

Caching

Today

- Quiz
- Design choices in cache architecture

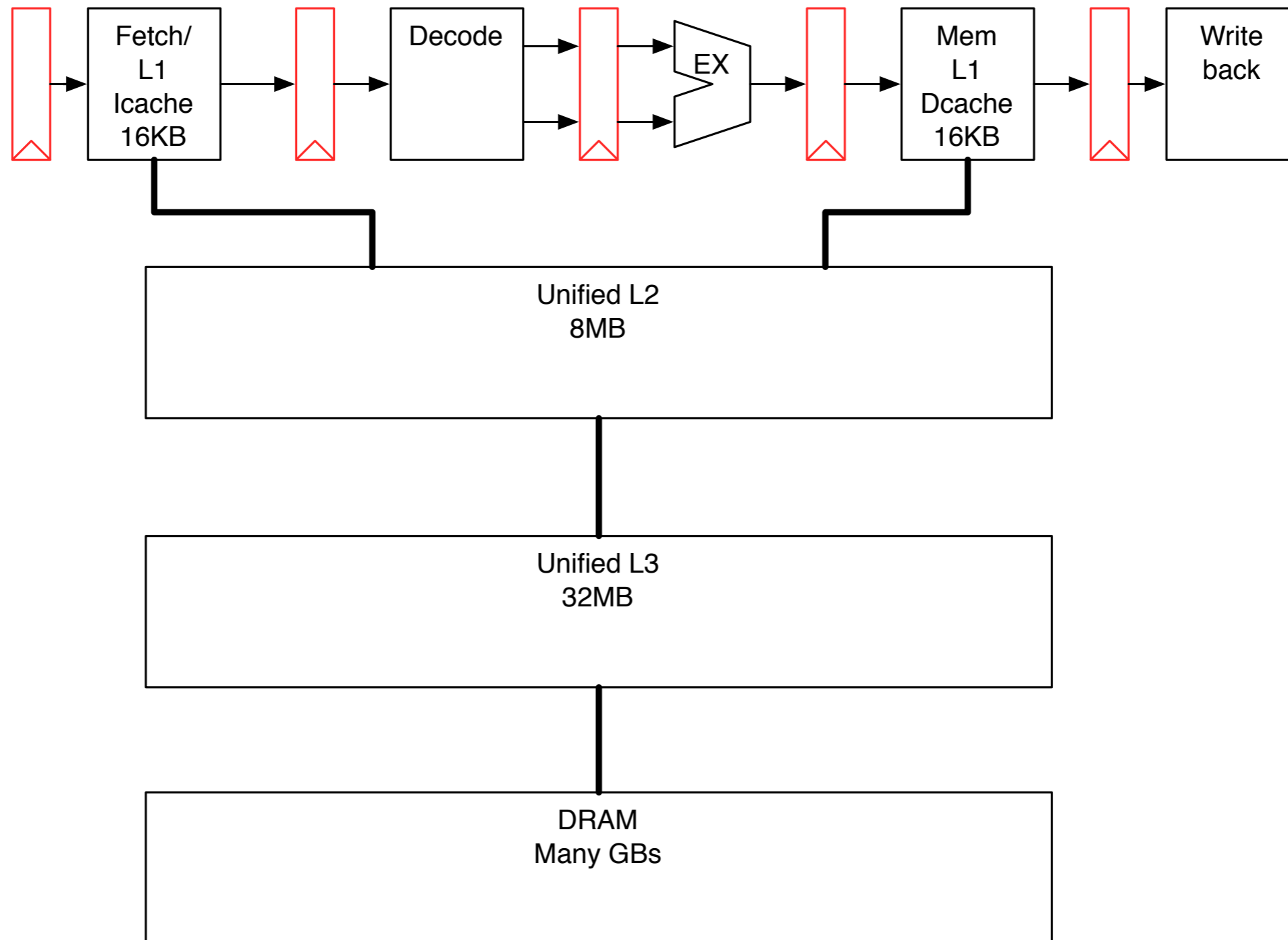
Key Point

- 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



Data vs Instruction Caches

- Why have different I and D caches?

Data vs Instruction Caches

- Why have different I and D caches?
 - Different areas of memory
 - Different access patterns
 - I-cache accesses have lots of spatial locality. Mostly sequential accesses.
 - I-cache accesses are also predictable to the extent that branches are predictable
 - D-cache accesses are typically less predictable
 - Not just different, but often across purposes.
 - Sequential I-cache accesses may interfere with the data the D-cache has collected.
 - This is “interference” just as we saw with branch predictors
 - At the LI level it avoids a structural hazard in the pipeline
 - Writes to the I cache by the program are rare enough that they can be prohibited (i.e., self modifying code)

The Cache Line

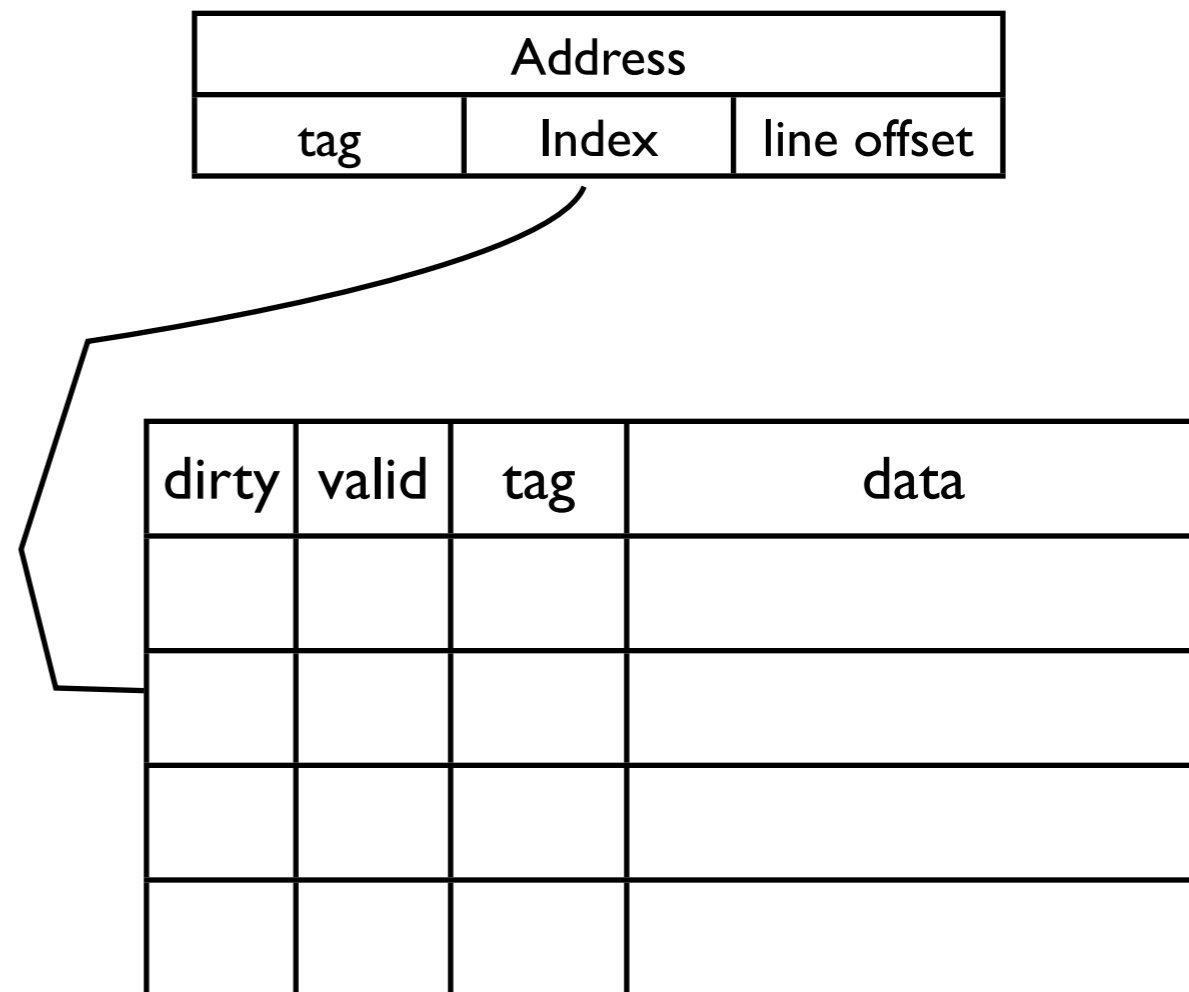
- Caches operate on “lines”
- Caches lines are a power of 2 in size
 - They contain multiple words of memory.
- Usually between 16 and 128 bytes
- The bit-width (i.e., 32 or 64 bits) does directly effect the cache configuration.
 - In fact almost all aspects of a cache and independent of the big-A architecture.
 - Caches are completely transparent to the processor.

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 1000s of lines)
- Where do we look for our data?
- How do we tell if we've found it and whether it's any good?

Basic Cache Organization

- Anatomy of a cache line entry
 - Dirty bit -- does this data match what is in memory
 - Valid -- does this mean anything at all?
 - Tag -- The high order bits of the address
 - Data -- The program's data
- Anatomy of an address
 - Index -- bits that determine the lines possible location
 - offset -- which byte within the line (low-order bits)
 - tag -- everything else (the high-order bits)
- Note that the index of the line, combined with the tag, uniquely identify one cache line's worth of memory



Cache line size

- How big should a cache line be?
- Why is bigger better?
- Why is smaller better?

Cache line size

- How big should a cache line be?
- Why is bigger better?
 - Exploits more spatial locality.
 - Large cache lines effectively *prefetch* data that we have not explicitly asked for.
- Why is smaller better?
 - Focuses on temporal locality.
 - If there is little spatial locality, large cache lines waste space and bandwidth.
 - More space devoted to tags.
- In practice 32-64 bytes is good for L1 caches where space is scarce and latency is important.
- Lower levels use 128-256 bytes.

2D Array

```
long long int array[10][10];  
  
int sum(int x, int count)  
{  
    int s = 0;  
    long long int i;  
    for(i = 0; i < count; i++) {  
        s+= array[x][i];  
    }  
    return s;  
}
```



Lots of spatial locality.

2D Array #2

```
nestLoop2.c
long long int array[5][5];
int sum(int x, int count)
{
    int s = 0;
    long long int i;
    for(i = 0; i < count; i++) {
        s+= array[i][x];
    }
    return s;
}
```



Little spatial locality.
(Temporal locality if we
execute this loop again)

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)
- Index bits = $\log_2(L)$
- Offset bits = $\log_2(B)$
- Tag bits = $A - (\text{index bits} + \text{offset bits})$

Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits:
- Tag bits:
- off set bits:

Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits: 10
- Tag bits:
- off set bits:

Practice

- 1024 cache lines. 32 Bytes per line.
- Index bits: 10
- Tag bits:
- off set bits: 5

Practice

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

Practice

- 32KB cache.
- 64byte lines.

- Index
- Offset
- Tag

Practice

- 32KB cache.
- 64byte lines.

- Index 9
- Offset
- Tag

Practice

- 32KB cache.
- 64byte lines.

- Index 9
- Offset
- Tag 17

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.
 - Is there a cache line available for the new data?
 - If so, fill the the line, and return the value
 - Otherwise choose a line to evict
 - Is it dirty? Write it back.
 - Otherwise, just replace it, and return the value

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.
 - Is there a cache line available for the new data?
 - If so, fill the the line, and return the value
 - Otherwise choose a line to evict <-- Replacement policy
 - Is it dirty? Write it back.
 - Otherwise, just replace it, and return the value

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: Finding Room

- We need space in the cache to hold the data that is missing
- The cache entry at the required index might be invalid, if so, great! there is space.
- Otherwise, we 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?)
 - Is there a cache line available for the new data?
 - If so, fill the the line, and update it
 - Otherwise choose a line to evict
 - Is it dirty? Write it back.
 - Otherwise, just replace it, and update it.
 - Alternate Otherwise
 - Forward the write request down the hierarchy

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?)
 - Is there a cache line available for the new data?
 - If so, fill the the line, and update it
 - Otherwise choose a line to evict <-- Replacement policy
 - Is it dirty? Write it back.
 - Otherwise, just replace it, and update it.
 - Alternate Otherwise
 - Forward the write request down the hierarchy

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?)
 - Is there a cache line available for the new data?
 - If so, fill the the line, and update it
 - Otherwise choose a line to evict **<-- Replacement policy**
 - Is it dirty? Write it back.
 - Otherwise, just replace it, and update it.
 - Alternate Otherwise **<-- Write allocation policy**
 - Forward the write request down the hierarchy

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 <-- Write back policy
- Otherwise
 - Retrieve the data from the lower down the cache hierarchy (why?)
 - Is there a cache line available for the new data?
 - If so, fill the the line, and update it
 - Otherwise choose a line to evict <-- Replacement policy
 - Is it dirty? Write it back.
 - Otherwise, just replace it, and update it.
 - Alternate Otherwise <-- Write allocation policy
 - Forward the write request down the hierarchy

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
- Write through advantages

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
- Write through advantages

No more write backs. Reads might be faster.

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 more write backs. Reads might be faster.

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
- No-write allocate advantages
 - Fewer spurious evictions. If the data is not read for a long time, the eviction is a waste.

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 for a long time, the eviction is a waste.

Write Allocate Caveat

- Write allocate caches must fill the cache line before they can complete the store, since the line contains multiple words.
- If they do not do the fill first, only the portion of the line that was actually written to would be valid.

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];
    }
}
```

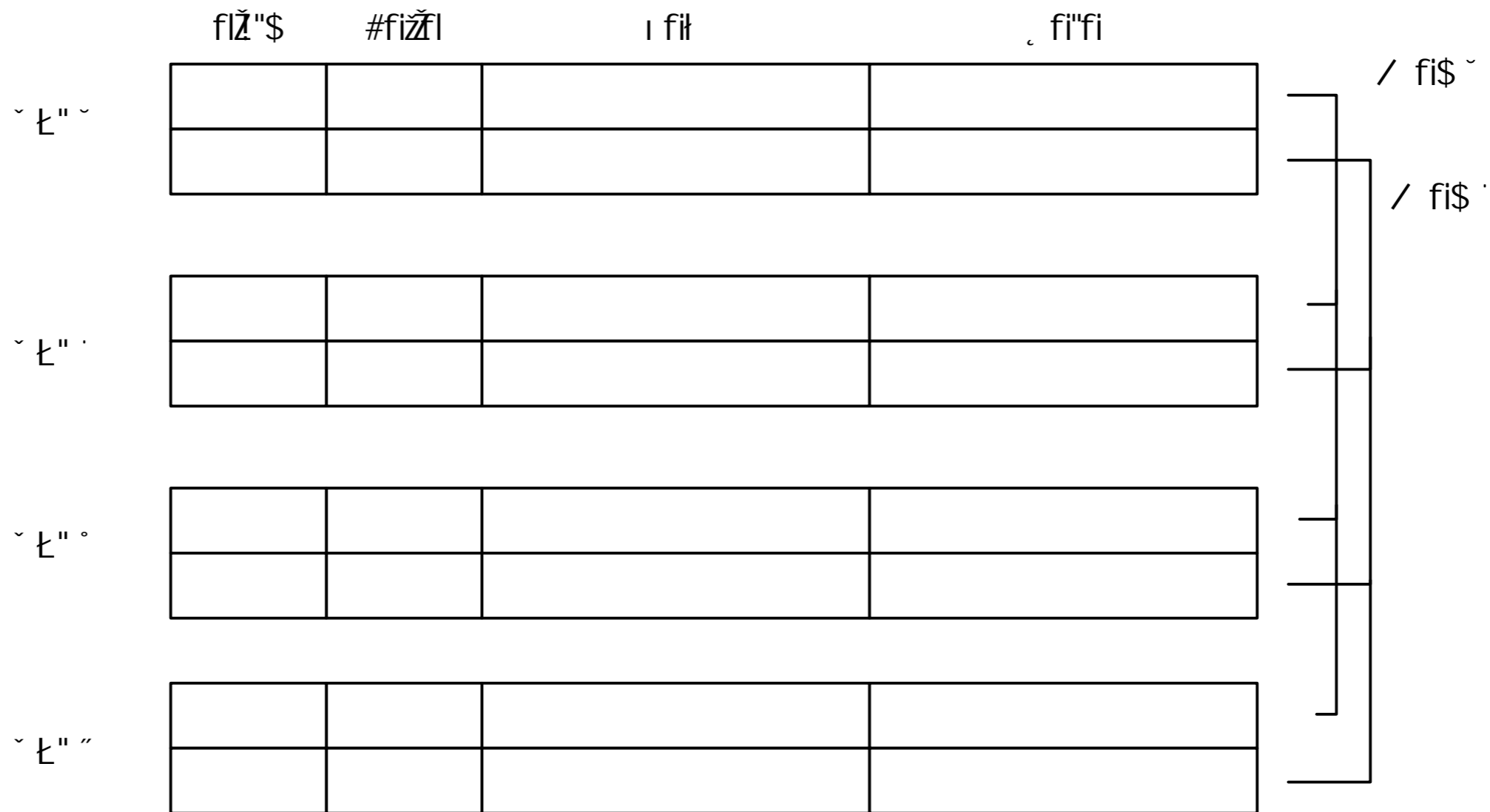
0x000	foo
...	
0x400	bar

- 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”

Associativity



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 = $A - (\text{index bits} + \text{offset bits})$

Practice

- 32KB, 2048 Lines, 4-way associative.
- Line size:
- Sets:
- Index bits:
- Tag bits:
- Offset bits:

Practice

- 32KB, 2048 Lines, 4-way associative.
- Line size: 16B
- Sets:
- Index bits:
- Tag bits:
- Offset bits:

Practice

- 32KB, 2048 Lines, 4-way associative.
- Line size: 16B
- Sets: 512
- Index bits:
- Tag bits:
- Offset bits:

Practice

- 32KB, 2048 Lines, 4-way associative.
- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits:
- Offset bits:

Practice

- 32KB, 2048 Lines, 4-way associative.
- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits:
- Offset bits: 4

Practice

- 32KB, 2048 Lines, 4-way associative.
- Line size: 16B
- Sets: 512
- Index bits: 9
- Tag bits: 19
- Offset bits: 4

Full Associativity

- In the limit, a cache can have one, large set.
- The cache is then fully associative
- A one-way associative cache is also called --
direct mapped

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. Hard to implement true randomness.
 - 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.
 - The fastest way is to use a “content addressable memory” They embed comparators in the memory array. -- try instantiating one in Xilinx.
- This limits associativity L1 caches to 2-4. In L2s to make 16 way.