Caching
Caches break down an address into which parts?

<table>
<thead>
<tr>
<th>Letter</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Tag, delay, length</td>
</tr>
<tr>
<td>B</td>
<td>Max, min, average</td>
</tr>
<tr>
<td>C</td>
<td>High-order and low-order</td>
</tr>
<tr>
<td>D</td>
<td>Tag, index, offset</td>
</tr>
<tr>
<td>E</td>
<td>Opcode, register, immediate</td>
</tr>
</tbody>
</table>
Caches operate on units of memory called…

<table>
<thead>
<tr>
<th>Letter</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Lines</td>
</tr>
<tr>
<td>B</td>
<td>Pages</td>
</tr>
<tr>
<td>C</td>
<td>Bytes</td>
</tr>
<tr>
<td>D</td>
<td>Words</td>
</tr>
<tr>
<td>E</td>
<td>None of the above</td>
</tr>
</tbody>
</table>

The correct answer is **Lines**.
The types of locality are...

<table>
<thead>
<tr>
<th>Letter</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Punctual, tardy</td>
</tr>
<tr>
<td>B</td>
<td><strong>Spatial and Temporal</strong></td>
</tr>
<tr>
<td>C</td>
<td>Instruction and data</td>
</tr>
<tr>
<td>D</td>
<td>Write through and write back</td>
</tr>
<tr>
<td>E</td>
<td>Write allocate and no-write allocate</td>
</tr>
</tbody>
</table>
Virtual memory can make the memory available appear to be…

<table>
<thead>
<tr>
<th>Letter</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>More secure</td>
</tr>
<tr>
<td>B</td>
<td>Smaller</td>
</tr>
<tr>
<td>C</td>
<td>Multifaceted</td>
</tr>
<tr>
<td>D</td>
<td>Cached</td>
</tr>
<tr>
<td>E</td>
<td>Larger</td>
</tr>
</tbody>
</table>
A sequence of caches, each larger and slower than the last is a…

<table>
<thead>
<tr>
<th>Letter</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Memory stack</td>
</tr>
<tr>
<td>B</td>
<td>Memory hierarchy</td>
</tr>
<tr>
<td>C</td>
<td>Paging system</td>
</tr>
<tr>
<td>D</td>
<td>Cache machine</td>
</tr>
<tr>
<td>E</td>
<td>Von Neumann Machine</td>
</tr>
</tbody>
</table>
Key Point

• What are
  • Cache lines
  • Tags
  • Index
  • offset

• How do we find data in the cache?
• How do we tell if it’s the right data?
• What decisions do we need to make in designing a cache?
• What are possible caching policies?
The Memory Hierarchy

• There can be many caches stacked on top of each other
• if you miss in one you try in the “lower level cache”  Lower level, mean higher number
• There can also be separate caches for data and instructions. Or the cache can be “unified”
• to wit:
  • the L1 data cache (d-cache) is the one nearest processor. It corresponds to the “data memory” block in our pipeline diagrams
  • the L1 instruction cache (i-cache) corresponds to the “instruction memory” block in our pipeline diagrams.
  • The L2 sits underneath the L1s.
  • There is often an L3 in modern systems.
Typical Cache Hierarchy
The Memory Hierarchy and the ISA

- The details of the memory hierarchy are not part of the ISA
  - These are implementations detail.
  - Caches are completely transparent to the processor.
- The ISA...
  - Provides a notion of main memory, and the size of the addresses that refer to it (in our case 32 bits)
  - Provides load and store instructions to access memory.
- The memory hierarchy is all about making main memory fast.
Recap: Locality

• Temporal Locality
  • Referenced item tends to be referenced again soon.

• Spatial Locality
  • Items close by referenced item tends to be referenced soon.
  • example: consecutive instructions, arrays
Cache organization
What is Cache?

- **Cache** is a hardware **hash** table!
  - each hash entry is a **block**
    - caches operate on “blocks”
    - cache blocks are a power of 2 in size. Contains multiple words of memory
    - usually between 16B-128Bs
    - need \( \lg(\text{block size}) \) bits offset field to select the requested word/byte
  - **hit**: requested data is in the table
  - **miss**: requested data is not in the table
  - basic hash function:
    - \( \text{block address} = \frac{\text{byte address}}{\text{block size}} \)
    - \( \text{block address} \mod \# \text{ of block} \)
Recap: Accessing cache

Tag: the high order address bits stored along with the data to identify the actual address of the cache line.

Block (cacheline): The basic unit of data in a cache. Contains data with the same block address (Must be consecutive)

Hit: The data was found in the cache.

Miss: The data was not found in the cache.

Offset: The position of the requesting word in a cache block.
Dealing the Interference

• By bad luck or pathological happenstance a particular line in the cache may be highly contended.
• How can we deal with this?
Interfering Code.

int foo[129]; // 4*129 = 516 bytes
int bar[129]; // Assume the compiler aligns these at 512 byte boundaries

while(1) {
    for (i = 0; i < 129; i++) {
        s += foo[i]*bar[i];
    }
}

• Assume a 1KB (0x400 byte) cache.
• Foo and Bar map into exactly the same part of the cache
• Is the miss rate for this code going to be high or low?
• What would we like the miss rate to be?
  • Foo and Bar should both (almost) fit in the cache!
Associativity

• (set) Associativity means providing more than one place for a cache line to live.
• The level of associativity is the number of possible locations
  • 2-way set associative
  • 4-way set associative
• One group of lines corresponds to each index
  • it is called a “set”
• Each line in a set is called a “way”
Way-associative cache

blocks sharing the same index are a “set”

Valid tag data

Hit?

Way 0

Way 1

Block / cacheline

Tag

Index

Offset

Block / line address
Way associativity and cache performance

- **Miss rate**
  - 1 KB
  - 2 KB
  - 4 KB
  - 8 KB
  - 16 KB
  - 32 KB
  - 64 KB
  - 128 KB

- **Associativity**
  - One-way
  - Two-way
  - Four-way
  - Eight-way

- **Cache Size**: 256
Fully Associative and Direct Mapped Caches

• At one extreme, a cache can have one, large set.
  • The cache is then fully associative
• At the other, it can have one cache line per set
  • Then it is direct mapped
C = ABS

- **C = ABS**
  - **C**: Capacity
  - **A**: Way-Associativity
    - How many blocks in a set
    - 1 for direct-mapped cache
  - **B**: Block Size (Cacheline)
    - How many bytes in a block
  - **S**: Number of Sets:
    - A set contains blocks sharing the same index
    - 1 for fully associate cache
Corollary of $C = ABS$

- offset bits: $\log_2(B)$
- index bits: $\log_2(S)$
- tag bits: $\text{address\_length} - \log_2(S) - \log_2(B)$
  - address\_length is 32 bits for 32-bit machine
- $(\text{address} / \text{block\_size}) \% S = \text{set\_index}$
Athlon 64

• L1 data (D-L1) cache configuration of Athlon 64
  • Size 64KB, 2-way set associativity, 64B block
  • Assume 32-bit memory address

Which of the following is correct?

A. Tag is 17 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above

\[
\text{Lines} = \frac{\text{Cache size}}{\text{Linesize}} = \frac{64KB}{64B} = 1024
\]

\[
\text{Sets} = \frac{\text{Lines}}{\text{Assoc}} = \frac{1024}{2} = 512
\]

\[
\log_2 (512) = 9
\]
Core 2

• L1 data (D-L1) cache configuration of Core 2 Duo
  • Size 32KB, 8-way set associativity, 64B block
  • Assume 32-bit memory address
  • Which of the following is NOT correct?
    A. Tag is 20 bits
    B. Index is 6 bits
    C. Offset is 6 bits
    D. The cache has 128 sets
How caches works
What happens on a write? (Write Allocate)

- **Write hit?**
  - Update in-place
  - Write to lower memory (Write-Through Policy)
  - Set dirty bit (Write-Back Policy)

- **Write miss?**
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
    - Miss penalty
Write-back v.s. write-through

How many of the following statements about write-back and write-through policies are correct?

- Write back can reduce the number of writes to lower-level memory hierarchy
- The average write response time of write-back is better
- A read miss may still result in writes if the cache uses write-back
- The miss penalty of the cache using write-through policy is constant.

A. 0  
B. 1  
C. 2  
D. 3  
E. 4
What happens on a write? (No-Write Allocate)

- **Write hit?**
  - Update in-place
  - Write to lower memory (Write-Through only)
    - write penalty (can be eliminated if there is a buffer)

- **Write miss?**
  - Write to the first lower memory hierarchy has the data
    - Penalty
What happens on a read?

- **Read hit**
  - hit time
- **Read miss?**
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
    - Data with the same “block address” will be fetch
  - Miss penalty

![Diagram showing cache hierarchy with CPU, L1 $, L2 $, and processes for read hit and read miss.](image)
Eviction in Associative caches

• We must choose which line in a set to evict if we have associativity
• How we make the choice is called the cache eviction policy
  • Random -- always a choice worth considering.
  • Least recently used (LRU) -- evict the line that was last used the longest time ago.
  • Prefer clean -- try to evict clean lines to avoid the write back.
  • Farthest future use -- evict the line whose next access is farthest in the future. This is provably optimal. It is also impossible to implement.
The Cost of Associativity

• Increased associativity requires multiple tag checks
  • N-Way associativity requires N parallel comparators
  • This is expensive in hardware and potentially slow.
• This limits associativity L1 caches to 2-8.
• Larger, slower caches can be more associative.
• Example: Nehalem
  • 8-way L1
  • 16-way L2 and L3.
• Core 2’s L2 was 24-way
Evaluating cache performance
How to evaluate cache performance

• If the load/store instruction hits in L1 cache where the hit time is usually the same as a CPU cycle
  • The CPI of this instruction is the base CPI

• If the load/store instruction misses in L1, we need to access L2
  • The CPI of this instruction needs to include the cycles of accessing L2

• If the load/store instruction misses in both L1 and L2, we need to go to lower memory hierarchy (L3 or DRAM)
  • The CPI of this instruction needs to include the cycles of accessing L2, L3, DRAM
How to evaluate cache performance

- **CPI\text{Average}**: the average CPI of a memory instruction

\[
\text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} + L_1\text{AccessTime} + \text{miss}\_\text{rate}_{L1} \times \text{miss}\_\text{penalty}_{L1}
\]

\[
\text{miss}\_\text{penalty}_{L1} = L_2\text{AccessTime} + \text{miss}\_\text{rate}_{L2} \times \text{miss}\_\text{penalty}_{L2}
\]

\[
\text{miss}\_\text{penalty}_{L2} = L_3\text{AccessTime} + \text{miss}\_\text{rate}_{L3} \times \text{miss}\_\text{penalty}_{L3}
\]

- **CPI\text{base}** = 1.
- If the problem (like those in your textbook) asks for average memory access time, transform the CPI values to/from time by multiplying/dividing by the cycle time!
Cache & Performance

- 5-stage MIPS processor.
  - Application: 80% ALU, 20% L/S
  - L1 I-cache miss rate: 5%, hit time: 1 cycle
  - L1 D-cache miss rate: 10%, hit time: 1 cycle
  - L2 U-Cache miss rate: 20%, hit time: 10 cycles
  - Main memory access time: 100 cycles
  - What’s the average CPI?

A. 0.75
B. 1.35
C. 1.75
D. 1.80
E. none of the above

\[ CPI_{average} = CPI_{base} + \text{miss-rate} \times \text{miss-penalty} \]

\[ CPI_{average} = 1 + 0.05 \times 10 + 0.2 \times 100 \]

\[ CPI_{average} = 1 + 0.2 \times 100 \]

\[ CPI_{average} = 1 + 20 \]

\[ CPI_{average} = 21 \]

\[ CPI_{average} = 3.1 \]
The End
Basic Problems in Caching

• A cache holds a small fraction of all the cache lines, yet the cache itself may be quite large (i.e., it might contain thousands of lines).
• Where do we look for our data?
• How do we tell if we’ve found it and whether it’s any good?
The Cache Line

- Caches operate on “lines”
- Caches lines are a power of 2 in size
  - They contain multiple words of memory.
- Usually between 16 and 128 bytes
Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits: 10
- Tag bits: 17
- Offset bits: 5
Practice

• 32KB cache.
• 64byte lines.

• Index  9
• Offset  6
• Tag     17
Reading from a cache

- Determine where in the cache, the data could be
- If the data is there (i.e., is it hit?), return it
- Otherwise (a miss)
  - Retrieve the data from the lower down the cache hierarchy.
  - Choose a line to evict to make room for the new line
    - Is it dirty? Write it back.
    - Otherwise, just replace it, and return the value
- The choice of which line to evict depends on the “Replacement policy”
Hit or Miss?

• Use the index to determine where in the cache, the data might be
• Read the tag at that location, and compare it to the tag bits in the requested address
• If they match (and the data is valid), it’s a hit
• Otherwise, a miss.
On a Miss: Making Room

• We need space in the cache to hold the data we want to access.
• We will need to evict the cache line at this index.
  • If it’s dirty, we need to write it back
  • Otherwise (it’s clean), we can just overwrite it.
Writing To the Cache (simple version)

- Determine where in the cache, the data could be
- If the data is there (i.e., is it hit?), update it
- Possibly forward the request down the hierarchy
- Otherwise
  - Retrieve the data from the lower down the cache hierarchy (why?)
  - **Option 1:** choose a line to evict  
    - Is it dirty? Write it back.
    - Otherwise, just replace it, and update it.
  - **Option 2:** Forward the write request down the hierarchy

Write allocation policy
Write Through vs. Write Back

• When we perform a write, should we just update this cache, or should we also forward the write to the next lower cache?
• If we do not forward the write, the cache is “Write back”, since the data must be written back when it’s evicted (i.e., the line can be dirty)
• If we do forward the write, the cache is “write through.” In this case, a cache line is never dirty.
• Write back advantages
  Fewer writes farther down the hierarchy. Less bandwidth. Faster writes
• Write through advantages
  No write back required on eviction.
Write Allocate/No-write allocate

- On a write miss, we don’t actually need the data, we can just forward the write request.
- If the cache allocates cache lines on a write miss, it is *write allocate*, otherwise, it is *no write allocate*.
- Write Allocate advantages

  Exploits temporal locality. Data written will likely be read soon, and that read will be faster.

- No-write allocate advantages

  Fewer spurious evictions. If the data is not read in the near future, the eviction is a waste.
# Associativity

<table>
<thead>
<tr>
<th>dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Set 0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Set 1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Set 2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Set 3</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Way 0

Way 1
New Cache Geometry Calculations

- Addresses break down into: tag, index, and offset.
- How they break down depends on the “cache geometry”

- Cache lines = L
- Cache line size = B
- Address length = A (32 bits in our case)
- Associativity = W

- Index bits = \( \log_2(L/W) \)
- Offset bits = \( \log_2(B) \)
- Tag bits = A - (index bits + offset bits)
Practice

• 32KB, 2048 Lines, 4-way associative.

• Line size: 16B
• Sets: 512
• Index bits: 9
• Tag bits: 19
• Offset bits: 4