# Lecture 25: Security, VM, Multiproc

- Today's topics:
  - Security wrap-up
  - Virtual memory
  - Multiprocessors, cache coherence

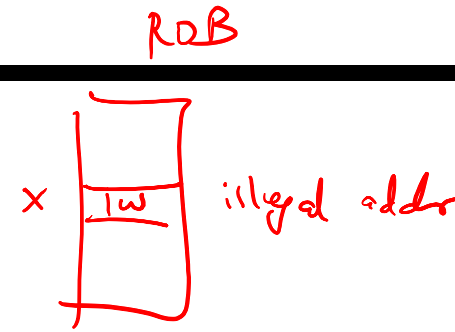
HW 10 posted later today

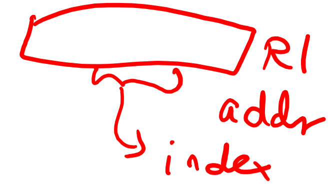
Final exam
85 - 15
post midterm → pre midterm

2018

# Meltdown

Defense: On illegal access, stifle speculation




1. HW design error/bug
2. An attacker prog running by itself & reading all of mem
3. step 1 Prime – Attacker fills cache with its own data
   a[0] → a[4095]
4. step 2
   lw R1 ← illegal addr
   lw ← [R1] → left a footprint in the cache
   Bring in a block & place it in cache. Set where the block is placed is f(R1)
5. Proc recovers & squashes ROB state (but cache is not cleaned up)
6. step 3 Probe the cache – Rd a[0], a[1], ... a[82] a[4095]
   latency 2 2 200 cyc 2

# Spectre: Variant 1

① Not a hw bug
② Progs are naturally leaky
③ Attacker runs alongside & examines the secret-dependent footprints in the system

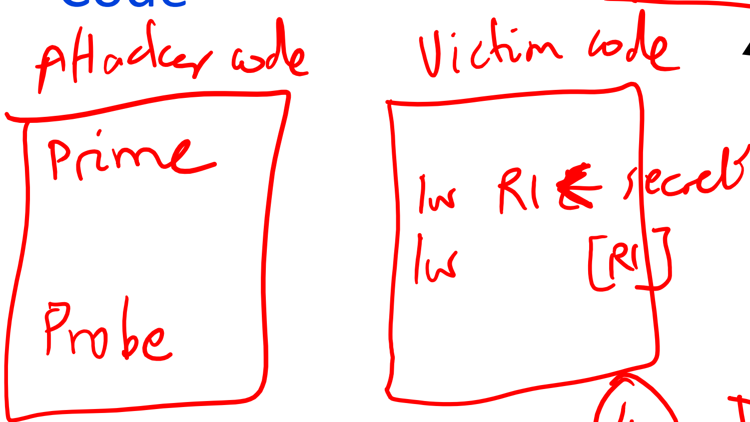
x is controlled by attacker

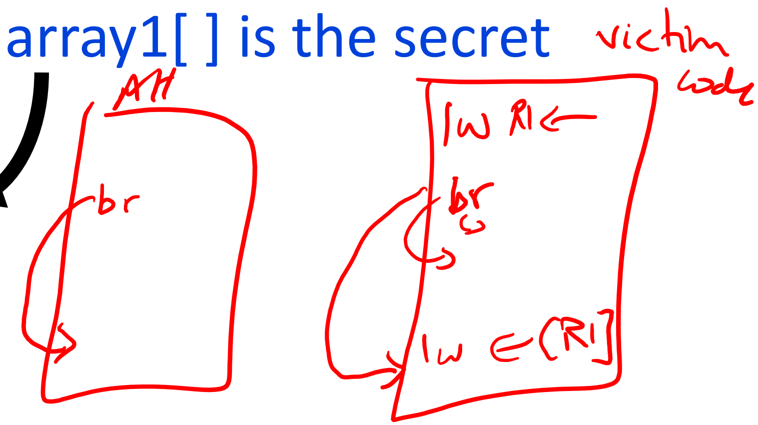
Thanks to bpred, x can be anything

array1[ ] is the secret

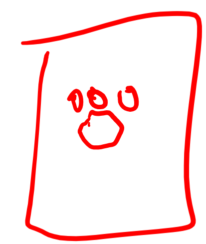
Victim Code

```
if (x < array1_size)
    y = array2[ array1[x] ];
```




Access pattern of array2[ ] betrays the secret



④ Defense: Isolate progs + resources

# Spectre: Variant 2

Attacker code

Label0: if (1)

Label1: ...

Victim code

R1 ← (from attacker)

R2 ← some secret

Label0: if (…)

… …

Victim code

Label1:

lw [R2]

# Virtual Memory

- Processes deal with virtual memory – they have the illusion that a very large address space is available to them

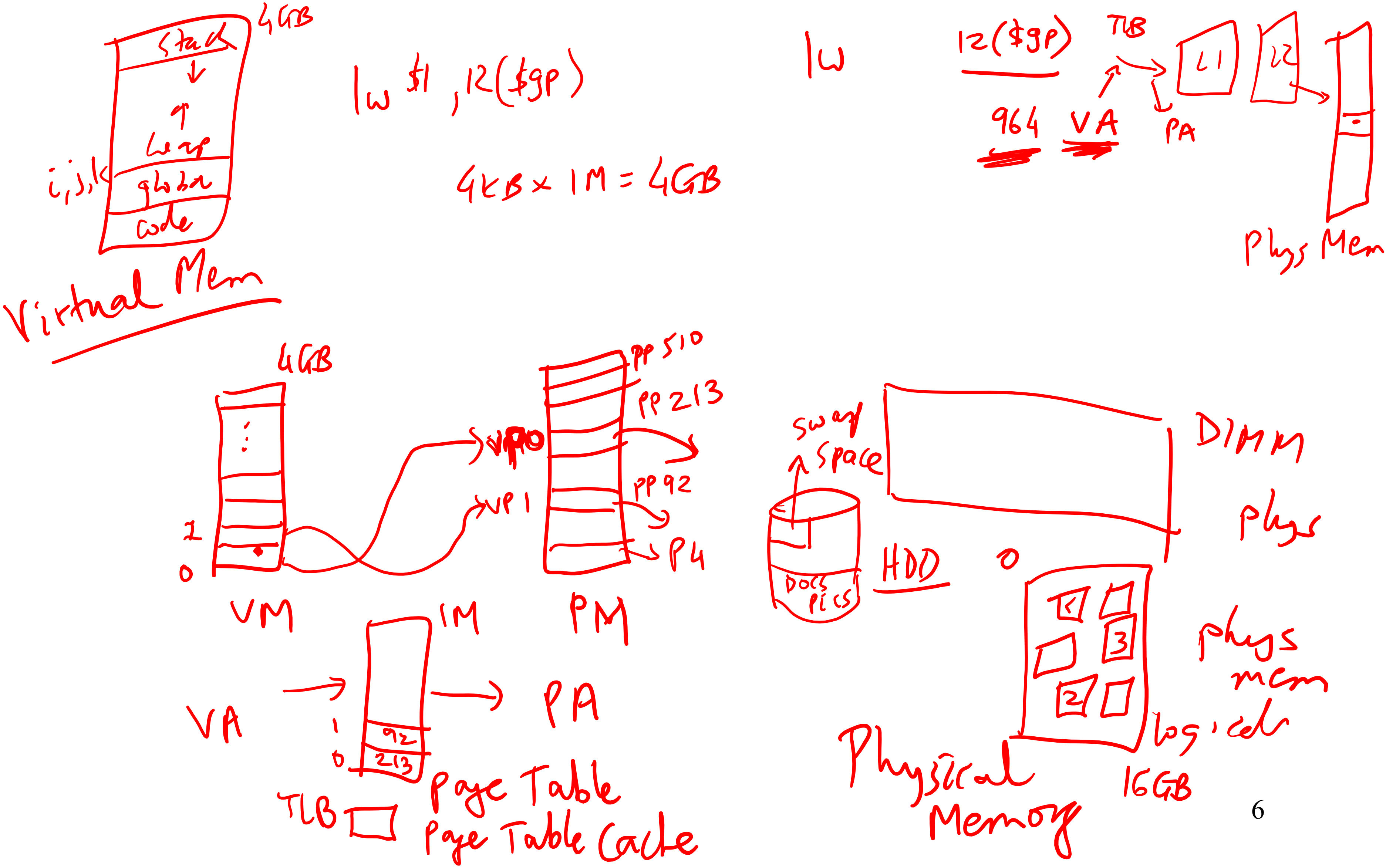
- There is only a limited amount of physical memory that is shared by all processes – a process places part of its virtual memory in this physical memory and the rest is stored on disk (called swap space)

- Thanks to locality, disk access is likely to be uncommon

- The hardware ensures that one process cannot access the memory of a different process

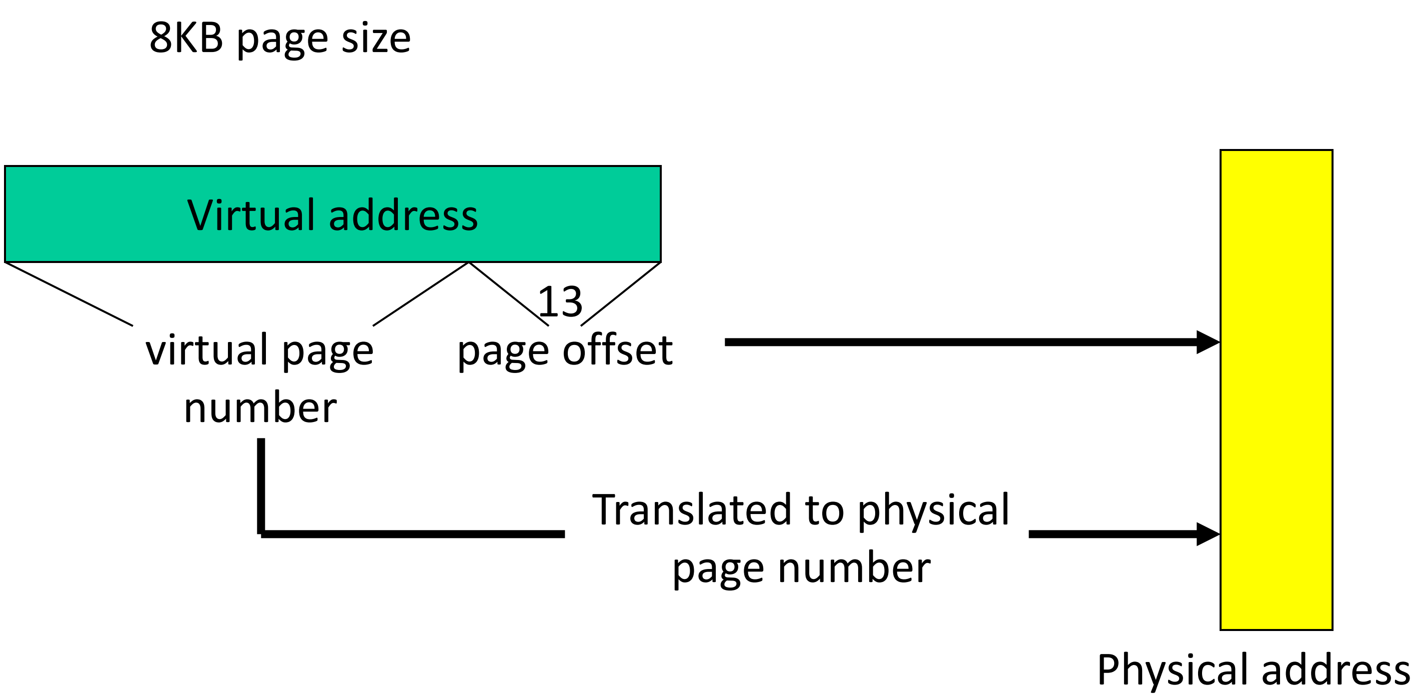
# Virtual Memory

1 page = 4KB → 1MB → 1GB



# Address Translation

- The virtual and physical memory are broken up into pages



# Memory Hierarchy Properties

- A virtual memory page can be placed anywhere in physical memory (fully-associative)
- Replacement is usually LRU (since the miss penalty is huge, we can invest some effort to minimize misses)
- A page table (indexed by virtual page number) is used for translating virtual to physical page number
- The page table is itself in memory

# TLB

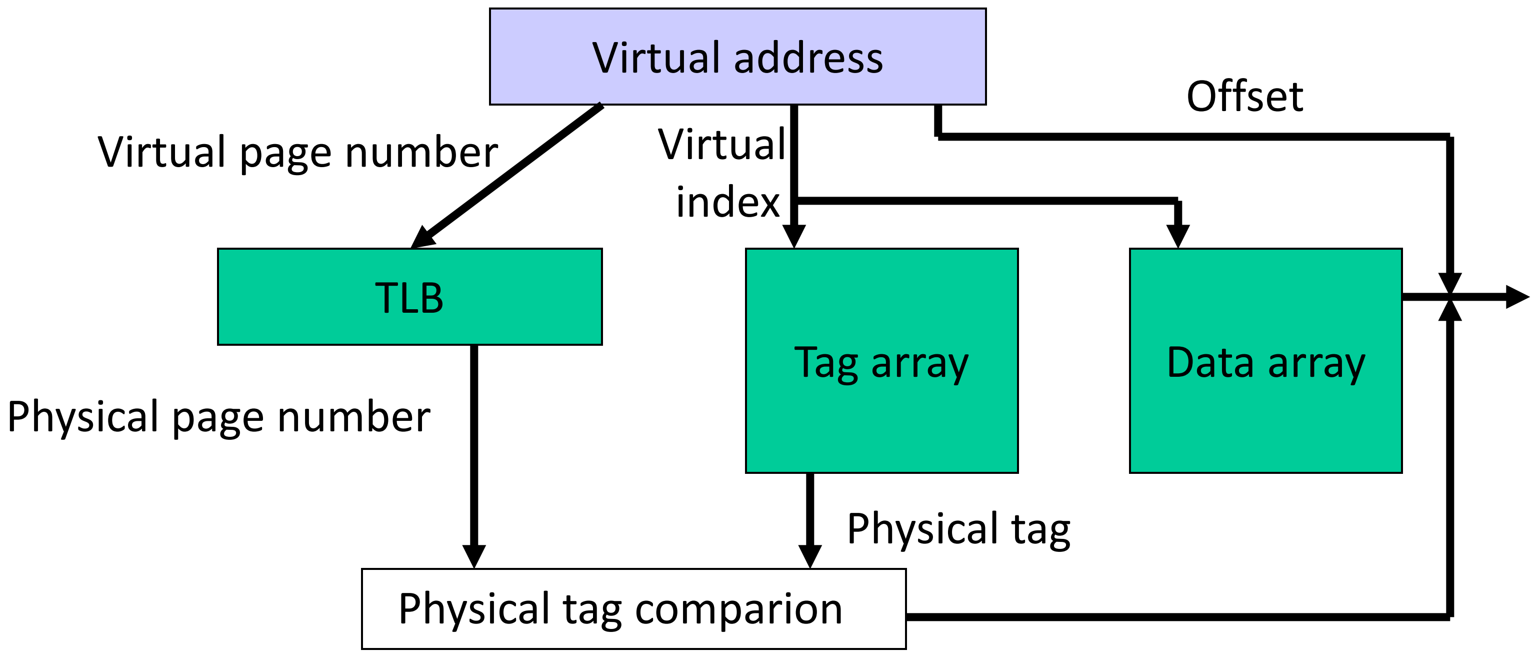
- Since the number of pages is very high, the page table capacity is too large to fit on chip
- A translation lookaside buffer (TLB) caches the virtual to physical page number translation for recent accesses
- A TLB miss requires us to access the page table, which may not even be found in the cache – two expensive memory look-ups to access one word of data!
- A large page size can increase the coverage of the TLB and reduce the capacity of the page table, but also increases memory waste

# TLB and Cache (weeds)

- Is the cache indexed with virtual or physical address?
  - To index with a physical address, we will have to first look up the TLB, then the cache → longer access time
  - Multiple virtual addresses can map to the same physical address – must ensure that these different virtual addresses will map to the same location in cache – else, there will be two different copies of the same physical memory word

- Does the tag array store virtual or physical addresses?
  - Since multiple virtual addresses can map to the same physical address, a virtual tag comparison can flag a miss even if the correct physical memory word is present

# Cache and TLB Pipeline




Virtually Indexed; Physically Tagged Cache

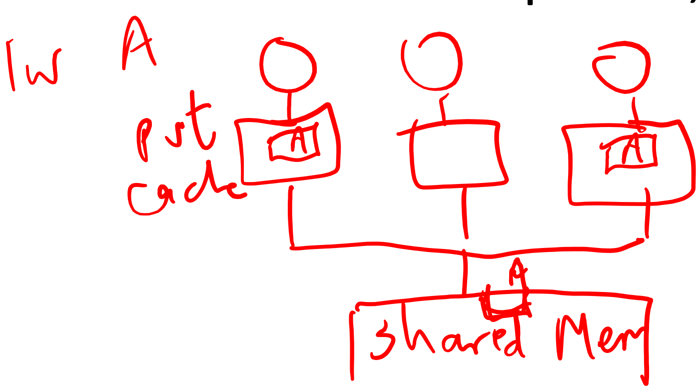
# Bad Events

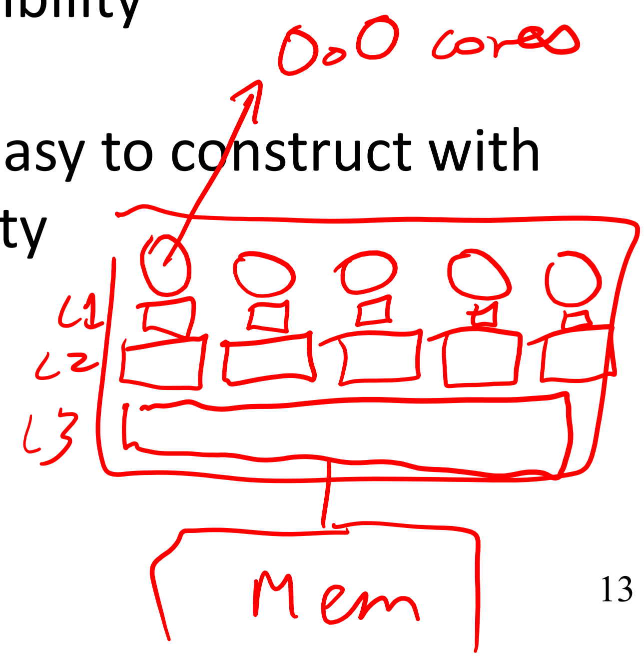
- Consider the longest latency possible for a load instruction:
  - TLB miss: must look up page table to find translation for v.page P
  - Calculate the virtual memory address for the page table entry that has the translation for page P – let's say, this is v.page Q
  - TLB miss for v.page Q: will require navigation of a hierarchical page table (let's ignore this case for now and assume we have succeeded in finding the physical memory location (R) for page Q)
  - Access memory location R (find this either in L1, L2, or memory)
  - We now have the translation for v.page P – put this into the TLB
  - We now have a TLB hit and know the physical page number – this allows us to do tag comparison and check the L1 cache for a hit
  - If there's a miss in L1, check L2 – if that misses, check in memory
  - At any point, if the page table entry claims that the page is on disk, flag a page fault – the OS then copies the page from disk to memory and the hardware resumes what it was doing before the page fault … phew!

# Multiprocessor Taxonomy

Multi-cores

- SISD: single instruction and single data stream: uniprocessor
- MISD: no commercial multiprocessor: imagine data going through a pipeline of execution engines
- SIMD: vector architectures: lower flexibility
- MIMD: most multiprocessors today: easy to construct with off-the-shelf computers, most flexibility



# Memory Organization - I

- Centralized shared-memory multiprocessor or Symmetric shared-memory multiprocessor (SMP)

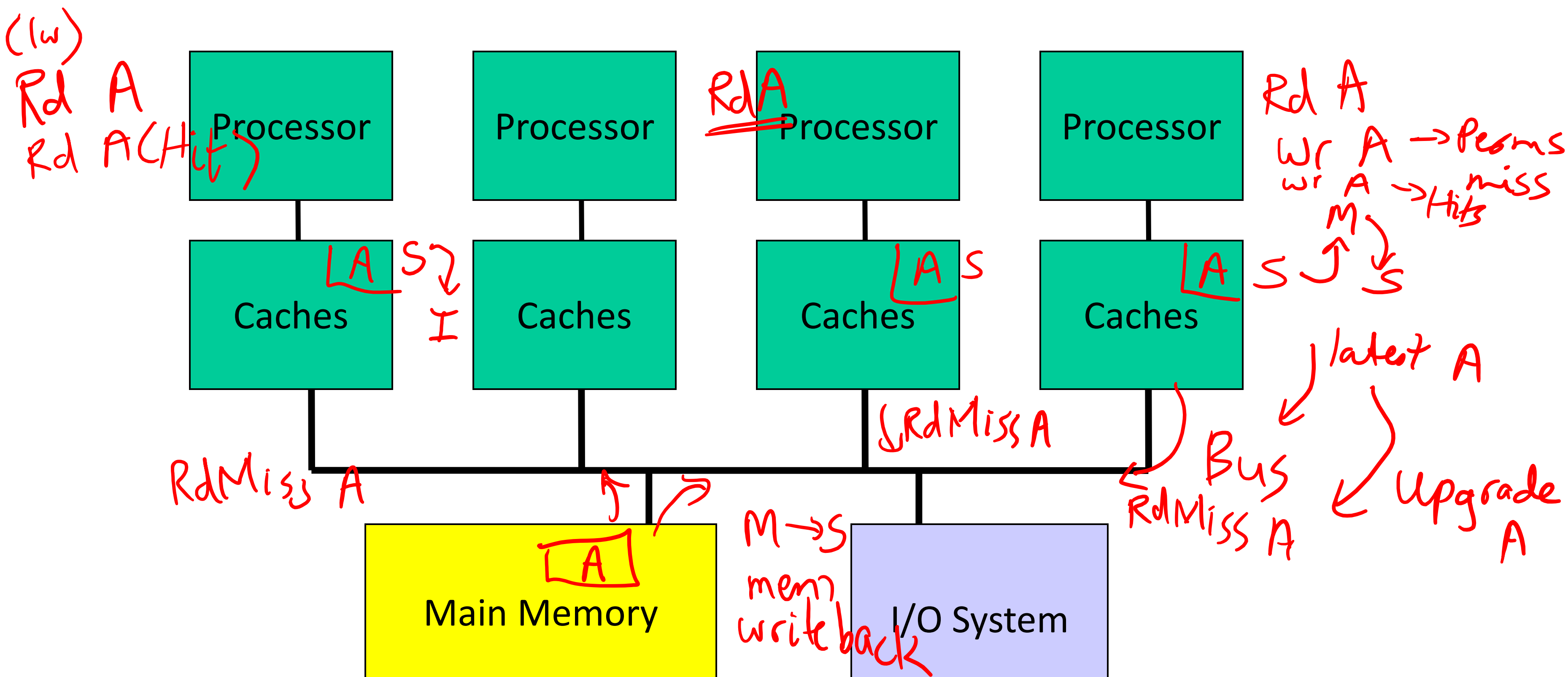
- Multiple processors connected to a single centralized memory – since all processors see the same memory organization → uniform memory access (UMA)

- Shared-memory because all processors can access the entire memory address space

- Can centralized memory emerge as a bandwidth bottleneck? – not if you have large caches and employ fewer than a dozen processors

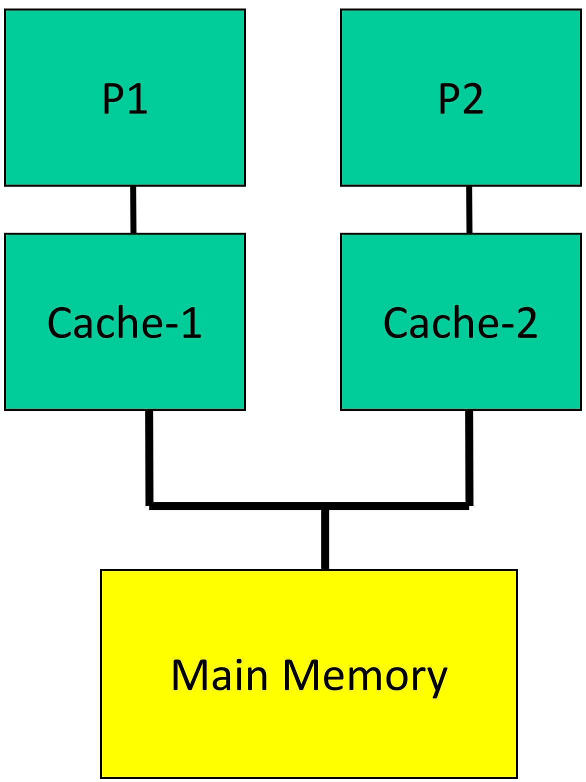
# Snooping-Based Protocols

- Three states for a block: invalid, shared, modified
- A write is placed on the bus and sharers invalidate themselves
- The protocols are referred to as MSI, MESI, etc.



# Example

- P1 reads X: not found in cache-1, request sent on bus, memory responds, X is placed in cache-1 in shared state
- P2 reads X: not found in cache-2, request sent on bus, everyone snoops this request, cache-1does nothing because this is just a read request, memory responds, X is placed in cache-2 in shared state




- P1 writes X: cache-1 has data in shared state (shared only provides read perms), request sent on bus, cache-2 snoops and then invalidates its copy of X, cache-1 moves its state to modified
- P2 reads X: cache-2 has data in invalid state, request sent on bus, cache-1 snoops and realizes it has the only valid copy, so it downgrades itself to shared state and responds with data, X is placed in cache-2 in shared state, memory is also updated

# Example

| Request | Cache Hit/Miss | Request on the bus | Who responds | State in Cache 1 | State in Cache 2 | State in Cache 3 | State in Cache 4 |
|---|---|---|---|---|---|---|---|
| | | | | Inv | Inv | Inv | Inv |
| P1: Rd X | Rd Miss | Rd X | Memory | S | Inv | Inv | Inv |
| P2: Rd X | Rd Miss | Rd X | Memory | S | S | Inv | Inv |
| P2: Wr X | Perms Miss | Upgrade X | No response. Other caches invalidate. | Inv | M | Inv | Inv |
| P3: Wr X | Wr Miss | Wr X | P2 responds | Inv | Inv | M | Inv |
| P3: Rd X | Rd Hit | - | - | Inv | Inv | M | Inv |
| P4: Rd X | Rd Miss | Rd X | P3 responds. Mem wrtbk | Inv | Inv | S | S |