

CS 31: Introduction to Computer Systems

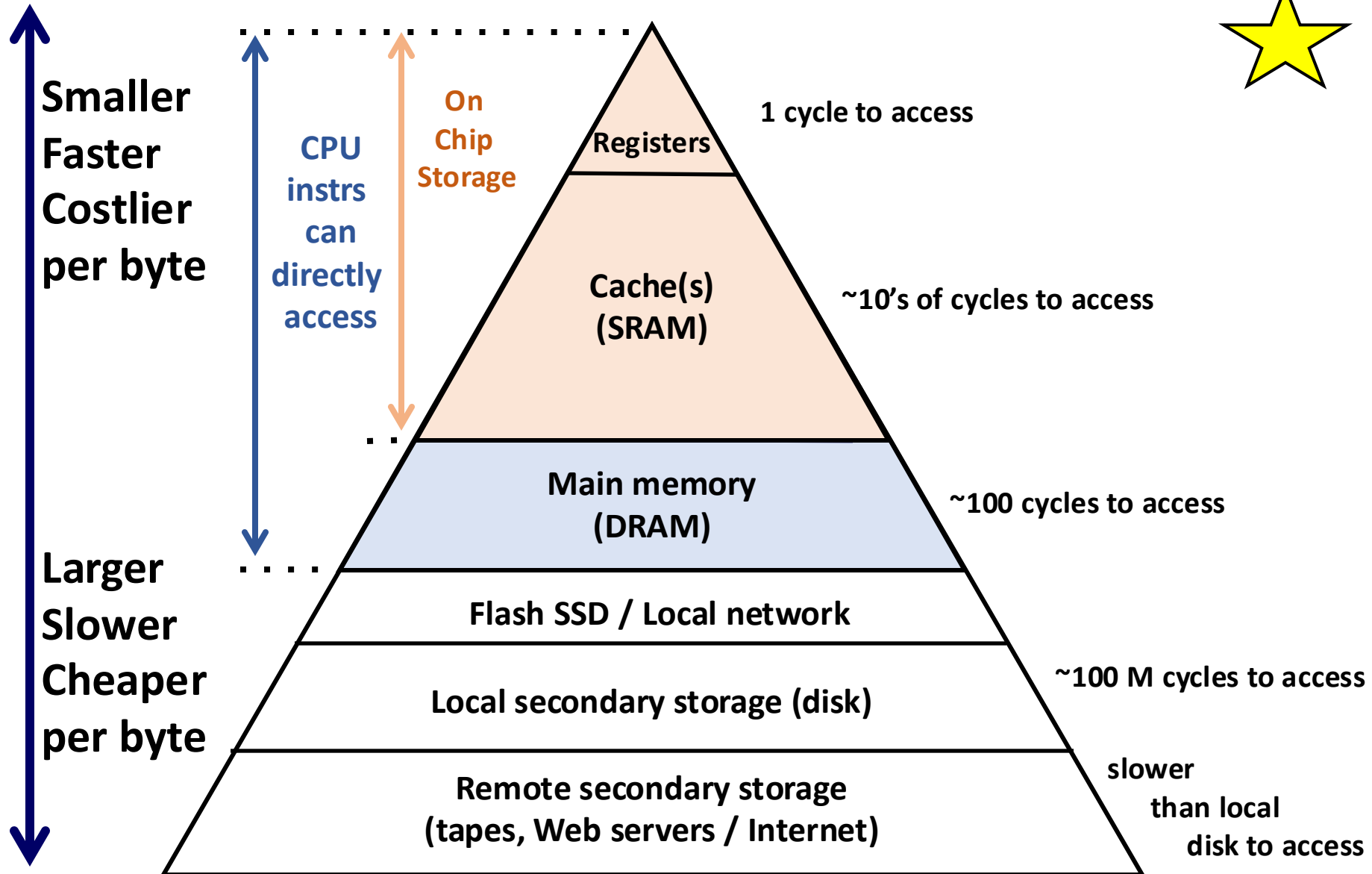
1.6 Storage and Locality

03-25-2025



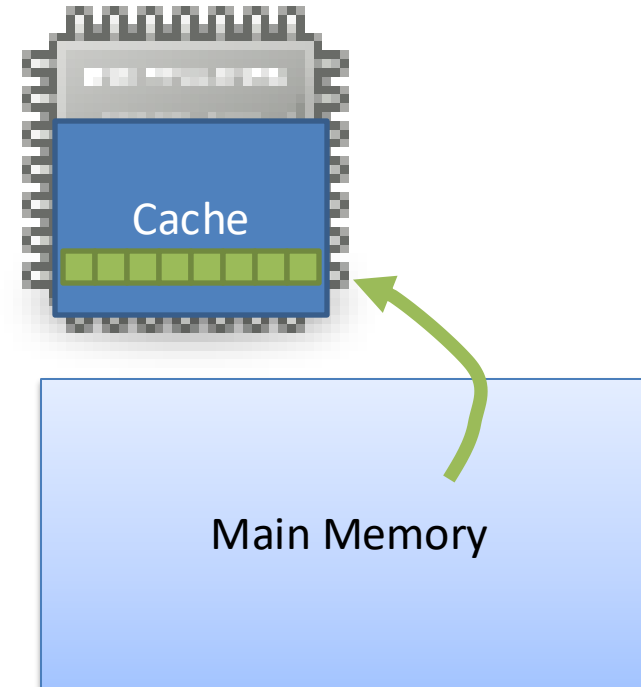
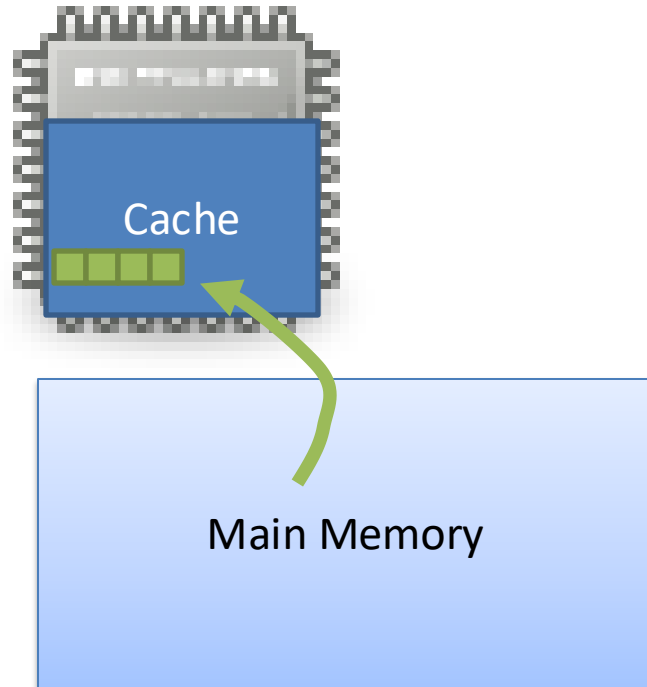
Reading Quiz

Last class: The Memory Hierarchy



Cache Design

- Lot's of characteristics to consider:
 - Where should data be stored in the cache?
 - What size data chunks should we store? (block size)



System Cache Management

The **Mechanism** of cache misses:

- How to detect a miss?
- How to fetch and load data from lower level of the memory hierarchy?
- Do we need to replace some current item's data in the cache with new data?

The **Policy** of cache miss:

- *Which data* to kick out of the cache to make room for the missed data?

System Cache Management

Separation of Policy and Mechanism

policy implementation

mechanism implementation

- Tune the mechanism for the very specific hardware
- Policy interacts with mechanism interface
- + Faster mechanisms
- + Easier to change policies

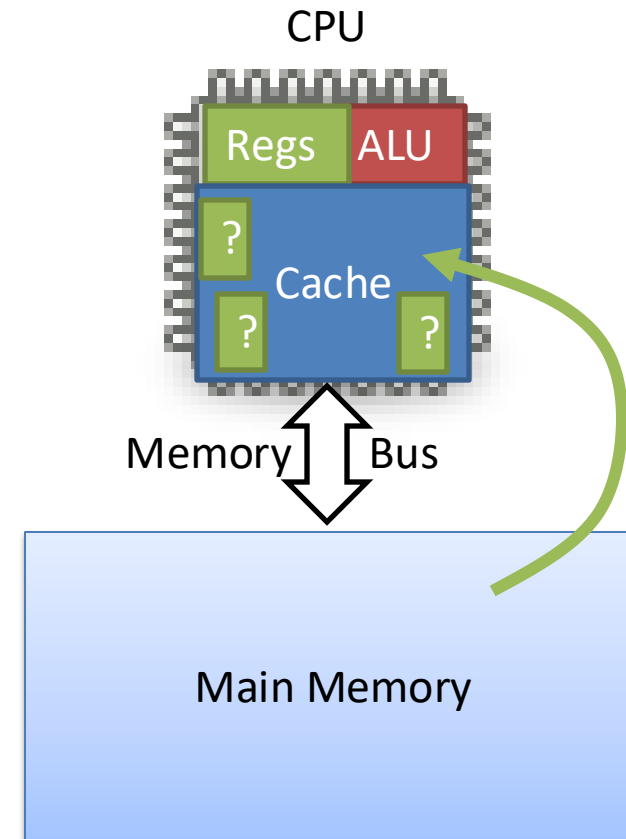
Cache Design

- Lot's of characteristics to consider:
 - Where should data be stored in the cache?
 - What size data chunks should we store? (block size)
- **Goals:**
 - Maximize hit rate
 - Maximize (temporal & spatial) locality benefits
 - Reduce cost/complexity of design

Suppose the CPU asks for data, it's not in cache.

We need to move in into cache from memory. Where in the cache should it be allowed to go?

- A. In exactly one place.
- B. In a few places.
- C. In most places, but not all.
- D. Anywhere in the cache.



A. Direct-Mapped: In exactly one place

- Every location in memory is directly mapped to one place in the cache.
- Easy to find data.

B. Set-Associative: In a few places.

- A memory location can be mapped to (2, 4, 8) locations in the cache.
- Middle ground.

~~C. In most places, but not all.~~

D. “Fully associative”: Anywhere in the cache.

- No restrictions on where memory can be placed in the cache.
- Fewer conflict misses, more searching.

What should we use to determine whether or not data is in the cache?

- A. The memory address of the data.
- B. The value of the data.
- C. The size of the data.
- D. Some other aspect of the data.

What should we use to determine whether or not data is in the cache?

A. The memory address of the data.

– Memory address is how we identify the data.

B. The value of the data.

– If we knew this, we wouldn't be looking for it!

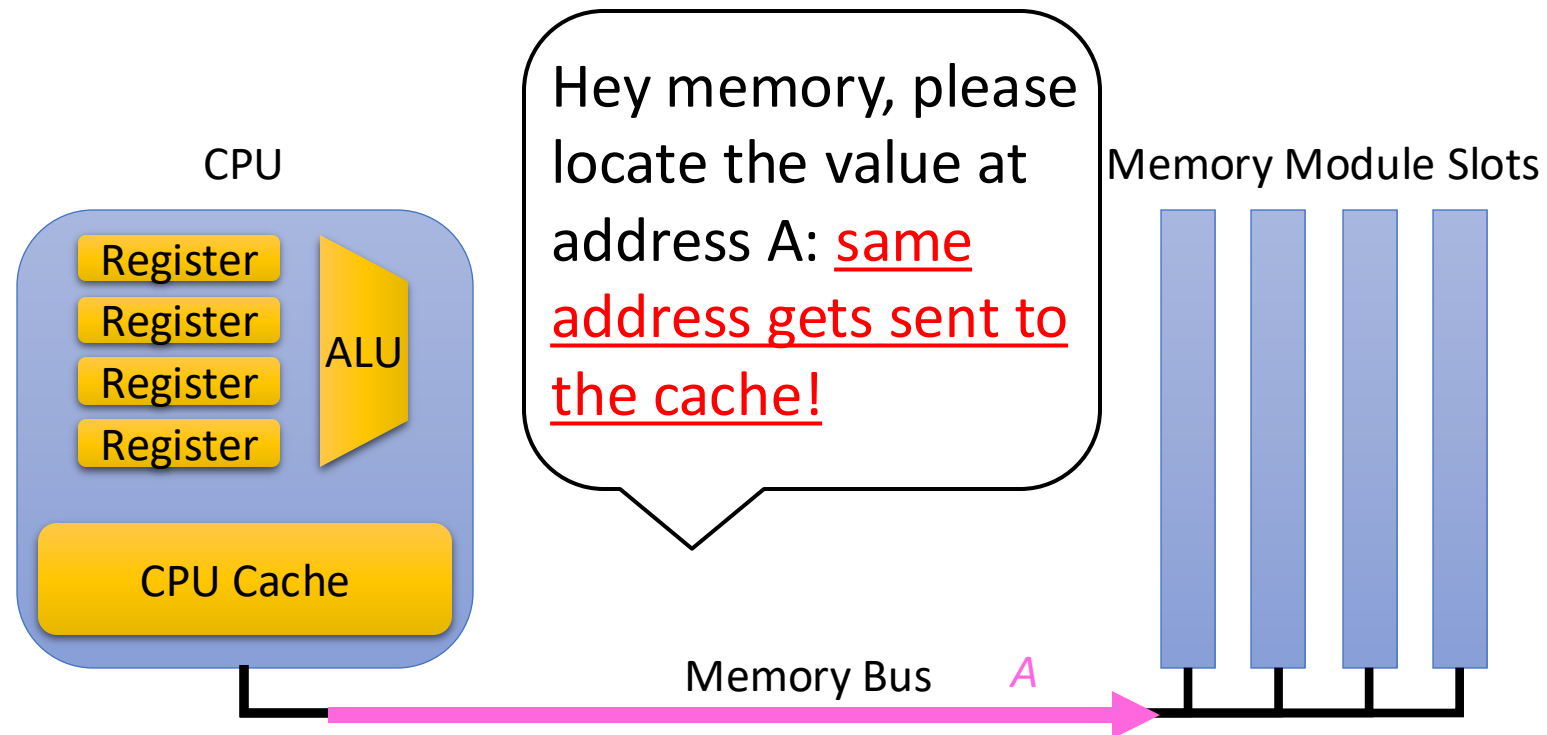
C. The size of the data.

D. Some other aspect of the data.

Recall: Memory Reads

CPU places address A on the memory bus.

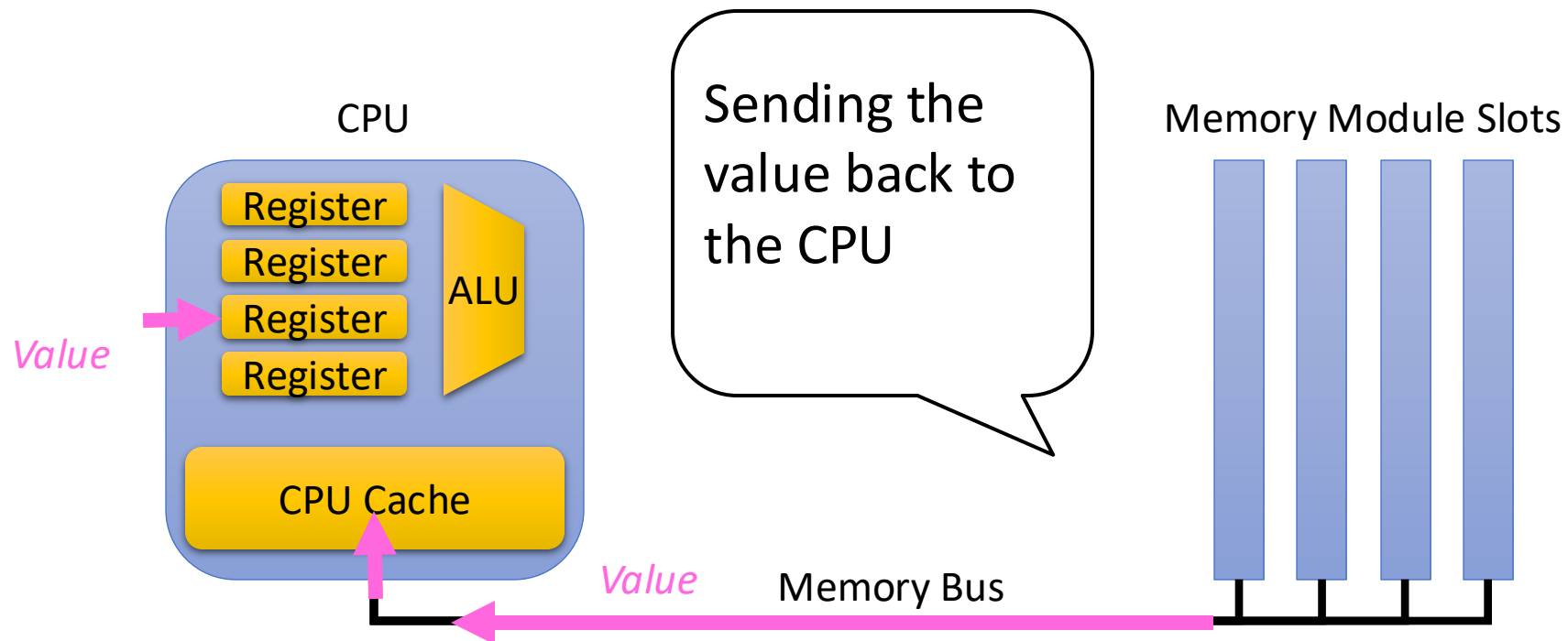
Load operation: `movl (A), %eax`



Recall: Memory Reads

Memory retrieves value and sends it across bus.

CPU reads value from the bus, and copies it into register %eax, a copy also goes into the on-chip cache memory.



Memory Address Tells Us...

- Is the block containing the byte(s) you want already in the cache?
- If not, where should we put that block?
 - Do we need to kick out (“evict”) another block?
- Which byte(s) within the block do you want?

Memory Addresses for use with Cache

- Like everything else: series of bits (x86_64 has 64 bit addresses)
- Keep in mind:
 - N bits gives us 2^N unique values.
- 64-bit address:
 - 1011000101110010110101000101011010110001011100101101010001010110



Divide into regions, each with distinct meaning.

Address Division

- **First section: Tag**
 - Of all the addresses that map to this location, which one is here?
 - Number of bits for this section is any bits left over after index and offset.
- **Second section: Index**
 - Which location(s) in the cache should we check for the data with this address?
 - Number of bits for this section depends on the number of cache locations.
- **Third section: Offset**
 - If we find a block of bytes in the cache (on a hit) which byte offset within the block do we actually want?
 - Number of bits for this section depends on the block size – must be able to uniquely identify every byte in the block.

Address Division

1011000101110010110101000101011010110001011100101101010001010110

| | | |
|--------------|----------------|----------------------|
| Tag (X bits) | Index (Y bits) | Byte offset (Z bits) |
|--------------|----------------|----------------------|

- First section: **Tag**
 - Of all the addresses that map to this location, which one is here?
 - Uniquely identify the subset of memory contained within a cache line.
 - Number of bits for this section is any bits left over **after** index and offset.
- Second section: **Index**
 - Which location(s) in the cache should we check for the data with this address?
 - Number of bits for this section depends on the number of cache locations.
- Third section: **Offset**
 - If we find a block of bytes in the cache (on a hit) which *bytes within the block* do we actually want?
 - Number of bits for this section depends on the block size – must be able to uniquely identify every byte in the block.

A. In exactly one place. (“Direct-mapped”)

- **Every location in memory is directly mapped to one place in the cache. Easy to find data.**

B. In a few places. (“Set associative”)

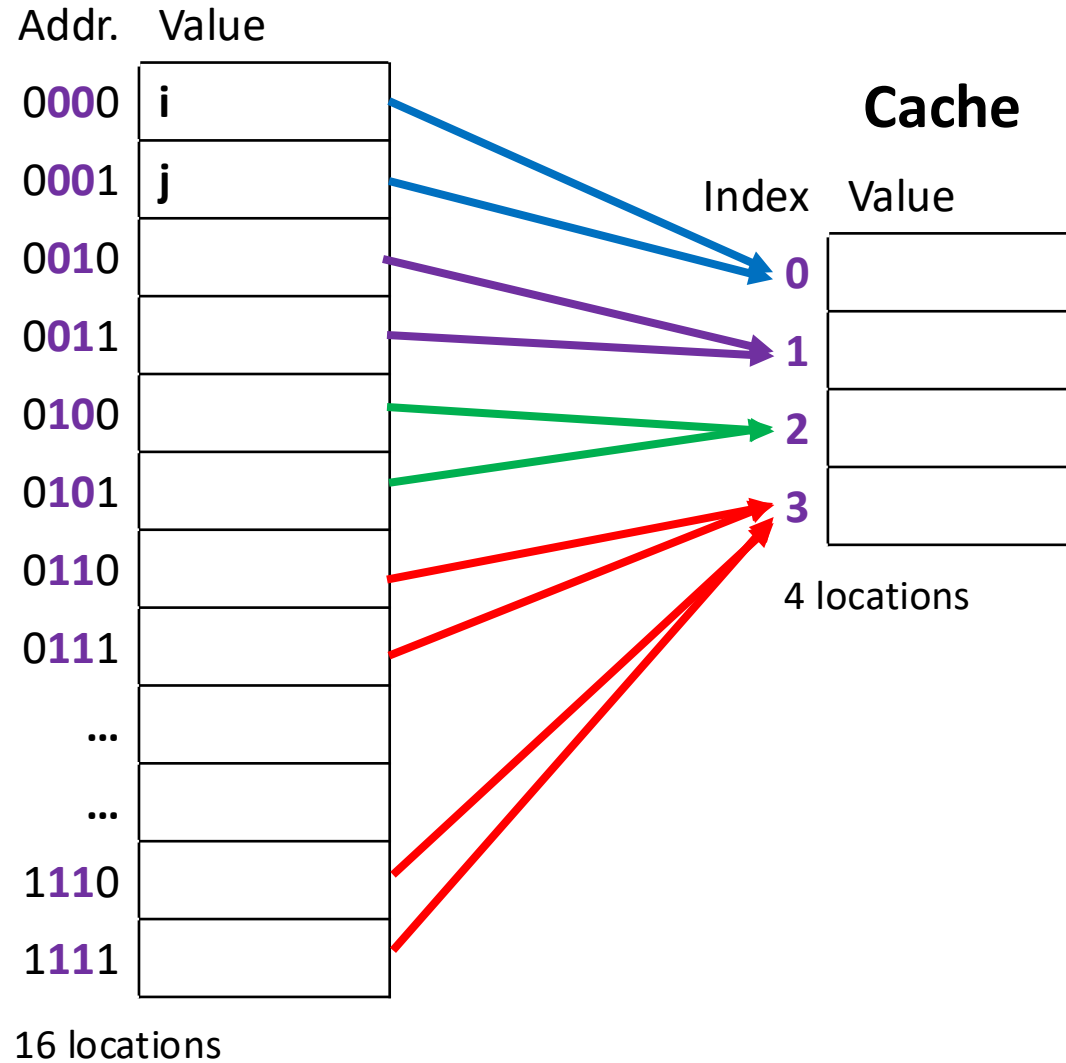
- A memory location can be mapped to (2, 4, 8) locations in the cache. Middle ground.

A. Anywhere in the cache. (“Fully associative”)

- No restrictions on where memory can be placed in the cache. Fewer conflict misses, more searching.

Direct Mapped Cache

Main Memory



i and j map to the same cache line
and may constantly evict each other!


Direct-Mapped

- One place data can be.
- Example: let's assume some parameters:
 - 1024 cache locations (every block mapped to one)
 - Block size of 8 bytes

Direct-Mapped

1024 cache locations (every block mapped to one)
Block size of 8 bytes

Metadata



| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|-----|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Cache meta-data

Metadata

Valid bit: is the entry valid?

If set: data is correct, use it if we 'hit' in cache

If not set: ignore 'hits', the data is garbage

Dirty bit: has the data been written?

Used by write-back caches

If set, need to update memory before eviction

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|-----|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Address division: Direct-Mapped

- Identify byte in block
 - How many bits do we need to represent each byte uniquely?
- Identify which row (line)
 - How many bits do we need to represent each line uniquely?

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|-----|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

- A. Block 8 bits Row 1024 bits B. Block 3 bits Row 10 bits
C. Block 10 bits Row 10 bits D. Block 32 bits Row 32 bits

Address division: Direct-Mapped

- Identify byte in block
 - How many bits? 3
- Identify which row (line)
 - How many bits? 10
- Tag:
 - 64 - 13: 51 bits

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|-----|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Direct-Mapped

Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| | | |

Index:

Which line (row) should we check?

Where could data be?

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|-----|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Direct-Mapped

Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| | | |



Index:

Which line (row) should we check?

Where could data be?

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|-----|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Direct-Mapped

Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| 4217 | 4 | |



In parallel, check:

Tag:

Does the cache hold the data we're looking for, or some other block?

Valid bit:

If entry is not valid, **don't trust garbage in that line (row).**

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | 1 | | 4217 | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

If tag doesn't match, or line is invalid, it's a miss!

Direct-Mapped

Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| 4217 | 4 | |

Byte offset tells us which subset of block to retrieve.

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | 1 | | 4217 | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|

Direct-Mapped

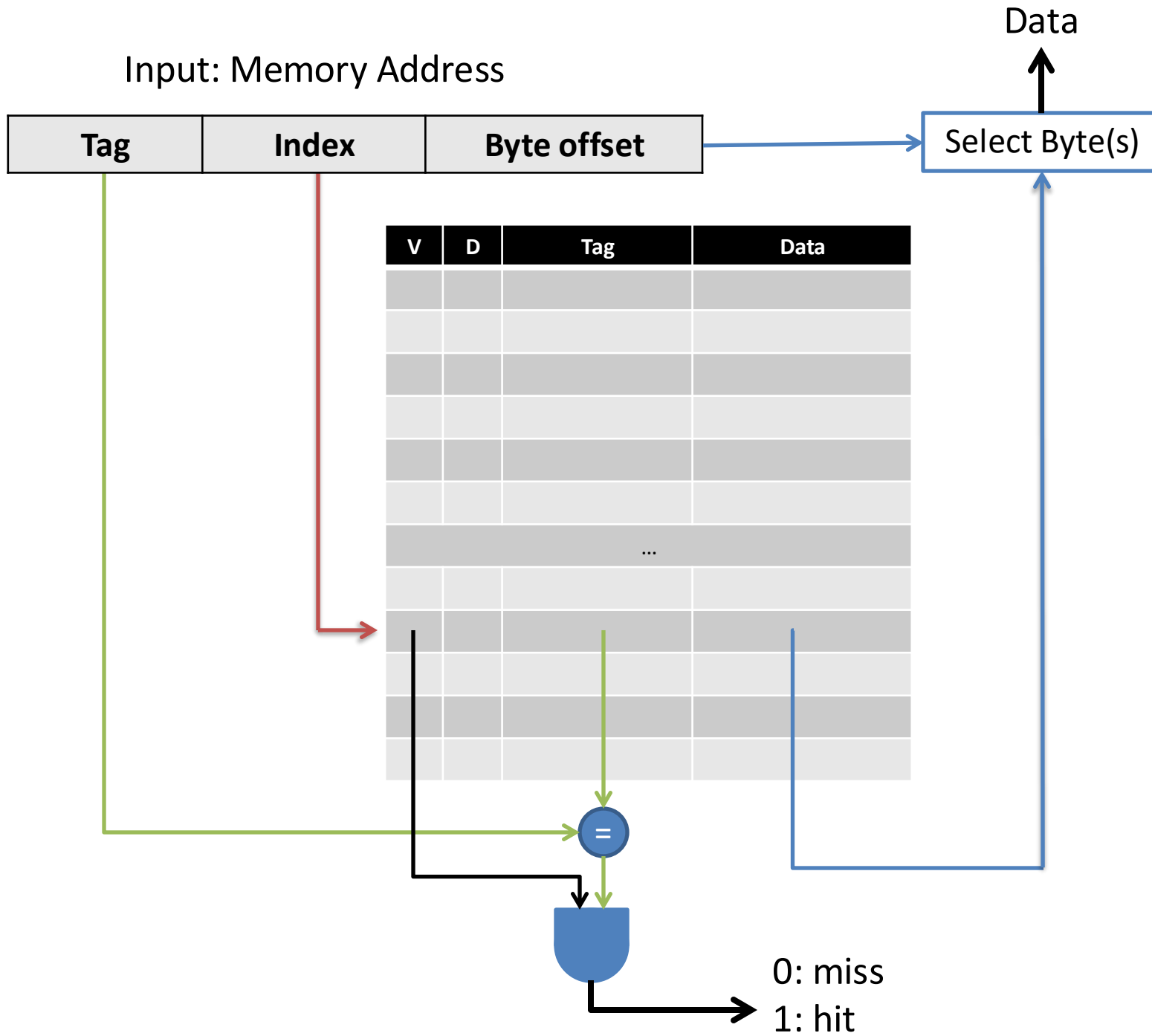
Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| 4217 | 4 | 4 |

Byte offset tells us which subset of block to retrieve.

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | 1 | | 4217 | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|



Direct-Mapped Example

- Suppose our addresses are 16 bits long.
- Our cache has 16 entries, block size of 16 bytes
 - 4 bits in address for the index
 - 4 bits in address for byte offset
 - Remaining bits (8): tag

Direct-Mapped Example

- Let's say we access memory at address:
 - 0110101100110100
- Step 1:
 - Partition address into tag, index, offset

| Line | V | D | Tag | Data (16 Bytes) |
|------|---|---|-----|--------------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| ... | | | | |
| 15 | | | | |

Direct-Mapped Example

- Let's say we access memory at address:
 - 01101011 0011 0100
- Step 1:
 - Partition address into tag, index, offset

| Line | V | D | Tag | Data (16 Bytes) |
|------|---|---|-----|--------------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| ... | | | | |
| 15 | | | | |

Direct-Mapped Example

- Let's say we access memory at address:
 - 01101011 0011 0100
- Step 2:
 - Use index to find line (row)
 - 0011 -> 3

| Line | V | D | Tag | Data (16 Bytes) |
|------|---|---|-----|--------------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| ... | | | | |
| 15 | | | | |

Direct-Mapped Example

- Let's say we access memory at address:

– 01101011 0011 0100

- Step 2:

- Use index to find line (row)
- 0011 -> 3

| Line | V | D | Tag | Data (16 Bytes) |
|------|---|---|-----|-----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| 5 | | | | |
| ... | | | | |
| 15 | | | | |

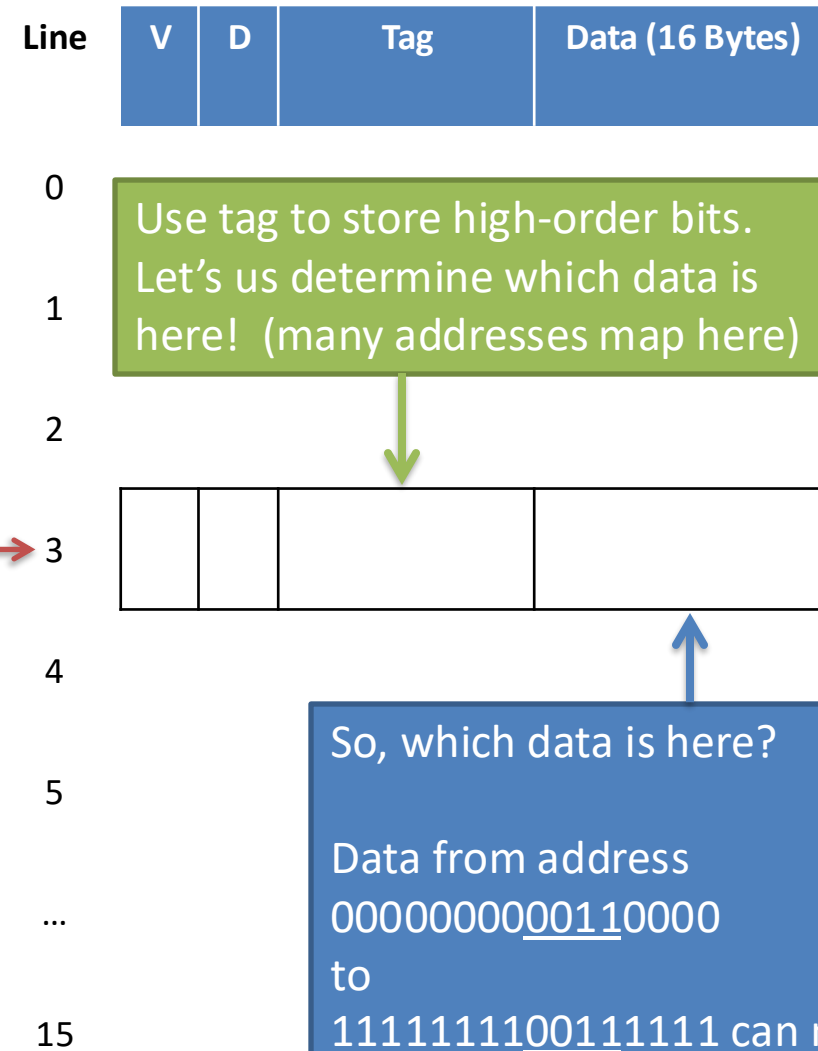
Direct-Mapped Example

- Let's say we access memory at address:

– 01101011 0011 0100

- Note:

- ANY address with 0011 (3) as the middle four index bits will map to this cache line.
- e.g. 11111111 0011 0000



Direct-Mapped Example

- Let's say we access memory at address:

– 01101011 0011 0100

- Step 3:

- Check the tag
- Is it 01101011 (hit)?
- Something else (miss)?
- (Must also ensure valid)

| Line | V | D | Tag | Data (16 Bytes) |
|------|---|---|----------|-----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | 01101011 | |
| 4 | | | | |
| 5 | | | | |
| ... | | | | |
| 15 | | | | |

Eviction

- If we don't find what we're looking for (miss), we need to bring in the data from memory.
- Make room by kicking something out.
 - If line to be evicted is dirty, write it to memory first.
- Another important systems distinction:
 - Mechanism: An ability or feature of the system. What you can do.
 - Policy: Governs the decisions making for using the mechanism. What you should do.

Eviction

- For direct-mapped cache:
 - Mechanism: overwrite bits in cache line, **updating**
 - Valid bit
 - Tag
 - Data
 - Policy: not many options for direct-mapped
 - Overwrite at the only location it could be!

Eviction: Direct-Mapped

- Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| 3941 | 1020 | |

Find line:

Tag doesn't match, bring in from memory.

If dirty, write back first!

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | 1 | 0 | 1323 | 57883 |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Eviction: Direct-Mapped

- Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| 3941 | 1020 | |

1. Send address to read main memory.



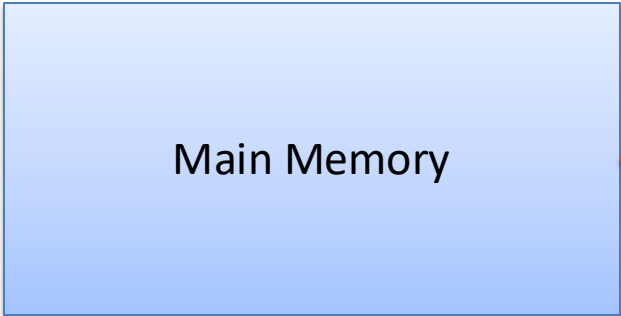
| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | 1 | 0 | 1323 | 57883 |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Eviction: Direct-Mapped

- Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| 3941 | 1020 | |

1. Send address to read main memory.



| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | | | | |
| ... | | | ... | |
| 1020 | 1 | 0 | 3941 | 92 |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

2. Copy data from memory. Update tag.

Direct-Mapped

- Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| 4217 | 4 | 2 |

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | 1 | | 4217 | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Byte offset tells us which subset of block to retrieve.

Can one read of a variable straddle multiple cache blocks?



Direct-Mapped

- Address division:

| Tag (51 bits) | Index (10 bits) | Byte offset (3 bits) |
|---------------|-----------------|----------------------|
| 4217 | 4 | 2 |

| Line | V | D | Tag | Data (8 Bytes) |
|------|---|---|------|----------------|
| 0 | | | | |
| 1 | | | | |
| 2 | | | | |
| 3 | | | | |
| 4 | 1 | | 4217 | |
| ... | | | ... | |
| 1020 | | | | |
| 1021 | | | | |
| 1022 | | | | |
| 1023 | | | | |

Byte offset tells us which subset of block to retrieve.

Can one read of a variable straddle multiple cache blocks?

No, recall mem. alignment!



Direct-Mapped Example

- Suppose our addresses are 16 bits long.
- Our cache has 16 entries, block size of 16 bytes
 - 4 bits in address for the index
 - 4 bits in address for byte offset
 - Remaining bits (8): tag

Suppose we had 8-bit addresses, a cache with 8 lines, and a block size of 4 bytes.

- How many bits would we use for:
 - Tag?
 - Index?
 - Offset?

Suppose a system has 8-bit addresses, a DM cache with 8 lines, and 4-byte block size

- How many bits would be used for:
 - **Byte-offset: 2 bits**
 - 4 byte block size, can address each byte in 2 bits (00, 01, 10, 11)
 - **Index: 3 bits**
 - With 8 lines, need 3 bits to encode each cache line number
 - **Tag: 3 bits**
 - Bits left over in the address after byte-offset and index ($8 - 2 - 3$)

- Which bits would be used for:
 - Tag? **high order**
 - Index? **middle** (right after byte offset bits)
 - Byte offset? **low order**

ex. **01010011**

How would the cache change if we performed the following memory operations?

Memory address



Read 01000100 (Value: 5)
Read 11100010 (Value: 17)
Write 01110000 (Value: 7)
Read 10101010 (Value: 12)
Write 01101100 (Value: 2)

| Line | V | D | Tag | Data (4 Bytes) |
|------|---|---|-----|-------------------|
| 0 | 1 | 0 | 111 | 17 |
| 1 | 1 | 0 | 011 | 9 |
| 2 | 0 | 0 | 101 | 15 |
| 3 | 1 | 1 | 001 | 8 |
| 4 | 1 | 0 | 011 | 4 |
| 5 | 0 | 0 | 111 | 6 |
| 6 | 0 | 0 | 101 | 32 |
| 7 | 1 | 0 | 110 | 3 |

WS #4: How would the cache change if we performed the following memory operations?

Memory address



Tag: 3
Index: 3
Offset: 2

→ Read 01000100 (Value: 5) **MISS**

Read 11100010 (Value: 17)

Read 10111101 (Value: 2)

Write 01111100 (Value 10)

Write 01110000 (Value: 7)

Read 10101010 (Value: 12)

Write 01101100 (Value: 2)



| Line | V | D | Tag | Data (4 Bytes) |
|------|---|---|--------------------|----------------|
| 0 | 1 | 0 | 111 | 17 |
| 1 | 1 | 0 | 011 010 | 9 5 |
| 2 | 0 | 0 | 101 | 15 |
| 3 | 1 | 1 | 001 | 8 |
| 4 | 1 | 0 | 011 | 4 |
| 5 | 0 | 0 | 111 | 6 |
| 6 | 0 | 0 | 101 | 32 |
| | 1 | 0 | 110 | 3 |

At line 1, V=1 but tags don't match, so we have a MISS.

Dirty bit is 0, so we can safely overwrite it.

Write: V = 1; D = 0 (we're reading, not writing); tag, data (value)

WS #4: How would the cache change if we performed the following memory operations?

Memory address

Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) MISS

→ Read 11100010 (Value: 17) HIT

Read 10111101 (Value: 2)

Write 01111100 (Value 10)

Write 01110000 (Value: 7)

Read 10101010 (Value: 12)

Write 01101100 (Value: 2)

| Line | V | D | Tag | Data (4 Bytes) |
|------|---|---|--------------------|----------------|
| → 0 | 1 | 0 | 111 | 17 |
| 1 | 1 | 0 | 011 010 | 9 5 |
| 2 | 0 | 0 | 101 | 15 |
| 3 | 1 | 1 | 001 | 8 |
| 4 | 1 | 0 | 011 | 4 |
| 5 | 0 | 0 | 111 | 6 |
| 6 | 0 | 0 | 101 | 32 |
| | 1 | 0 | 110 | 3 |

At line 1, V=1 and tags match, so we have a HIT.
What we wanted in cache is there, so we saved time!

WS #4: How would the cache change if we performed the following memory operations?

Memory address

Tag: 3
Index: 3
Offset: 2

Read 01000100 (Value: 5) MISS

Read 11100010 (Value: 17) HIT

→ Read 10111101 (Value: 2) MISS

Write 01111100 (Value 10)

Write 01110000 (Value: 7)

Read 10101010 (Value: 12)

Write 01101100 (Value: 2)

| Line | V | D | Tag | Data (4 Bytes) |
|------|---|---|--------------------|----------------|
| 0 | 1 | 0 | 111 | 17 |
| 1 | 1 | 0 | 011 010 | 9 5 |
| 2 | 0 | 0 | 101 | 15 |
| 3 | 1 | 1 | 001 | 8 |
| 4 | 1 | 0 | 011 | 4 |
| 5 | 0 | 0 | 111 | 6 |
| 6 | 0 | 0 | 101 | 32 |
| 7 | 0 | 0 | 110 101 | 3 2 |

At line 7, V=0 and tags don't match, so we have a MISS.
Dirty bit is 0, so we can safely overwrite it.
Write: V = 1; tag, data (value)

WS #4: How would the cache change if we performed the following memory operations?

Memory address

Tag: 3
Index: 3
Offset: 2

- Read 01000100 (Value: 5) MISS
- Read 11100010 (Value: 17) HIT
- Read 10111101 (Value: 2) MISS
- Write 01111100 (Value 10) MISS
- Write 01110000 (Value: 7)
- Read 10101010 (Value: 12)
- Write 01101100 (Value: 2)

| Line | V | D | Tag | Data (4 Bytes) |
|------|----------------|----------------|--------------------------------------|----------------|
| 0 | 1 | 0 | 111 | 17 |
| 1 | 1 | 0 | 011 010 | 9 5 |
| 2 | 0 | 0 | 101 | 15 |
| 3 | 1 | 1 | 001 | 8 |
| 4 | 1 | 0 | 011 | 4 |
| 5 | 0 | 0 | 111 | 6 |
| 6 | 0 | 0 | 101 | 32 |
| 7 | 0 1 | 0 1 | 110 101 011 | 3 2 10 |

At line 7, V=1 but tags don't match, so we have a MISS. D=0, so we can safely overwrite it. Write: V = 1; D = 1 (we're updating cache but it is now out of sync with main memory); tag, data (value)

WS #4: How would the cache change if we performed the following memory operations?

Memory address

Tag: 3
Index: 3
Offset: 2

- Read 01000100 (Value: 5) MISS
- Read 11100010 (Value: 17) HIT
- Read 10111101 (Value: 2) MISS
- Write 01111100 (Value 10) MISS
- Write 01110000 (Value: 7) HIT
- Read 10101010 (Value: 12)
- Write 01101100 (Value: 2)

| Line | V | D | Tag | Data (4 Bytes) |
|------|-------------------|-------------------|--------------------------------------|----------------|
| 0 | 1 | 0 | 111 | 17 |
| 1 | 1 | 0 | 011 010 | 9 5 |
| 2 | 0 | 0 | 101 | 15 |
| 3 | 1 | 1 | 001 | 8 |
| → 4 | 1 | 0 | 011 | 4 7 |
| 5 | 0 | 0 | 111 | 6 |
| 6 | 0 | 0 | 101 | 32 |
| 7 | 0 1 | 0 1 | 110 101 011 | 3 2 10 |

At line 4, V=1 and tags match, so we have a HIT. D=0. Write value 7. Set data (value); D = 1 (we're updating cache and it is now out of synch with main memory)

WS #4: How would the cache change if we performed the following memory operations?

Memory address

Tag: 3
Index: 3
Offset: 2

- Read 01000100 (Value: 5) MISS
- Read 11100010 (Value: 17) HIT
- Read 10111101 (Value: 2) MISS
- Write 01111100 (Value 10) MISS
- Write 01110000 (Value: 7) HIT
- Read 10101010 (Value: 12) MISS
- Write 01101100 (Value: 2)



| Line | V | D | Tag | Data (4 Bytes) |
|------|--------------------------------|-------------------|--------------------------------------|------------------|
| 0 | 1 | 0 | 111 | 17 |
| 1 | 1 | 0 | 011 010 | 9 5 |
| 2 | 0 1 | 0 | 101 101 | 15 12 |
| 3 | 1 | 1 | 001 | 8 |
| 4 | 1 | 0 | 011 | 4 7 |
| 5 | 0 | 0 | 111 | 6 |
| 6 | 0 | 0 | 101 | 32 |
| 7 | 0 1 1 | 0 1 | 110 101 011 | 3 2 10 |

At line 2, tags match but V=0, so we have a MISS. D=0, so no need to evict anything. Read value in from memory and store it in cache. Set V=1, D=0; tag; data (value).

WS #4: How would the cache change if we performed the following memory operations?

Memory address

Tag: 3
Index: 3
Offset: 2

- Read 01000100 (Value: 5) MISS
- Read 11100010 (Value: 17) HIT
- Read 10111101 (Value: 2) MISS
- Write 01111100 (Value 10) MISS
- Write 01110000 (Value: 7) HIT
- Read 10101010 (Value: 12) MISS
- Write 01101100 (Value: 2) MISS

| Line | V | D | Tag | Data (4 Bytes) |
|------|--------------------------------|-------------------|--------------------------------------|------------------|
| 0 | 1 | 0 | 111 | 17 |
| 1 | 1 | 0 | 011 010 | 9 5 |
| 2 | 0 1 | 0 | 101 101 | 15 12 |
| → 3 | 1 1 | 1 1 | 001 011 | 8 2 |
| 4 | 1 | 0 | 011 | 4 7 |
| 5 | 0 | 0 | 111 | 6 |
| 6 | 0 | 0 | 101 | 32 |
| 7 | 0 1 1 | 0 1 | 110 101 011 | 3 2 10 |

At line 3, V=1 and tags don't match, so we have a MISS. D=1, so we need to save it to memory before we overwrite it. Set V=1, D=1; tag; data (value).

Associativity

- Problem: suppose we're only using a small amount of data (e.g., 8 bytes, 4-byte block size)
- Bad luck: (both) blocks map to same cache line
 - Constantly evicting one another
 - Rest of cache is going unused!
- Associativity: allow a set blocks to be stored at the same index. Goal: reduce conflict misses.

Comparison

Direct-mapped

- Tag tells you if you found the correct data.
- Offset specifies which byte within block.
- Middle bits (index) tell you which 1 line to check.

- (+) Low complexity, fast.
- (-) Conflict misses.

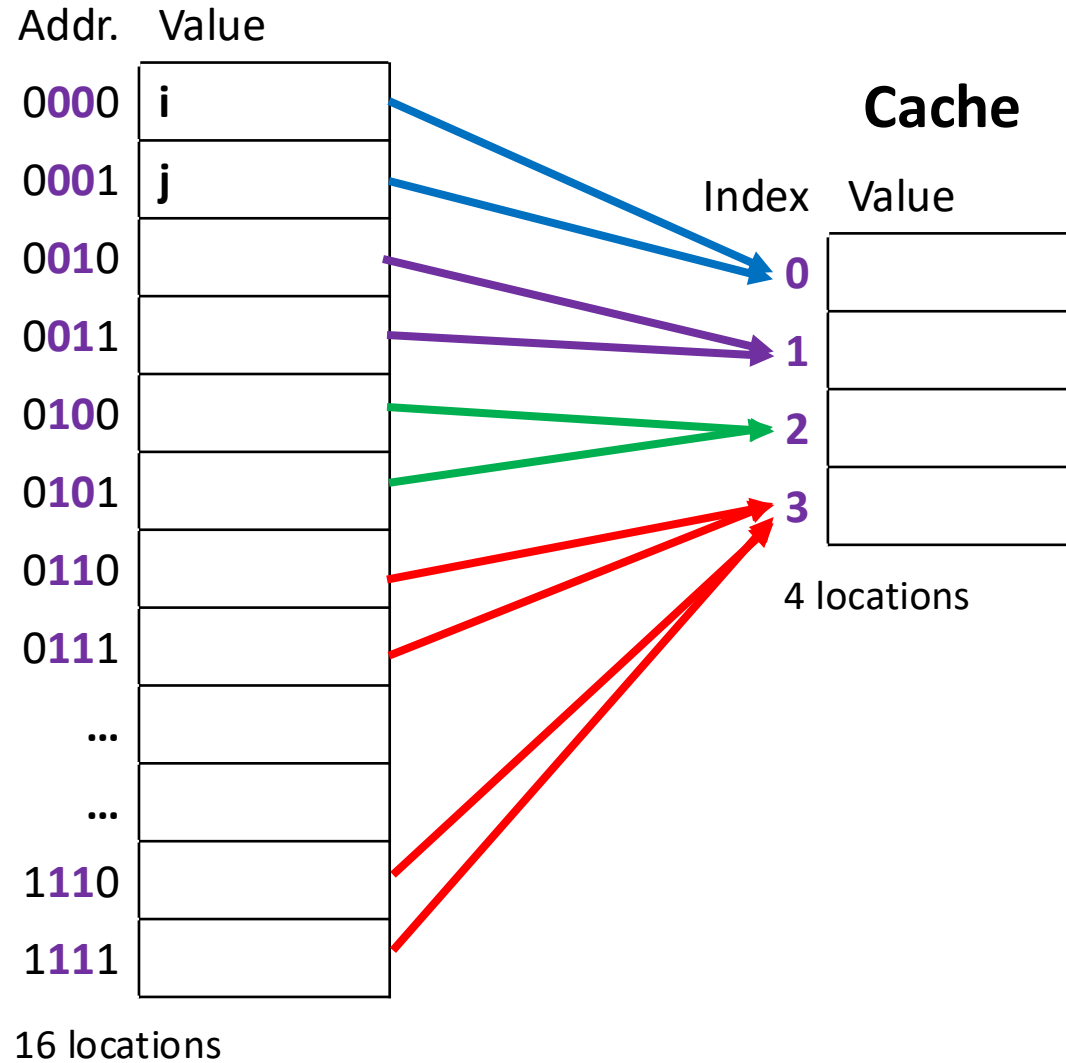
N-way set associative

- Tag tells you if you found the correct data.
- Offset specifies which byte within block.
- Middle bits (set) tell you which N lines to check.

- (+) Fewer conflict misses.
- (-) More complex, slower, consumes more power.

Direct Mapped Cache

Main Memory



i and j map to the same cache line
and may constantly evict each other!

Set Associative Cache

Main Memory

| Addr. | Value |
|-------|-------|
| 0000 | i |
| 0001 | j |
| 0010 | |
| 0011 | |
| 0100 | |
| 0101 | |
| 0110 | |
| 0111 | |
| ... | |
| ... | |
| 1110 | |
| 1111 | |

16 locations

Cache

| Index | Line 1 | Line 2 |
|-------|--------|--------|
| 0 | | |
| 1 | | |

2 sets, 4 locations

i and j map to the same cache line
but different locations in the cache.

2-Way Set Associative

| Tag (52 bits) | Set (9 bits) | Byte offset (3 bits) |
|---------------|--------------|----------------------|
| 3941 | 4 | |

| Set # | V | D | Tag | Data (8 Bytes) | V | D | Tag | Data (8 Bytes) |
|-------|---|---|------|----------------|---|---|------|----------------|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | 1 | 1 | 4063 | | 1 | 0 | 3941 | |
| ... | | | ... | | | | ... | |
| 508 | | | | | | | | |
| 509 | | | | | | | | |
| 510 | | | | | | | | |
| 511 | | | | | | | | |

Check all locations in the set, in parallel.

2-Way Set Associative

| Tag (52 bits) | Set (9 bits) | Byte offset (3 bits) |
|---------------|--------------|----------------------|
| 3941 | 4 | |

| Set # | V | D | Tag | Data (8 Bytes) | V | D | Tag | Data (8 Bytes) |
|-------|---|---|------|----------------|---|---|------|----------------|
| 0 | | | | | | | | |
| 1 | | | | | | | | |
| 2 | | | | | | | | |
| 3 | | | | | | | | |
| 4 | 1 | 1 | 4063 | | 1 | 0 | 3941 | |
| ... | | | ... | | | | ... | |
| 508 | | | | | | | | |
| 509 | | | | | | | | |
| 510 | | | | | | | | |
| 511 | | | | | | | | |

0 1 2 3 4 5 6 7

0 1 2 3 4 5 6 7

Multiplexer

Select correct value.

0 1 2 3 4 5 6 7

Eviction

- **Mechanism** is the same...
 - Overwrite bits in cache line: update tag, valid, data
- **Policy:** choose which line in the set to evict
 - Pick a random line in set
 - Choose an invalid line first
 - Choose the least recently used block
 - Has exhibited the least locality, kick it out!

} Common
combo in
practice.

Least Recently Used (LRU)

- **Intuition:** if it hasn't been used in a while, we have no reason to believe it will be used soon.
- Need extra state to keep track of LRU info: which line is least recently used -> left or right?

| Set # | LRU | V | D | Tag | Data (8 Bytes) | V | D | Tag | Data (8 Bytes) |
|-------|-----|---|---|------|----------------|---|---|------|----------------|
| 0 | 0 | | | | | | | | |
| 1 | 1 | | | | | | | | |
| 2 | 1 | | | | | | | | |
| 3 | 0 | | | | | | | | |
| 4 | 1 | 1 | 1 | 4063 | | 1 | 0 | 3941 | |
| ... | | | | ... | | | | ... | |

Least Recently Used (LRU)

- Intuition: if it hasn't been used in a while, we have no reason to believe it will be used soon.
- Need extra state to keep track of LRU info.
- For perfect LRU info:
 - 2-way: 1 bit
 - 4-way: 8 bits
 - N-way: $N * \log_2 N$ bits

Another reason why associativity often maxes out at 8 or 16.

These are metadata bits, not “useful” program data storage.

(Approximations make it not quite as bad.)

Suppose a system has 8-bit addresses, a two-way set associative cache with 8 lines, and 4-byte block size

- How many bits would we use for:
 - Tag?
 - Index?
 - Offset?

Suppose a system has 8-bit addresses, a **two-way set associative cache** with 8 lines, and 4-byte block size

- How many bits would we use for:
 - Tag? **3**
 - Set? **3**
 - Offset? **2**

Cache Conscious Programming

Knowing about caching and designing code around it can significantly effect performance

(ex) 2D array accesses

```
for(i=0; i < N; i++) {  
    for(j=0; j < M; j++) {  
        sum += arr[i][j];  
    }  
}
```

```
for(j=0; j < M; j++) {  
    for(i=0; i < N; i++) {  
        sum += arr[i][j];  
    }  
}
```

Algorithmically, both $O(N * M)$.

Is one faster than the other?

Cache Conscious Programming

Knowing about caching and designing code around it can significantly effect performance

(ex) 2D array accesses

```
for(i=0; i < N; i++) {  
  for(j=0; j < M; j++) {  
    sum += arr[i][j];  
  }  
}
```

A. is faster.

```
for(j=0; j < M; j++) {  
  for(i=0; i < N; i++) {  
    sum += arr[i][j];  
  }  
}
```

B. is faster.

Algorithmically, both $O(N * M)$.

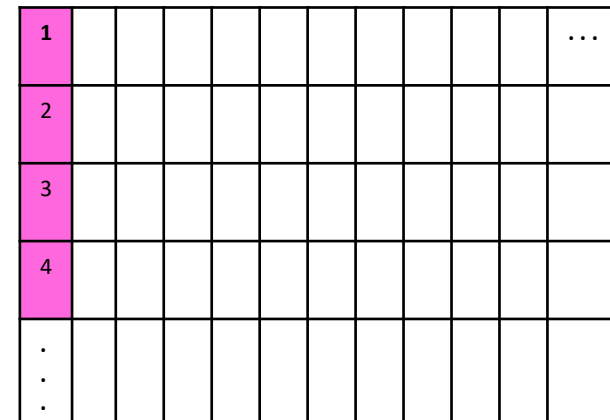
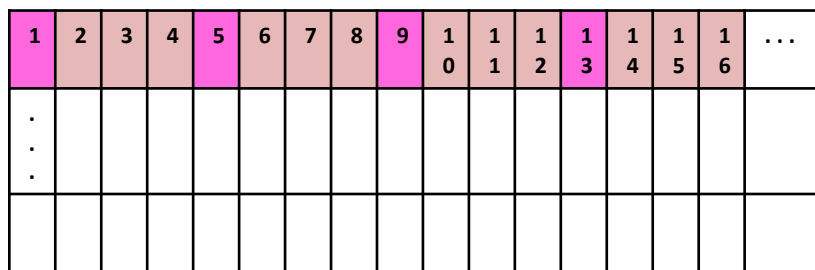
Is one faster than the other?

C. Both would exhibit roughly equal performance.

Cache Conscious Programming

The first nested loop is more efficient if the cache block size is larger than a single array bucket (for arrays of basic C types, it will be).

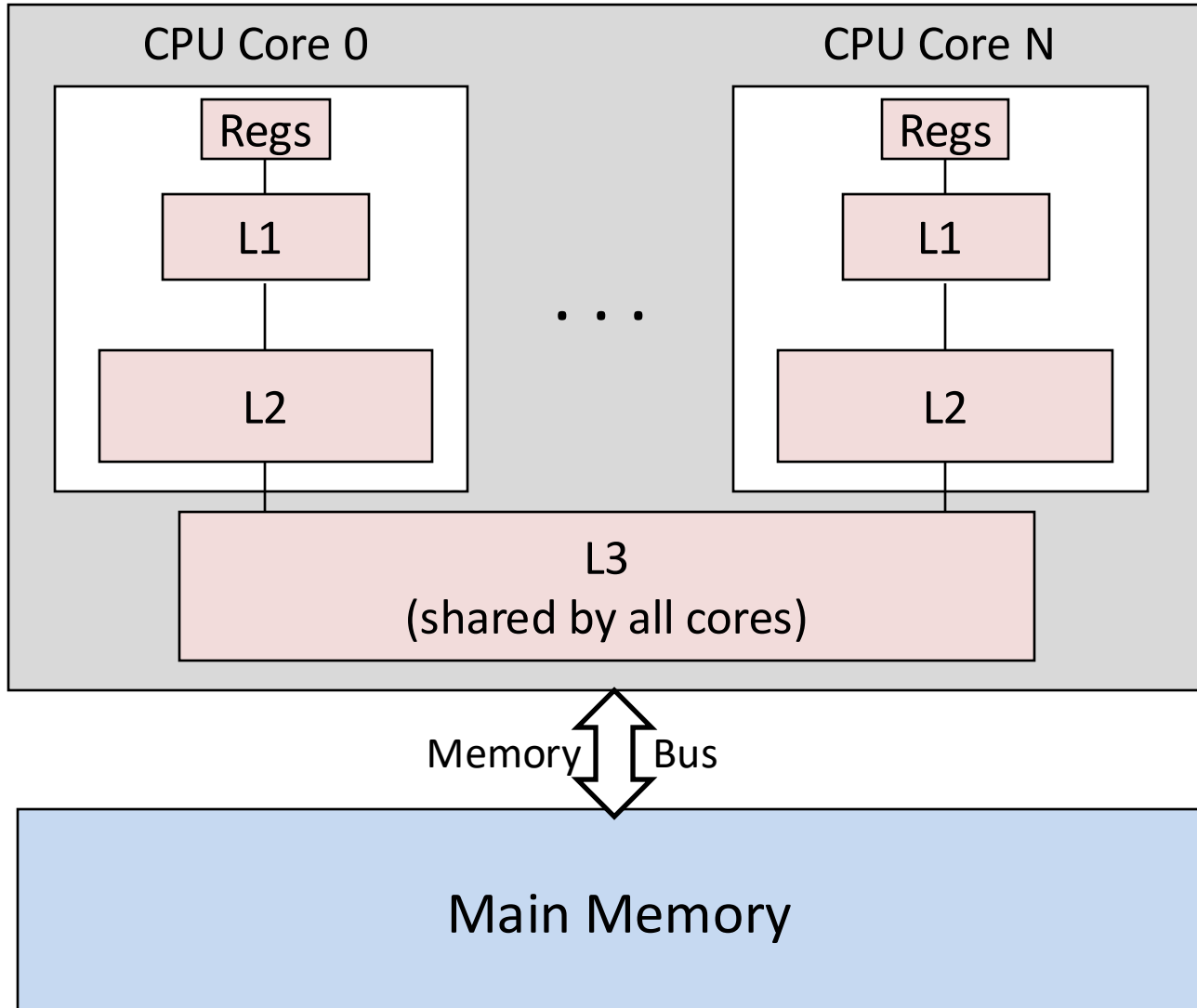
| | |
|--|--|
| <pre>for(i=0; i < N; i++) { for(j=0; j < M; j++) { sum += arr[i][j]; } }</pre> | <pre>for(j=0; j < M; j++) { for(i=0; i < N; i++) { sum += arr[i][j]; } }</pre> |
|--|--|



(ex) 1 miss every 4 buckets vs. 1 miss every bucket

FYI: Example Cache Organization

Multicore processor: multiple CPU cores/chip



Example Sizes

& Access Times

(KB: 2^{10} MB: 2^{20} GB: 2^{30})

L1: Size: 32 KB
Access: 4 cycles
write through
low associativity

L2: Size: 256 KB
Access: 10 cycles

L3: Size: 8 MB
Access: 40 cycles
write back
higher associativity

Main Memory (off chip):
Size: 16 GB
Access: 100 cycles

Program Efficiency and Memory

- Be aware of how your program accesses data
 - Sequentially, in strides of size X , randomly, ...
 - How data is laid out in memory
- Will allow you to structure your code to run much more efficiently based on how it accesses its data
- Don't go nuts...
 - Optimize the most important parts, ignore the rest
 - “Premature optimization is the root of all evil.” -Knuth

Amdahl's Law

Idea: an optimization can improve total runtime at most by the fraction it contributes to total runtime

If program takes 100 secs to run, *and you optimize a portion of the code that accounts for 2% of the runtime*, the best your optimization can do is improve the runtime by 2 secs!

Amdahl's Law tells us **to focus our optimization efforts on the code that matters:**

Speed-up what is accounting for the largest portion of runtime to get the largest benefit. And, don't waste time on the small stuff.