COM SCI M151B Week 6

Aidan Jan

November 7, 2024

Out of Order Processor Recap

How to build an out-of-order processor?

- Dispatching and Renaming
- Retiring and Re-order buffer (ROB)
- Scheduling and reservation station
- Functional units, fire, and complete

What are the Challenges?

- Different design choices
- How to recover?
- How to forward?
- New issues?

Different Design Choices

Centralized RoB vs. ROB and RS

- In the centralized model, RS can be a part of ROB.
- Simplified design when sharing both.

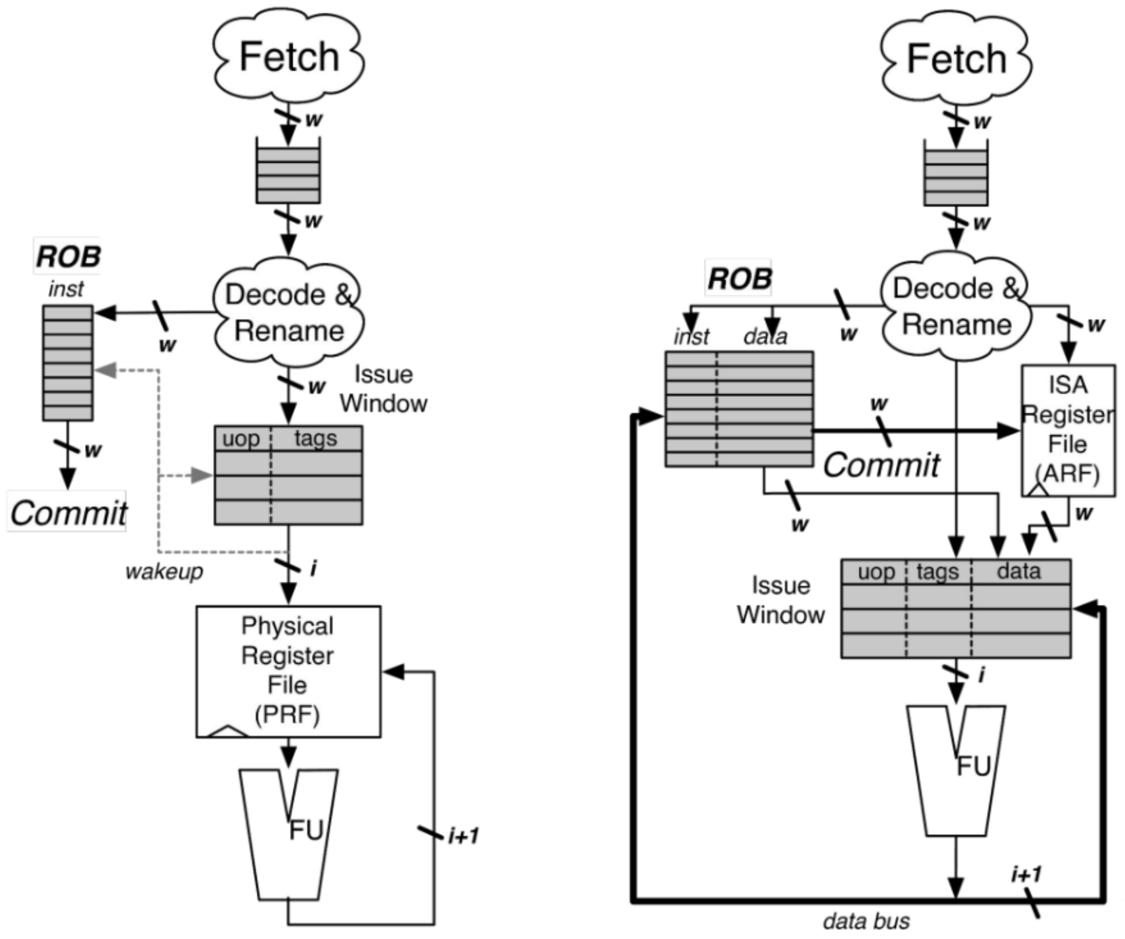
Dispatch

Scheduling Queue/Reservations Station

- Centralized?
  - You have one ROB connected to many functional units (single datapaths), and each instruction is assigned a functional unit to be run on.
  - Benefit: efficient
  - Detriment: complicated to implement
- Per execution unit?
  - Each functional unit has an ROB and you assign instructions directly to each one.
  - Benefit: easier, simpler wiring
  - Detriment: you have to schedule in such a way that the work loads for each functional unit is balanced.
- When to read data?

  - Read before issue → also called data capture
  - Read after issue → non-data capture
  - This is important because it changes the renaming algorithm!
- How to do the renaming?
  - Use a RAT and a physical register.
  - Use a tag system, store and broadcast those tags (need only an *architectural* register file.)

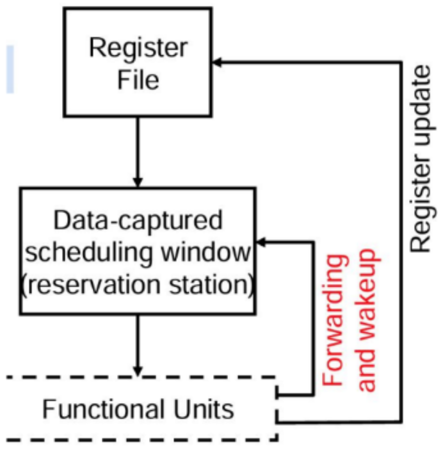
Explicit vs. Implicit

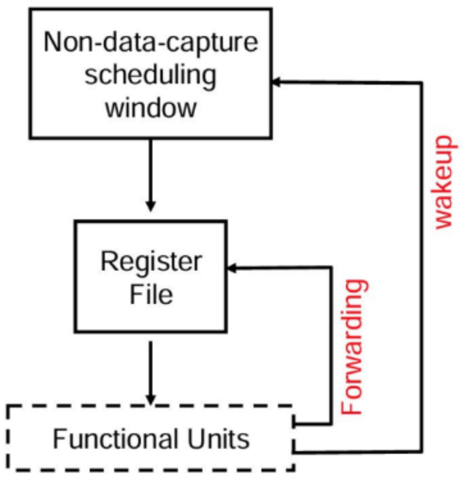


Explicit: left. Implicit: right.

Wakeup

- The load instruction is slow, and we only have one register file. The register can only load so many values at a time.
- Therefore, we have a process called wakeup. Instructions are "woken up" when data they request from the register file is ready for usage.





More Nuances

Where to store operands?

- Store values and tags in Reservation Station (RS), need only a separate physical register file.
  - (i.e., "reg read" happens during *dispatch.*)
- Store (physical) register numbers in RS, data held in a unified physical register file.
  - (i.e., "reg read" happens during *execute.*)

When to forward?

- Once instruction is complete (need more ports in ROB)
- Wait for "retire" (simpler but much slower)

How to recover/flush?

- Steps:
  - Flush from RS
  - Remove from ROB
  - Restore RAT
  - Free destination registers
  - Free functional units
  - Update PC
- Options:
  - Log-based: store "old" destination register, and restore RAT one instruction at a time
  - Checkpoint-based: have two versions for each table "messy" and "safe". Use "messy" for RS, copy values from "messy" to "safe" when an instruction is retired. If recovery is needed, copy "safe" to "messy".
  - We can always use a "hybrid" approach!

Handling Hazards

- RAW hazards: "ready" bits in RS take care of that!
- Control hazards: With ROB and RAT we can recover!
- Structural hazard: Ready bits for FU, stall if ROB or RS is full → need write-enable for fetch and decode pipeline register (similar to in-order design)

New hazard: memory dependency

- RAW: reading from a memory address BEFORE writing to it was finished.
- WAW: writing to the same address twice.
- WAR: do load first and then store.

How to fix?

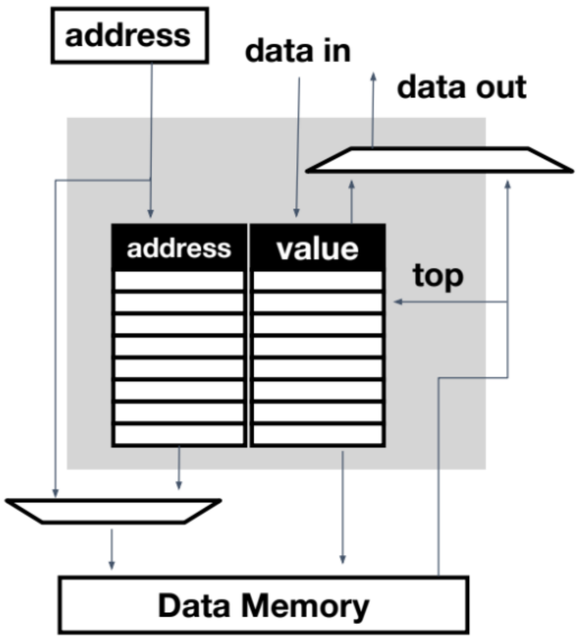
- Renaming won't work since addresses are not known! We use a store queue!

Store Queue (SQ)

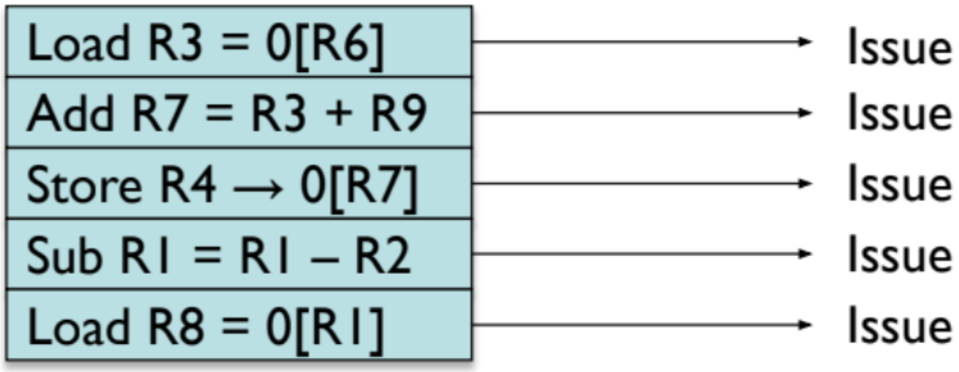
- Acts similar to ROB but for store instructions
- Assign an entry during decode (stall if full).
- Store the values in the queue when complete.
- Only write to the memory if the instruction is retired (ROB will send signal)
- This fixes WAW and WAR! (What about RAW?)

LW-Store Queue (LSQ)

- LW can scan the queue and data can be forwarded from store to loads.
- We need to make sure that we preserve the order of writes.
- This fixes RAW!



Memory (dis)Ambiguation



```
Load R3 = 0[R6]
Add R7 = R3 + R9
Store R4 → 0[R7]
Sub R1 = R1 – R2
Load R8 = 0[R1]
```

Two scenarios:

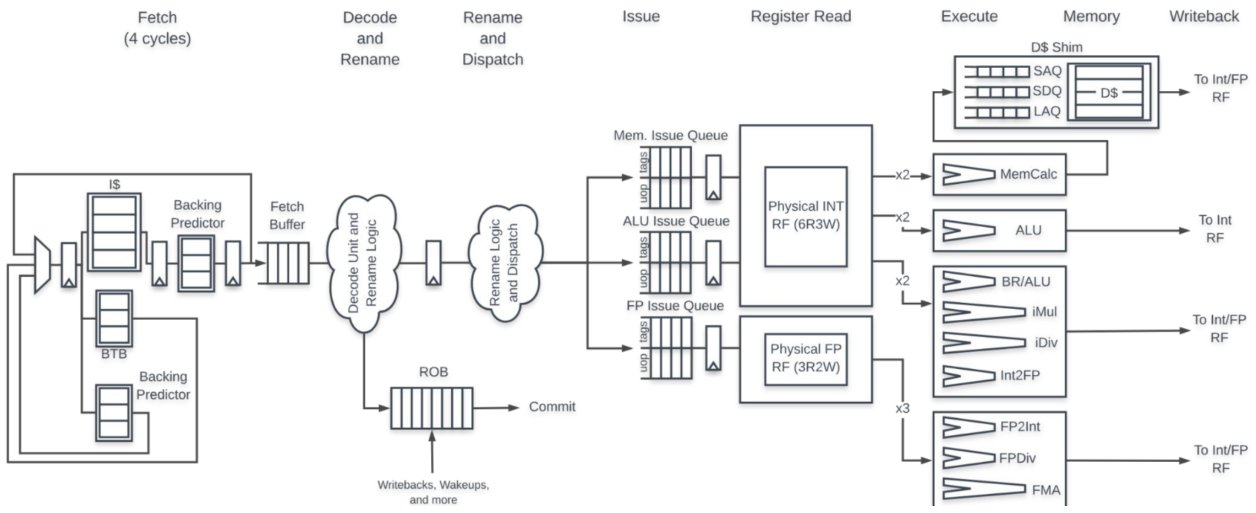
- R1 != R7
- R1 == R7

Why?

- value of R7 is dependent on loading the address of R6.
- Meanwhile, R8 is dependent on loading R1. What if they point to the same virtual location?
- Before the load finishes, we don't know if R1 and R7 are the same! (Stalling a few cycles to find out is slow.)
- We can instead guess whether or not they are equal!

This is like branch prediction, but with values instead.

Simplified Pipeline



Memory

We have spent the last five weeks looking at the CPU, and caught up with the current CPU design. What's next?

- Before, we always abstracted away the memory and assumed that it is always there, but in reality, memory is a big challenge
- Memory as it exists now is very slow. This is called the "Memory Wall".
- Accessing DRAM takes 1000's of cycles in modern processors. CPU speed is growing faster than memory speed.
  - The performance gap between memory and CPU is growing by around 50% per year.

Caching

The fundamental challenge of memory: memories can be **either** large **or** fast.
We need a memory hierarchy.

- Latches and Flip-Flops (aka Registers)
  - Very fast, parallel access.
  - Very *expensive* (one bit costs 10+ transistors.)
  - **Size:** a few KB
  - **Access time:** within a cycle (<1ns)
- Static RAM (SRAM)
  - Relatively fast, only one data word at a time.
  - Expensive (one bit costs 6+ transistors.)
  - **Size:** 10-100 KB
  - **Access time:** 2-5 to 10s cycles (1-3/10s ns)
- Dynamic RAM (DRAM)
  - Slower, one data word at a time, reading *destroys* content (refresh), needs special process for manufacturing.
  - Cheap (one bit costs only **one** transistor plus one capacitor.)
  - **Size:** 1-10s GB
  - **Access time:** 500-1000s cycles (>100ns)
- Disk (flash memory, hard disk)
  - Much slower, access takes a long time, **non-volatile.**
  - Very cheap.
  - **Size:** 1-10 TB.
  - **Access time:** 10k-100k cycles (>10ms = 10000000 ns)

We build a hierarchy of memory to compromise between fast and large memory!

- Data that is currently used goes into the registers.
- Commonly accessed data goes into SRAM
- Data that is used sometimes goes into DRAM
- Data that is rarely used goes into Disk.
- A given piece of data can be moved between the layers of the hierarchy if its usage changes.

Why Hierarchy would Work?

- **Temporal Locality**
  - Items accessed recently are likely to be accessed again soon.
  - Examples: instructions in a loop, induction variables
- **Spatial Locality**
  - Items *near* those accessed recently are likely to be accessed soon.
  - Examples: sequential instruction access, array data.

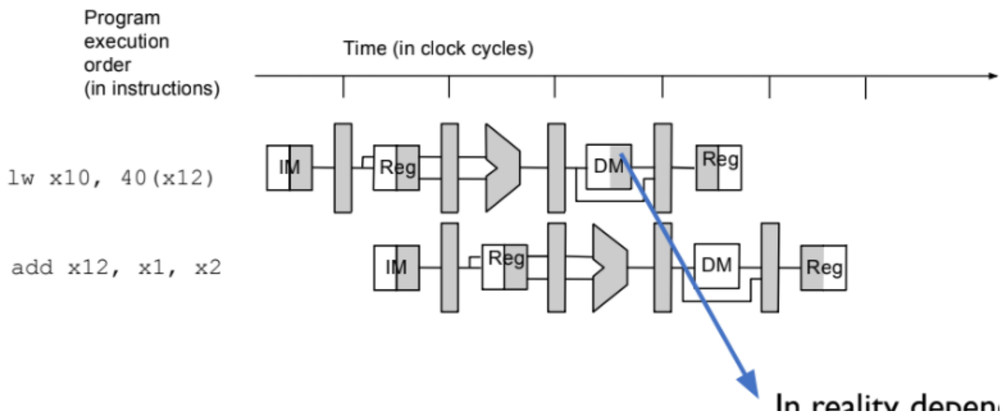
Cache

The cache stores temporary data close to the CPU.

- If the requested data exists in the cache, we call it a **cache hit**.
- If the requested data does not exist in the cache, it is a **cache miss**.
  - When cache miss, we need to bring the data from the main memory and put it in the cache.

Memory Delay

Recall the pipeline:



In reality, depending on where the data is, the MEM stage may take 1-1000 cycles! (Stalling!)

- If in L1 cache? 1 cycle.
- Else if in L2? 20 cycles. (STALL!)
- Else if in L3? 100 cycles. (STALL!)
- Else. . .

This is the reason why memory is so important.

Cache Design

The goal: minimizing the access to the main memory! (i.e., minimizing stalls!)

- Average time to access the memory:

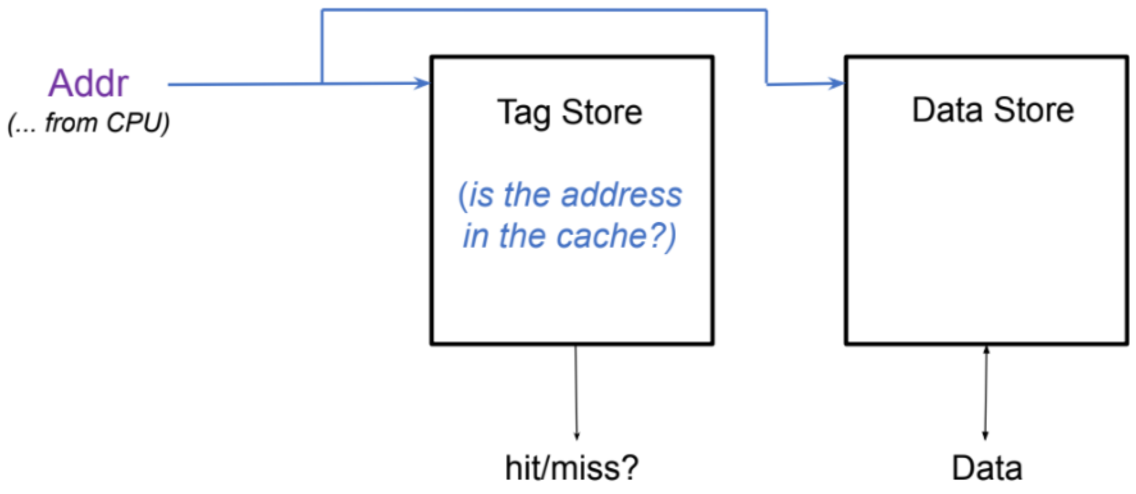
$$AAT = HitTime + MissRate \times MissPenalty$$

  - $HitTime$: Time it takes to access the (L1) cache.
  - $MissRate$: The average frequency of misses (in L1).
  - $MissPenalty$: The time required to access the main memory.

Problem: we have limited space in cache!
Solution: Cache *management*!

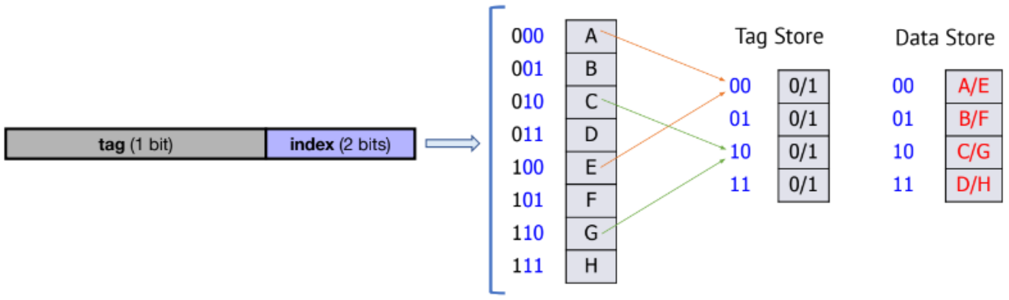
Where to store/find data?

- We have the address (32 bit)
- We need:
  1. Whether this address exists in the cache.
  2. The value/content
- Assuming a 32-bit PC, each unique value of PC is one line in memory = 4 billion lines of memory! This is not practical because it would be slow.
- The cache is only a tiny fraction of memory space! Therefore, one cache row is assigned to **many** memory lines.



What do we store as the tag?

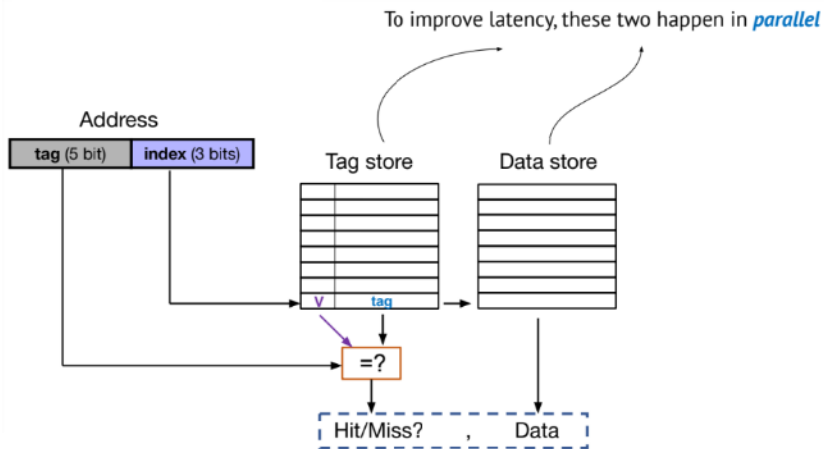
- Ideally, we have a one-to-one mapping of bits to tag. However, we need a lot of tags, and we have limited space. We will not achieve the one-to-one mapping. As a result, we may for example only store the last few bits of the tag, and use an index.
- Consider the example of three-bit tags. (e.g., we only have 3 bits to store an address. Instead of doing a 1:1 mapping, we divide it to a 1-bit tag and a 2-bit index.)



How to Organize Data in Cache?

- There are different addressing modes
  - Find the first available spot! (problem?)
  - Use index to only go to one line! (problem?)
- Solution: use a *Direct-Mapped Cache*
  - Same index should go to the same row!

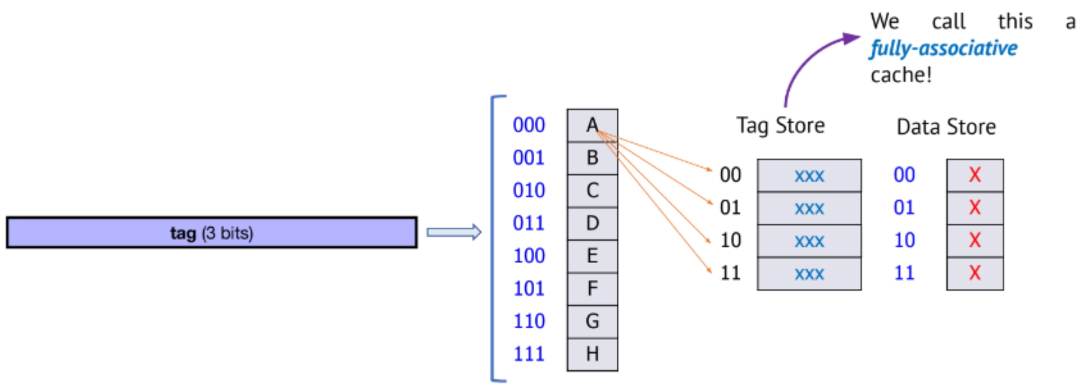
Direct-Mapped Cache Lookup



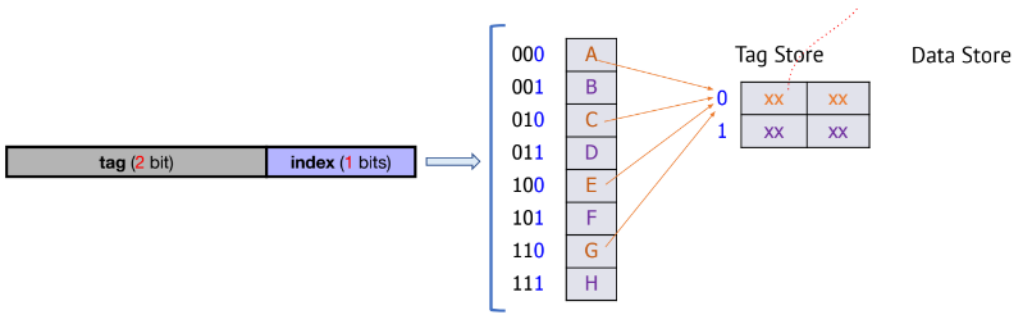
Suppose we want to search indexes 0, 8, 0, 8, 0, 8, in that order. Since our cache is not big enough, it misses every time!

Alternative to Direct-Mapped?

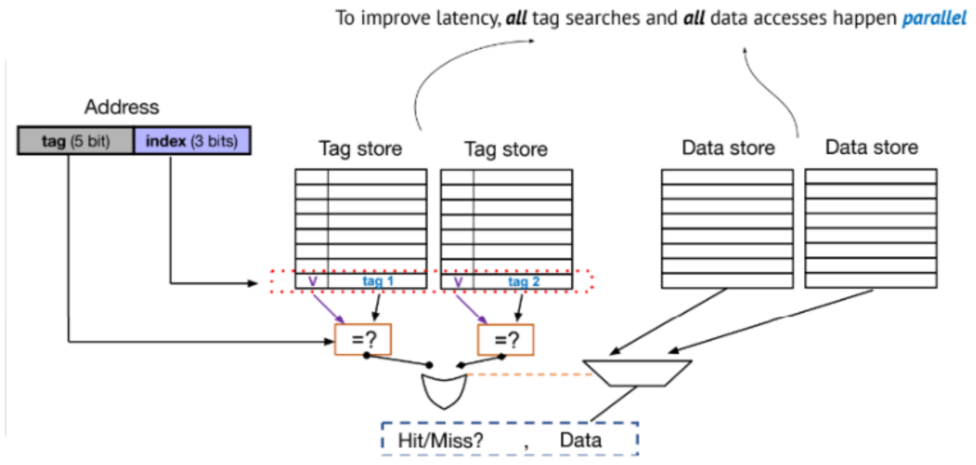
- Instead of assigning each memory address into a predetermined set, why don't we put the new data into any free set?



- Here, we can exploit locality; if the search indexes are 0, 8, 0, 8, 0, 8, then they can be stored next to each other. The miss rate is zero after the initial loads. However, this now comes with the tradeoff that the list is unsorted, which means searching for tags takes a lot longer.
  - Average Access Time = $HitTime + MissRate \times MissPenalty$
  - $HitTime$: Direct Mapped « Fully Associative
  - $MissRate$: Direct Mapped » Fully Associative
  - $MissPenalty$: Direct Mapped == Fully Associative
- Here, hit time and miss rate are tradeoffs. What if we have both?
- Introducting, *set-associative* cache
  - Add associativity *within* each set!



We call each column **way**. This cache is 2-way set-associative.



    - Average Access Time = $HitTime + MissRate \times MissPenalty$
    - $HitTime$: DM < SA « FA
    - $MissRate$: DM » SA > FA
    - $MissPenalty$: DM == FA == SA

- Set-Associative mapping is a compromise between Direct-Mapped and Fully Associative.

What happens on a miss?

Specifically, which cache entry do we evict and replace?

- Option 1: pick at **random**!
    - Very easy to implement
    - Not the best usage for *temporal* locality
- Option 2: pick the **least-recently-used (LRU)**!
    - Find the oldest line and evice that!
    - This preserves locality!
    - This needs to store age and adds overhead (bad)!
    - Not practical for large caches.
- Option 3: The compromise: **Pseudo-LRU** Replacement Policy
    - Store only one bit per row, all start assigned to 0.
    - On a hit, set the bit to 1.
    - On a miss, find the first row with flag==0.
    - If all bits are 1, pick the first row and reset all bits to 0.
- Can we do better?

Replacement Policy Tradeoffs

- Storage vs. Accuracy
    - Possible solutions:
        * Approximate (pseudo-LRU)
        * Adaptive (Change strategies based on how it has been running)
        * Hybrid (Combine multiple strategies)
    - **Important Note:** replacement policy performance is quite dependent on the access pattern (i.e., what type of locality are we seeing).

Three Types of Miss in a Cache

- Compulsory Miss
    - To have something in the cache, first it must be *fetched*.
    - The initial fetch of anything is a *miss*.
    - Also called unique references or first-time references.
- Capacity Miss
    - A miss that occurs due to the **limited capacity** in a <u>fully-associative cache</u>.

- Conflict Miss
  - For <u>set associative</u> or *direct-mapped* only.
  - Misses due to the index bits matching (conflicts).
  - Also called mapping misses.