Final Exam Review

Dr. Tao Xie

These slides are adapted from notes by Dr. David Patterson (UCB)

Final Exam Review

Focus on materials after the midterm, including

- Instruction Level Parallelism
  - Scoreboard
  - Tomasula Method
- Cache & Memory Design
- Cache & Memory Performance

Forms of Parallelism

- Process-level
  - How do we exploit it? What are the challenges?
  - Examples?
- Thread-level
  - How do we exploit it? What are the challenges?
  - Examples?
- Loop-level
  - What is really loop level parallelism? What percentage of a program's time is spent inside loops?
- Instruction-level
  - Focus of Chapter 2

Coarse grain

Human intervention?

Fine Grain

Increasing Instruction-Level Parallelism (ILP)

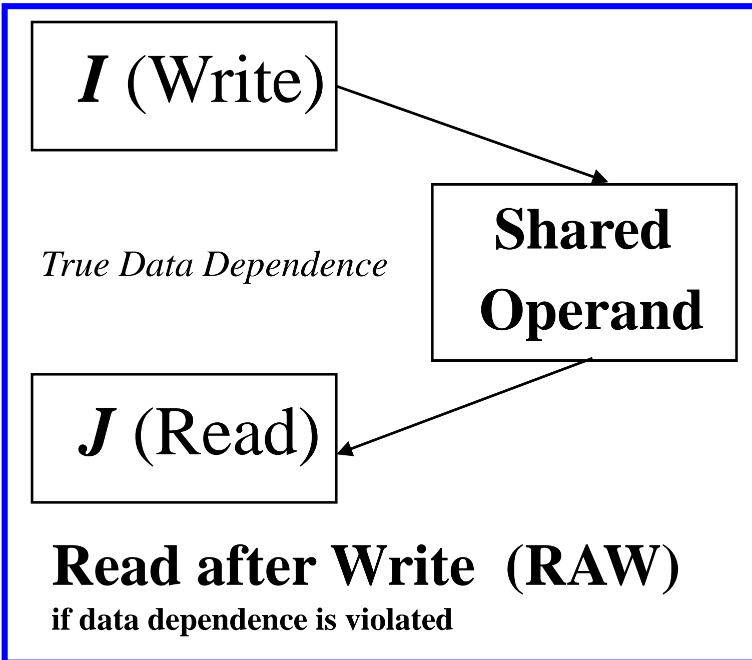
- A common way to increase parallelism among instructions is to exploit parallelism among iterations of a loop
  - (i.e Loop Level Parallelism, LLP).
- This is accomplished by unrolling the loop either statically by the compiler, or dynamically by hardware, which increases the size of the basic block present. This resulting larger basic block provides more instructions that can be scheduled or re-ordered by the compiler to eliminate more stall cycles.
- In this loop every iteration can overlap with any other iteration. Overlap within each iteration is minimal.

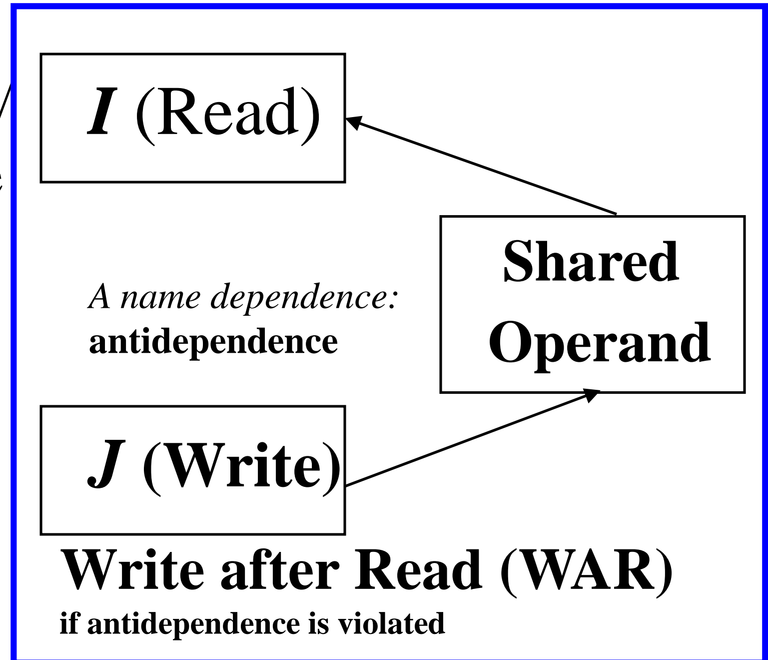
```
for (i=1; i<=1000; i=i+1;)
        x[i] = x[i] + y[i];
```

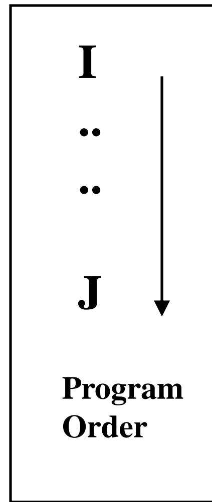
4 vector instructions:

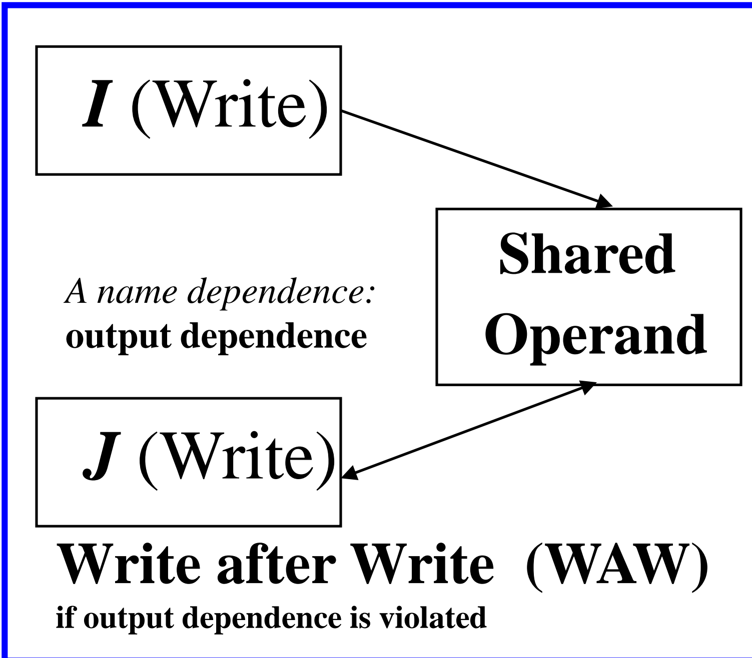
Load Vector X
Load Vector Y
Add Vector X, X, Y
Store Vector X

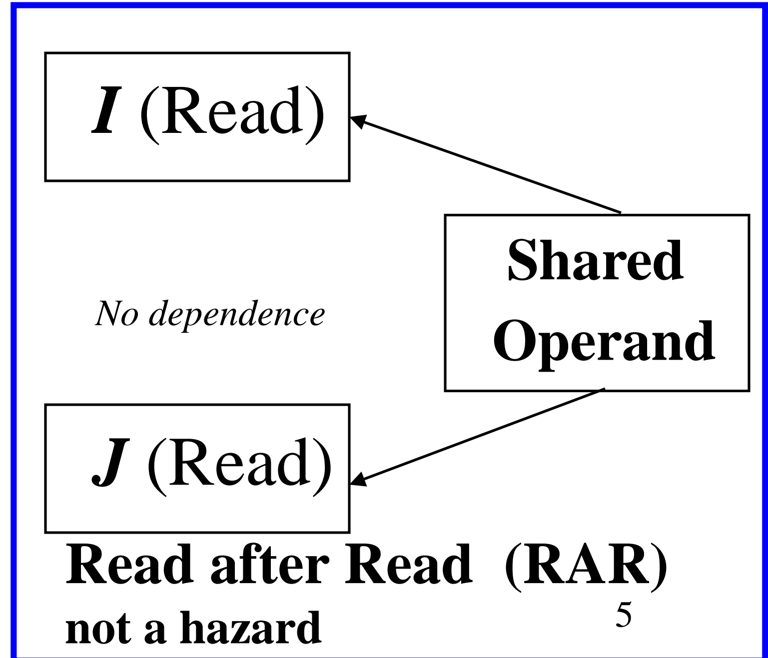
Data Hazard/ Dependence











Control Dependencies

- Determines the ordering of an instruction with respect to a branch instruction.
- Example of control dependence in the **then** part of an **if** statement:

```
if  p1 {
        S1;
};
If p2  {
        S2;
}
```

S1 is control dependent on p1
S2 is control dependent on p2 but not on p1

- Branch predictions
- Branch delay slot

Dynamic Pipeline Scheduling

- Dynamic instruction scheduling is accomplished by:
  - **Dividing the Instruction Decode ID stage into two stages:**
    - Issue: Decode instructions, check for structural hazards.
      - A record of data dependencies is constructed as instructions are issued
      - This creates a dynamically-constructed dependency graph for the window of instructions in-flight (being processed) in the CPU.
    - Read operands: Wait until data hazard conditions, if any, are resolved, then read operands when available (then start execution)

    (All instructions pass through the issue stage in order but can be stalled or pass each other in the read operands stage).
  - **In the instruction fetch stage IF, fetch an additional instruction every cycle into a latch or several instructions into an instruction queue.**
  - **Increase the number of functional units to meet the demands of the additional instructions in their EX stage.**

Always done in program order (→ Issue)

Can be done out of program order (→ Read operands)

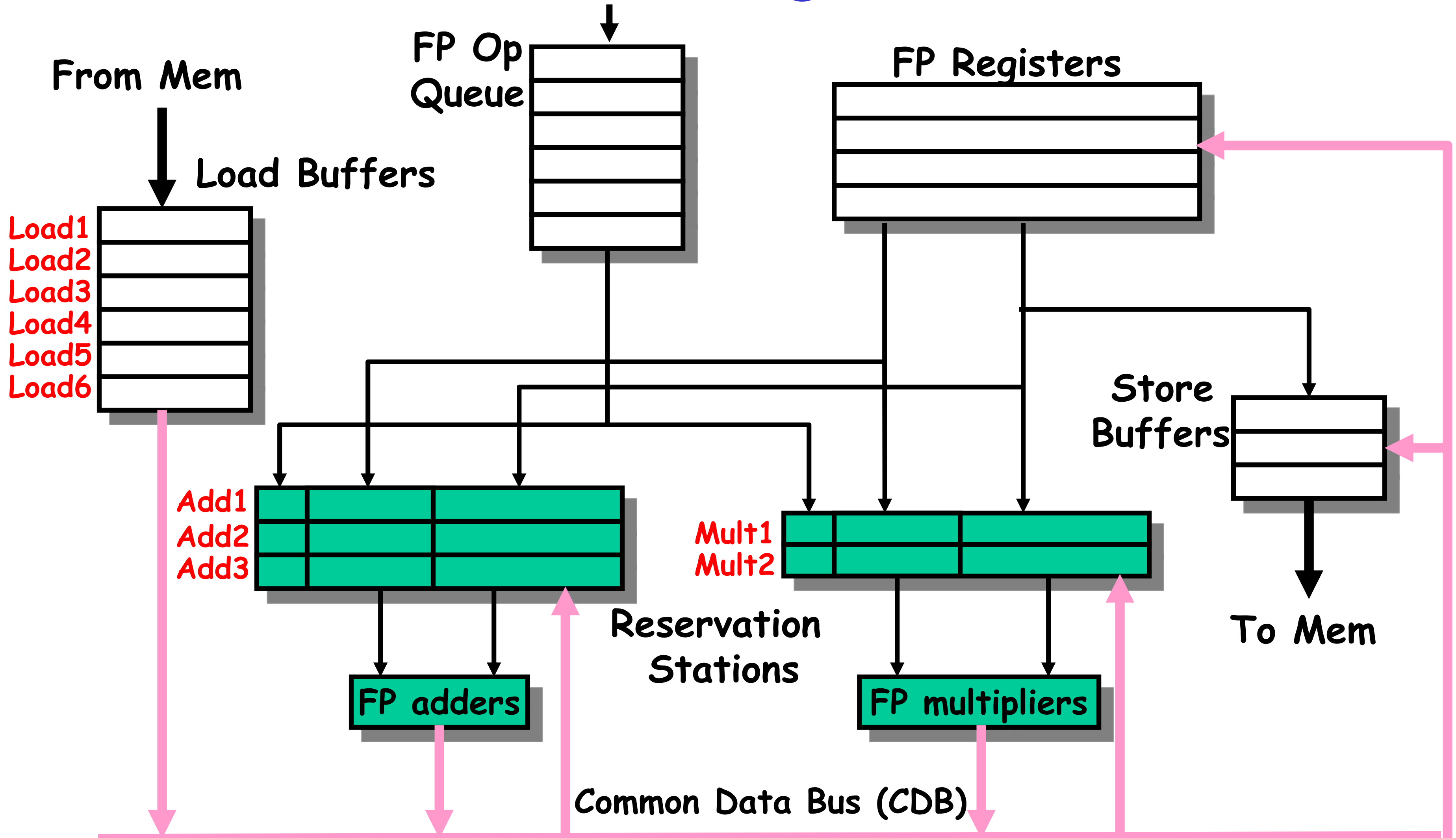
- Two approaches to dynamic scheduling:
  - **Dynamic scheduling with the Scoreboard used first in CDC6600 (1963)**
  - **The Tomasulo approach pioneered by the IBM 360/91 (1966)**

Tomasulo Algorithm Vs. Scoreboard

- Control & buffers *distributed* with Functional Units (FUs) Vs. centralized in Scoreboard:
  - **FU buffers are called *"reservation stations"* which have pending instructions and operands and other instruction status info (including data dependencies).**
  - **Reservations stations are sometimes referred to as "physical registers" or "renaming registers" as opposed to architecture registers specified by the ISA.**
- ISA Registers in instructions are replaced by either values (if available) or pointers (renamed) to reservation stations (RS) that will supply the value later:
  - **This process is called *register renaming*.**
    - Register renaming eliminates WAR, WAW hazards.
  - **Allows for a *hardware-based* version of loop unrolling.**
  - **More reservation stations than ISA registers are possible, leading to optimizations that compilers can't achieve and prevents the number of ISA registers from becoming a bottleneck.**
- Instruction results go (forwarded) from RSs to RSs , *not through registers*, over *Common Data Bus (CDB)* that broadcasts results to all waiting RSs (dependant instructions).
- Loads and Stores are treated as FUs with RSs as well.

(In Chapter 2)

Tomasulo Organization

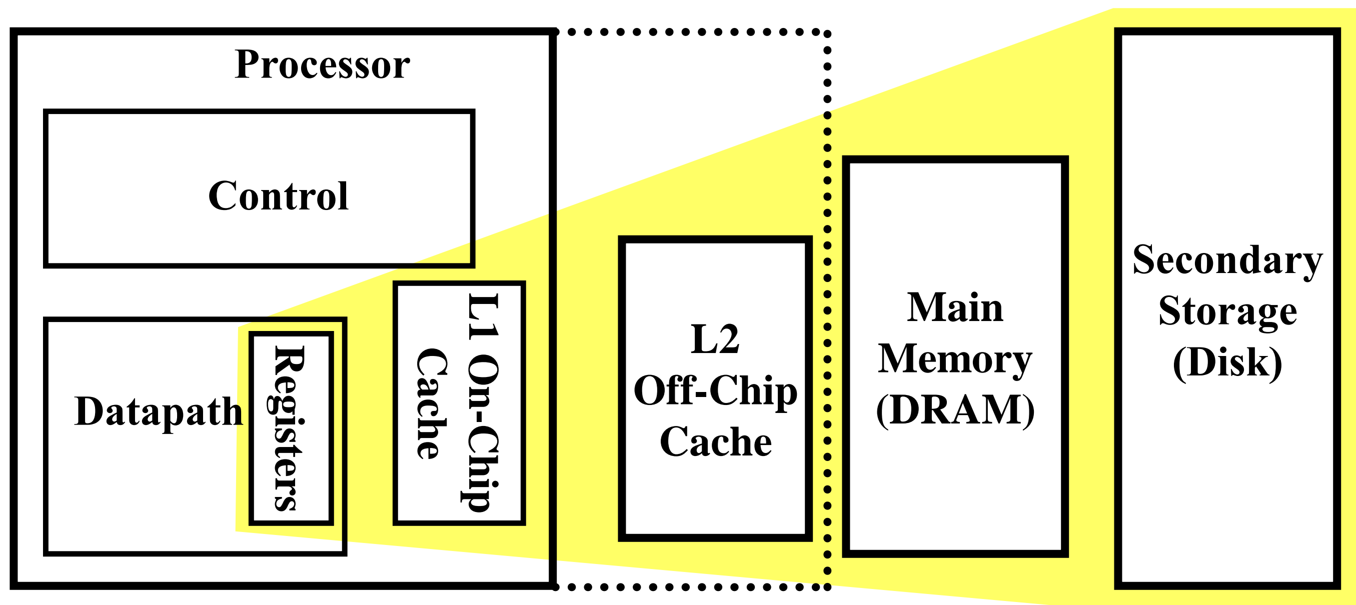


Normal data bus: data + destination

Common data bus: data + source

Memory Hierarchy - the Big Picture

- Problem: memory is too slow and too small
- Solution: memory hierarchy



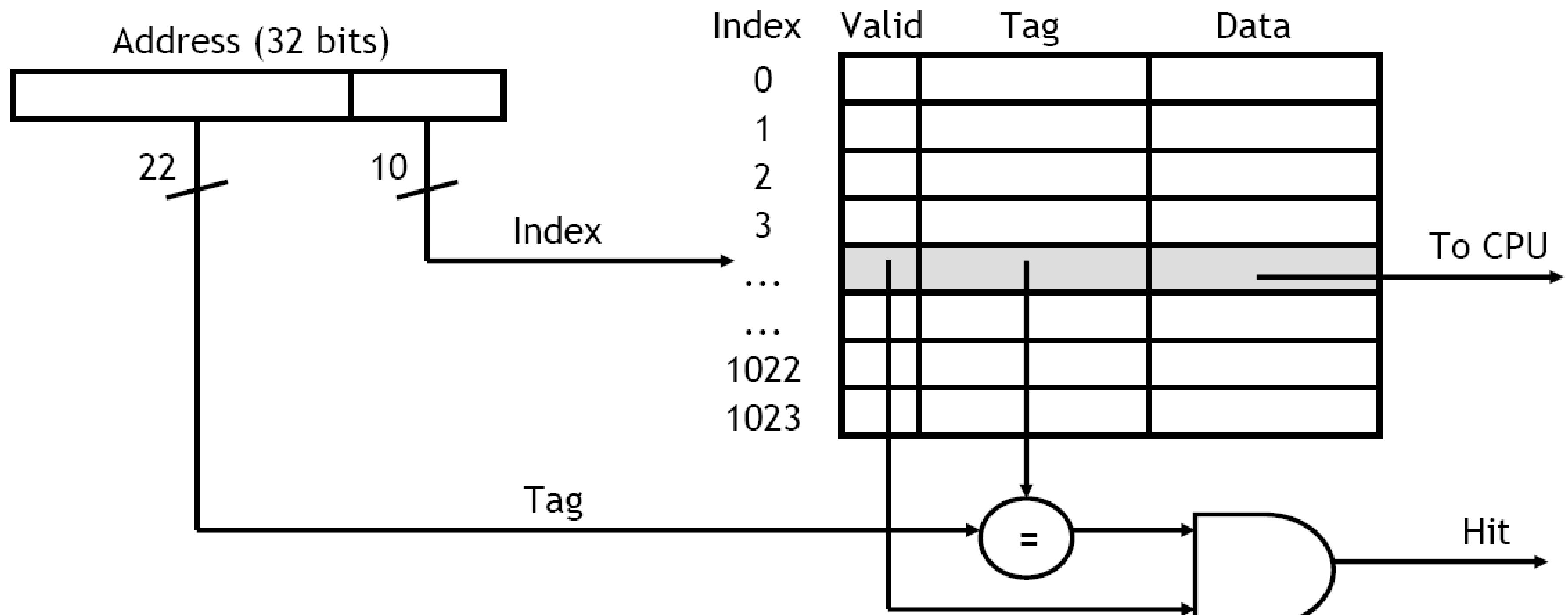
Speed (ns):	**0.25-0.5**	**0.5-25**	**80-250**	**5,000,000 (5ms)**
Size (bytes):	**<1K**	**<16M**	**<16G**	**>100G**

Fundamental Cache Questions

- Q1: Where can a block be placed in the upper level?
  (Block placement)
- Q2: How is a block found if it is in the upper level?
  (Block identification)
- Q3: Which block should be replaced on a miss?
  (Block replacement)
- Q4: What happens on a write?
  (Write strategy)

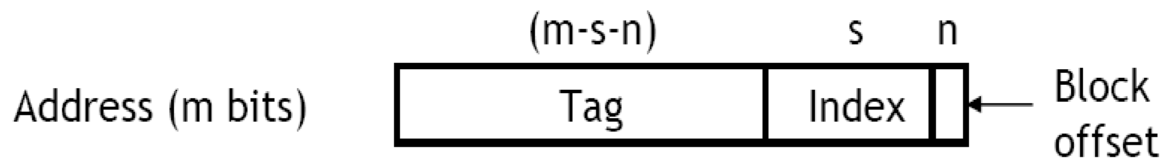
What happens on a cache hit

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest k bits of the address will index a block in the cache.
  - If the block is valid and the tag matches the upper $(m - k)$ bits of the m-bit address, then that data will be sent to the CPU.
- Here is a diagram of a 32-bit memory address and a 2^{10}-byte cache.



Locating a set associative block

- We can determine where a memory address belongs in an associative cache in a similar way as before.
- If a cache has 2^s sets and each block has 2^n bytes, the memory address can be partitioned as follows.



- Our arithmetic computations now compute a set index, to select a *set* within the cache instead of an individual block.

$$\text{Block Offset} = \text{Memory Address mod } 2^n$$

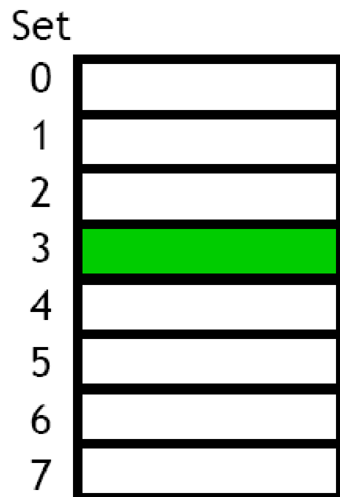
$$\text{Block Address} = \text{Memory Address } / \ 2^n$$

$$\text{Set Index} = \text{Block Address mod } 2^s$$

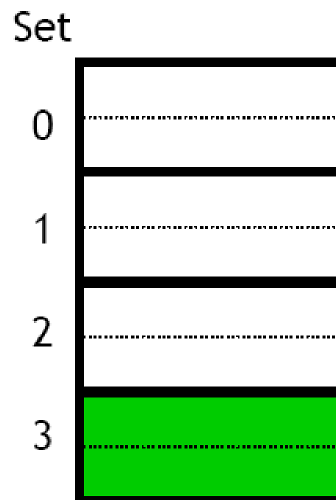
Example placement in set-associative caches

- Where would data from memory byte address 6195 be placed, assuming the eight-block cache designs below, with 16 bytes per block?
- 6195 in binary is 00...0110000 011 0011.
- Each block has 16 bytes, so the lowest 4 bits are the block offset.
- For the 1-way cache, the next three bits (011) are the set index.
  For the 2-way cache, the next two bits (11) are the set index.
  For the 4-way cache, the next one bit (1) is the set index.
- The data may go in *any* block, shown in green, within the correct set.

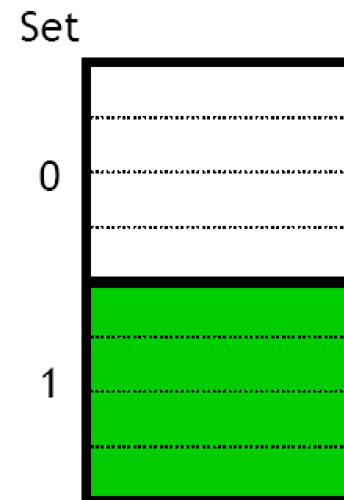
1-way associativity
8 sets, 1 block each



2-way associativity
4 sets, 2 blocks each



4-way associativity
2 sets, 4 blocks each

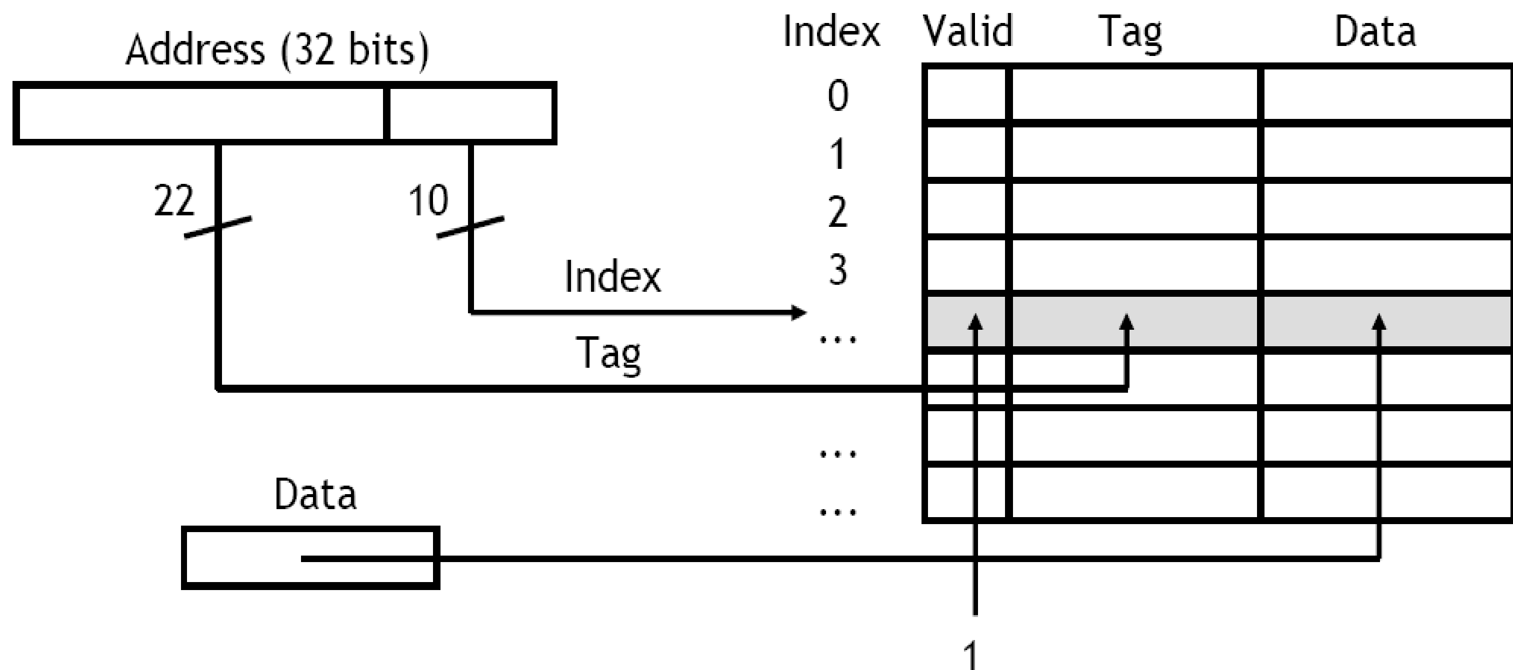


What happens on a cache miss

- The delays that we've been assuming for memories (e.g., 2ns) are really assuming cache hits.
  - If our CPU implementations accessed main memory directly, their cycle times would have to be much larger.
  - Instead we assume that most memory accesses will be cache hits, which allows us to use a shorter cycle time.
- However, a much slower main memory access is needed on a cache miss. The simplest thing to do is to stall the pipeline until the data from main memory can be fetched (and also copied into the cache).

Loading a block into the cache

- After data is read from main memory, putting a copy of that data into the cache is straightforward.
  - The lowest k bits of the address specify a cache block.
  - The upper $(m - k)$ address bits are stored in the block's tag field.
  - The data from main memory is stored in the block's data field.
  - The valid bit is set to 1.

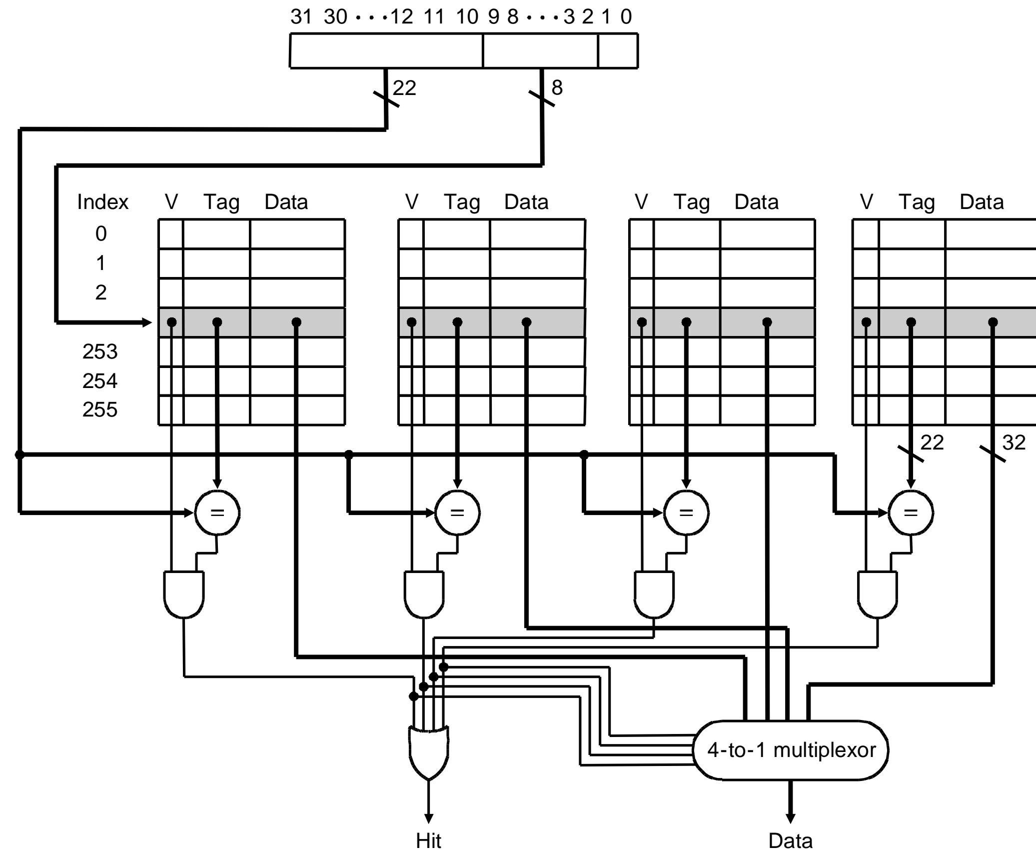


Summary

- Basic ideas of caches.
  - By taking advantage of spatial and temporal locality, we can use a small amount of fast but expensive memory to dramatically speed up the average memory access time.
  - A cache is divided into many blocks, each of which contains a valid bit, a tag for matching memory addresses to cache contents, and the data itself.
- Next we'll look at some more advanced cache organizations and see how to measure the performance of memory systems.

Set Associative Cache Design

- Key idea:
  - Divide cache into <u>sets</u>
  - Allow block <u>anywhere</u> in a set
- Advantages:
  - Better hit rate
- Disadvantage:
  - More tag bits
  - More hardware
  - Higher access time



A Four-Way Set-Associative Cache

Cache Performance Measures

- *Hit rate*: fraction found in the cache
  - So high that we usually talk about *Miss rate = 1 - Hit Rate*
- *Hit time*: time to access the cache
- *Miss penalty*: time to replace a block from lower level, including time to replace in CPU
  - *access time*: time to acccess lower level
  - *transfer time*: time to transfer block
- Average memory-access time (AMAT)

  = Hit time + Miss rate x Miss penalty (ns or clocks)

Cache Performance

- Miss-oriented Approach to Memory Access:

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemAccess}{Inst} \times MissRate \times MissPenalty \right) \times CycleTime$$

$$CPUtime = IC \times \left(CPI_{Execution} + \frac{MemMisses}{Inst} \times MissPenalty \right) \times CycleTime$$

  - $CPI_{Execution}$ includes ALU and Memory instructions

- Separating out Memory component entirely
  - AMAT = Average Memory Access Time
  - CPI_{ALUOps} does not include memory instructions

$$CPUtime = IC \times \left(\frac{AluOps}{Inst} \times CPI_{AluOps} + \frac{MemAccess}{Inst} \times AMAT \right) \times CycleTime$$

$$\begin{aligned} AMAT &= HitTime + MissRate \times MissPenalty \\ &= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst} \right) + \\ &\quad \left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data} \right) \end{aligned}$$

Impact on Performance

- Suppose a processor executes at
  - Clock Rate = 200 MHz (5 ns per cycle), Ideal (no misses) CPI = 1.1
  - 50% arith/logic, 30% ld/st, 20% control
- Suppose that 10% of memory operations get 50 cycle miss penalty
- Suppose that 1% of instructions get same miss penalty

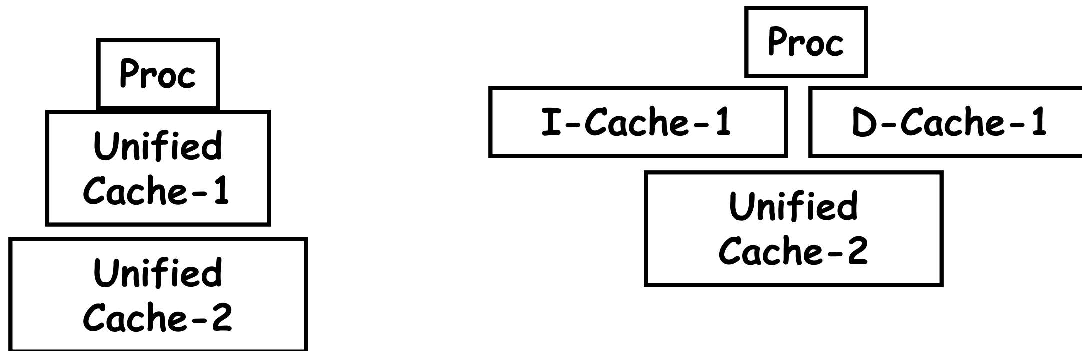
- CPI = ideal CPI + average stalls per instruction
  =1.1(cycles/ins) +
  [0.30 (DataMops/ins) x 0.10 (miss/DataMop) x 50 (cycle/miss)] +
  [1 (InstMop/ins) x 0.01 (miss/InstMop) x 50 (cycle/miss)]
  = (1.1 + 1.5 + .5) cycle/ins = 3.1

- AMAT=(1/1.3)x[1+0.01x50]+(0.3/1.3)x[1+0.1x50]=2.54

$$
\begin{aligned}
AMAT &= HitTime + MissRate \times MissPenalty \\
&= \left(HitTime_{Inst} + MissRate_{Inst} \times MissPenalty_{Inst}\right) + \\
&\quad \left(HitTime_{Data} + MissRate_{Data} \times MissPenalty_{Data}\right)
\end{aligned}
$$

Unified vs Split Caches

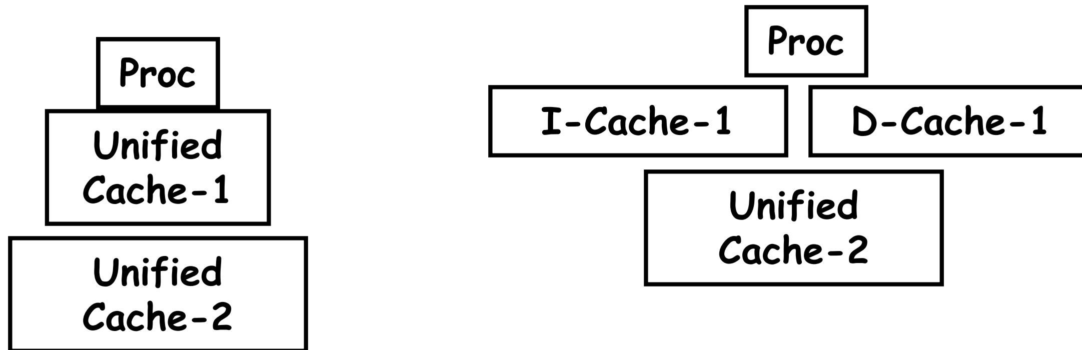
- Unified vs Separate I&D



- Example:
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- Which is better (ignore L2 cache)?
  - Assume 33% data ops ⇒ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that *data* hit has 1 stall for unified cache (only one port)

Unified vs Split Caches

- Unified vs Separate I&D



- Example:
  - 16KB I&D: Inst miss rate=0.64%, Data miss rate=6.47%
  - 32KB unified: Aggregate miss rate=1.99%
- Which is better (ignore L2 cache)?
  - Assume 33% data ops ⇒ 75% accesses from instructions (1.0/1.33)
  - hit time=1, miss time=50
  - Note that *data* hit has 1 stall for unified cache (only one port)

$AMAT_{Harvard}$=75%x(1+0.64%x50)+25%x(1+6.47%x50) = 2.05

$AMAT_{Unified}$=75%x(1+1.99%x50)+25%x(1+1+1.99%x50)= 2.24

Email Support

Provide online email support Until noon on Dec. 18.