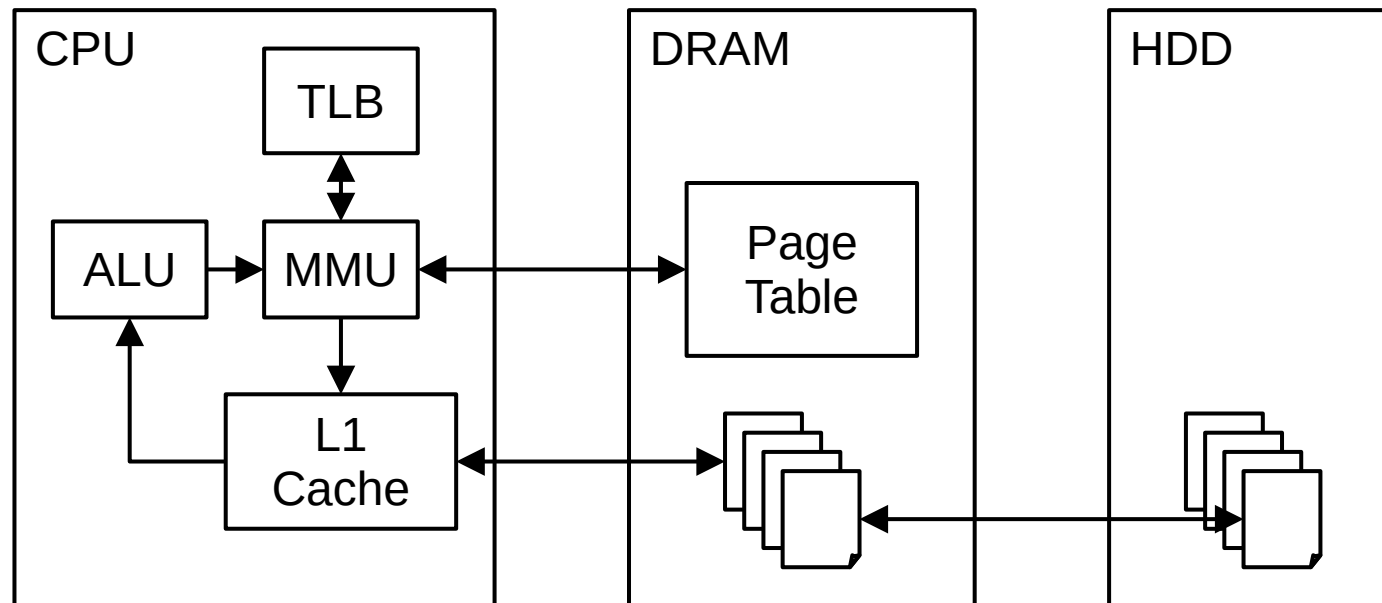
Page hit



Page miss

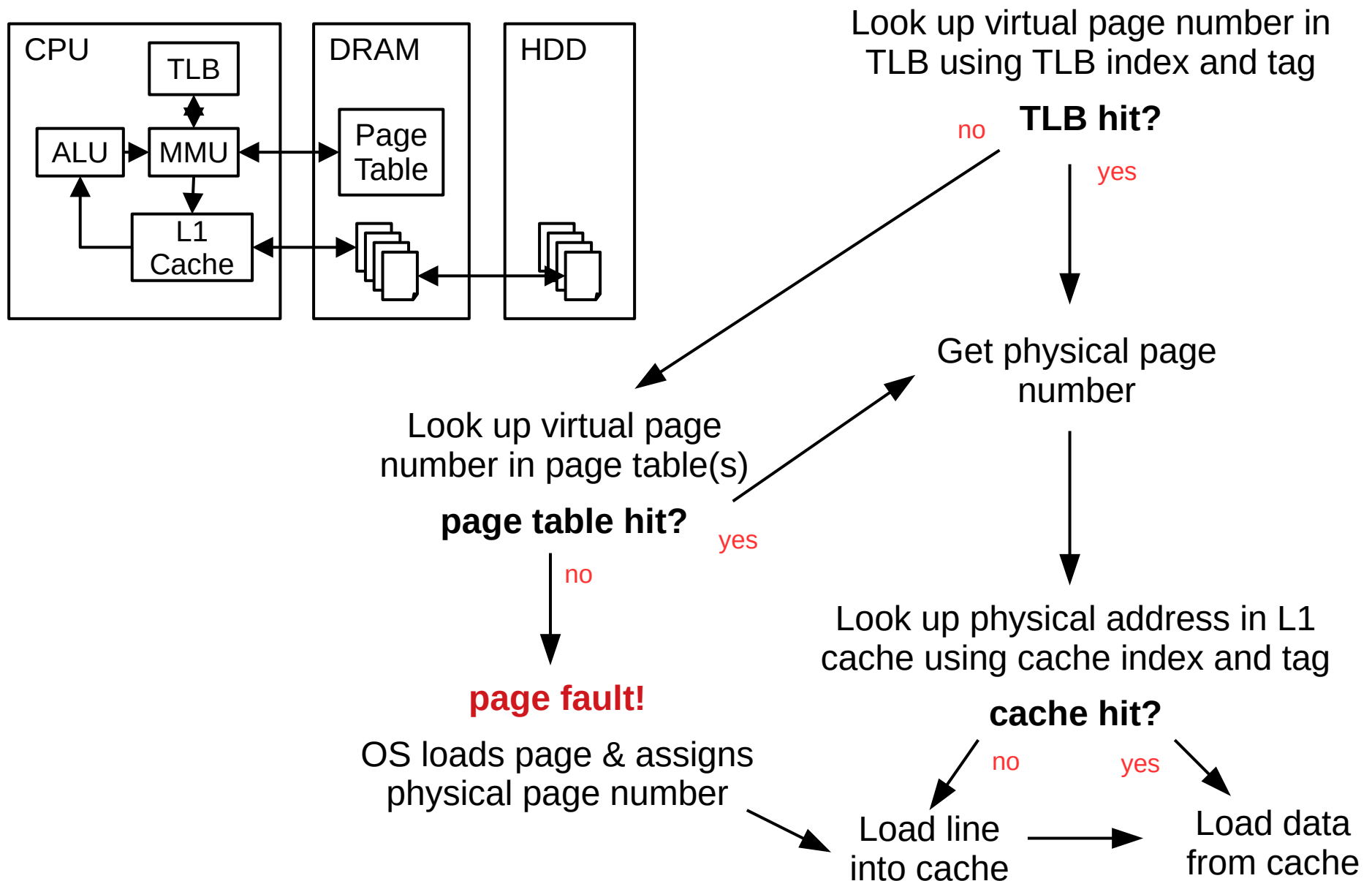


# Address translation w/ L1 cache



- CPU contains ALU, L1 cache, MMU, and TLB
  - L1 cache contains data pages
- DRAM contains page table and data pages
- HDD contains data pages

# Address translation w/ L1 cache



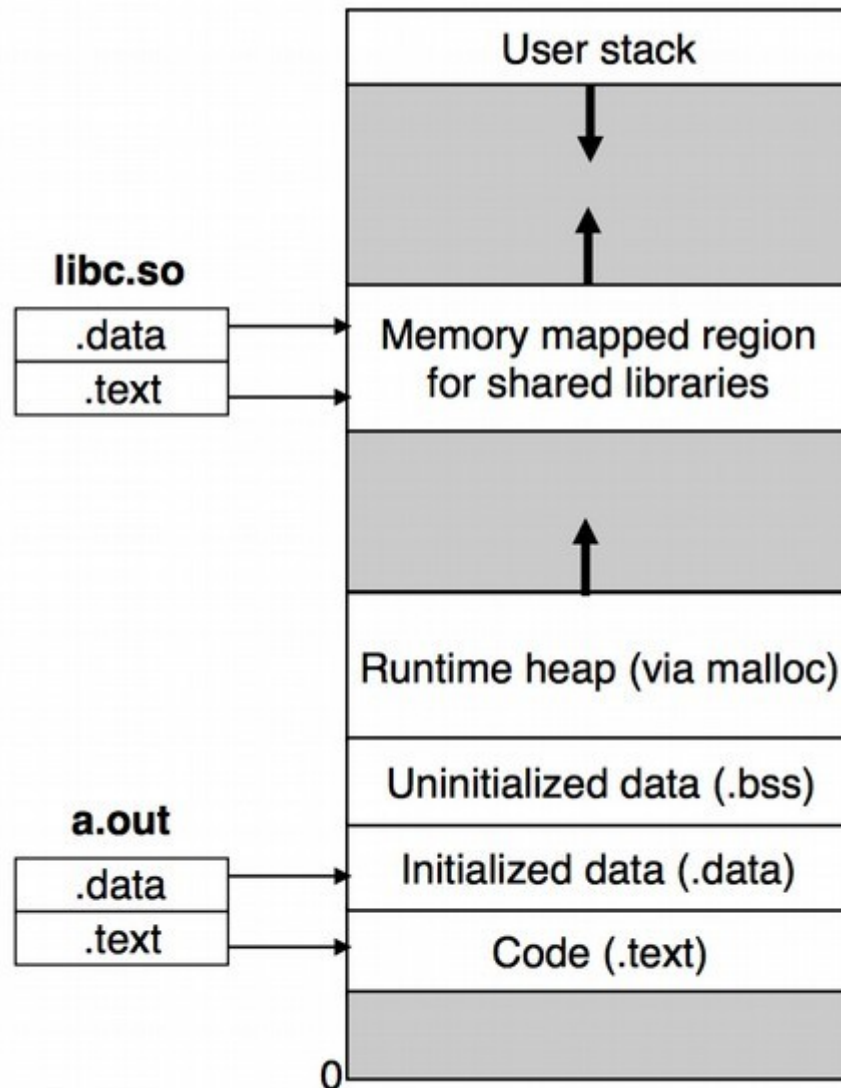
# Memory management

- Operating system provides memory allocation service
  - `mmap` system call (`malloc` uses this)
  - Creates virtual memory allocation
  - **Private** regions: changes are only seen by owner
    - Private, variable-sized region called the **heap**
  - **Shared** regions: changes are seen by all processes
    - Usually between heap and stack
    - Multiple virtual addresses map to the same physical address
    - Changes are seen by all processes
    - Usually a read-only region for shared library code



# Process address spaces

Kernel uses higher addresses



Typical Linux process address space

# Process address spaces

- OSes maintain a separate page table for every process
  - Provides program **linking consistency**
    - E.g., code always begins at 0x400000
  - Simplifies **efficient loading**
    - Don't actually load data from disk until needed (more efficient than P2!)
  - Streamlines **library sharing**
    - Keep one physical copy with multiple virtual mappings
  - Simplifies **memory allocation**
    - `malloc()` doesn't need to find contiguous physical memory
  - Improves **security**
    - Processes can't see/edit each others' address spaces

# Virtual memory caveats

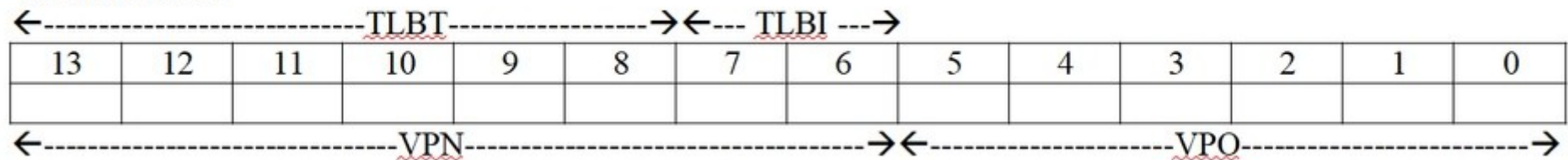
- Virtual memory works well **if a program has good locality**
  - Especially temporal locality
  - This is a compelling reason to design for good locality
- Virtual memory works well **if a program has a working set that fits in main memory**
  - If this is not true, the system may need to continuously swap pages in and out
  - This is called **thrashing**, and is a significant cause of poor program performance
  - Can be detected by **profilers** (via counting page faults)

# Our final module

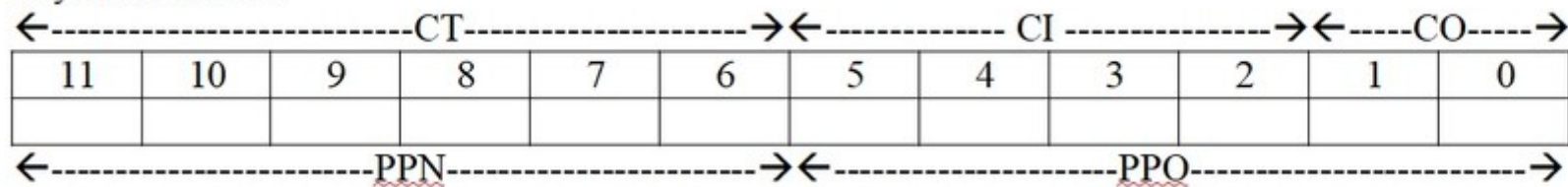
- For the rest of the semester, we will continue discussing **operating systems** principles
  - Layers of abstraction that simplify development
  - Theme: *systems software is a foundation*
  - If you like this material, plan on taking **CS 450**

# Virtual address translation

Virtual Address



Physical Address

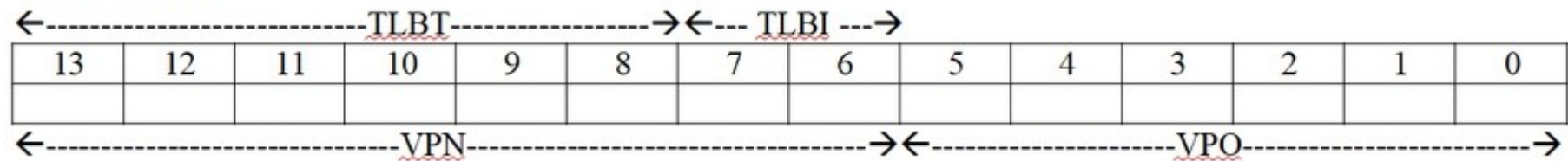


Section of Page Tables in Main Memory

<u>VPN</u>	<u>PPN</u>	Valid		<u>VPN</u>	<u>PPN</u>	Valid
00	28	1		08	13	1
01	---	0		09	17	1
02	33	1		0A	09	1
03	02	1		0B	---	0
04	---	0		0C	---	0
05	16	1		0D	2D	1
06	---	0		0E	11	1
07	---	0		0F	0D	1

# TLB cache

Virtual Address

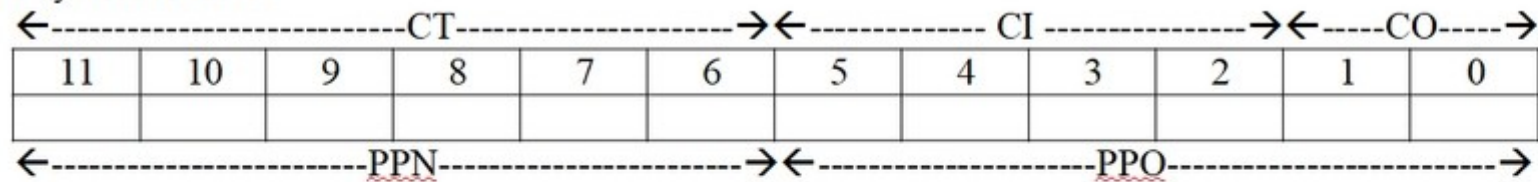


TLB (Translation Lookaside Buffer)

Set	Tag	<u>PPN</u>	Valid	Tag	<u>PPN</u>	Valid	Tag	<u>PPN</u>	Valid	Tag	<u>PPN</u>	Valid
0	03	---	0	09	0D	1	00	---	0	07	02	1
1	03	2D	1	02	---	0	04	---	0	0A	---	0
2	02	---	0	08	---	0	06	---	0	03	---	0
3	07	---	0	03	0D	1	0A	34	1	02	---	0

# L1 data cache

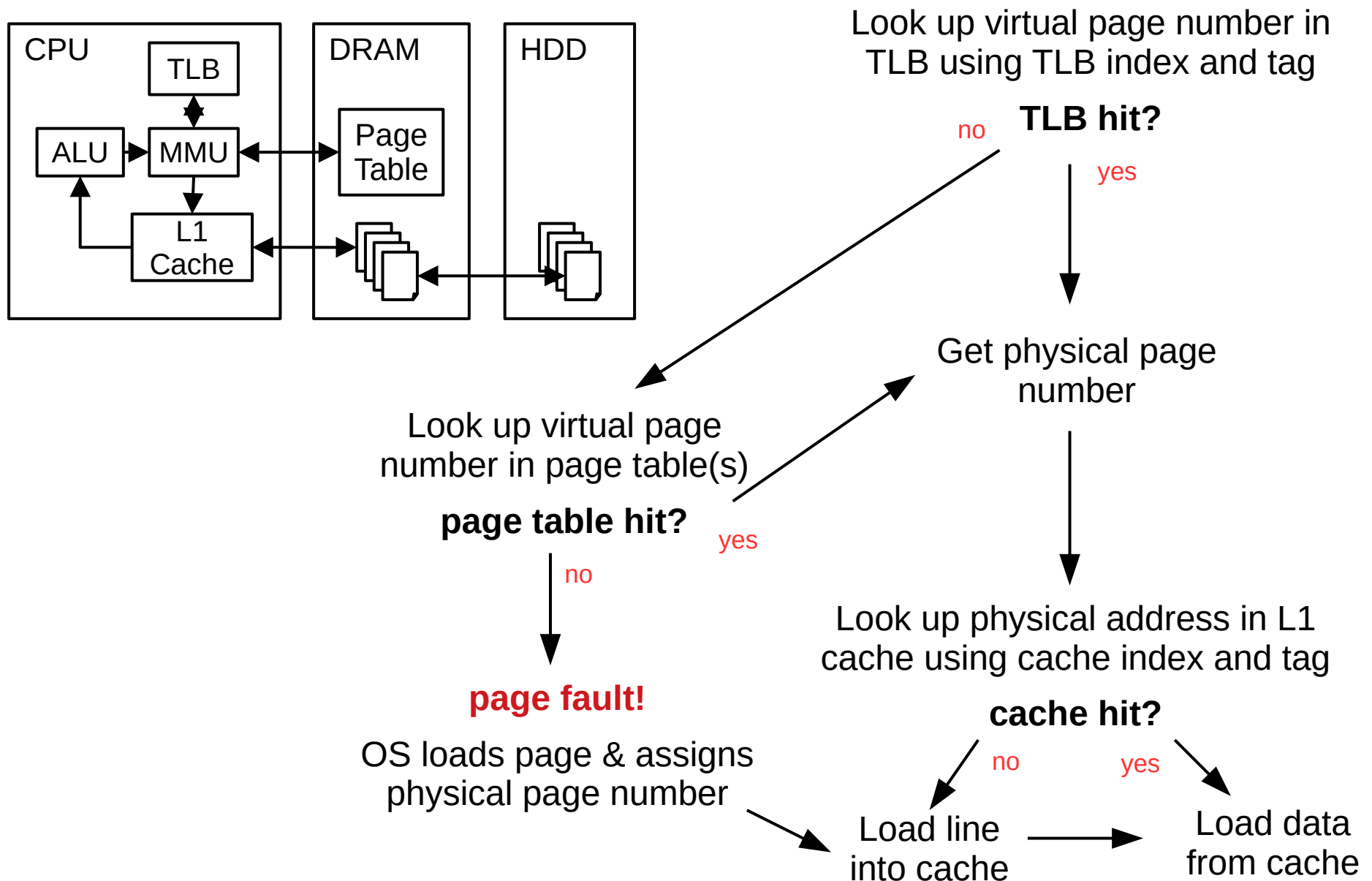
Physical Address



Direct-mapped L1 Data Cache

Index	Tag	Valid	<u>BLK 0</u>	<u>BLK 1</u>	<u>BLK 2</u>	<u>BLK 3</u>
0	19	1	99	11	23	11
1	15	0	---	---	---	---
2	1B	1	00	02	04	08
3	36	0	---	---	---	---
4	32	1	43	6D	8F	09
5	0D	1	38	72	F0	1D
6	31	0	---	---	---	---
7	16	1	11	C2	<u>DF</u>	03
8	24	1	3A	00	51	89
9	2D	0	---	---	---	---
A	2D	1	93	15	DA	3B
B	0B	0	---	---	---	---
C	12	0	---	---	---	---
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	---	---	---	---

# Address translation w/ L1 cache





# Address translation w/ L1 cache

- 1) Convert hex virtual address to binary representation
  - Fill in virtual address bits from RIGHT TO LEFT (extra is zeros on left)
- 2) Extract page number (VPN) and page offset (VPO) from virtual address
- 3) Extract TLB index and TLB tag from virtual address
- 4) In TLB, look up TLB index and tag to find PPN
  - If not valid: TLB miss!
- 5) If not in TLB look up VPN in page table to find PPN
  - If not in page table: page fault!
- 6) Assemble physical address from page number (PPN) and page offset (PPO)
  - Physical page offset (PPO) is the same as the virtual page offset (VPO)
- 7) Extract cache index and cache tag from physical address
- 8) In cache, look up cache index and tag
  - If not found, cache miss!
  - If found, return data