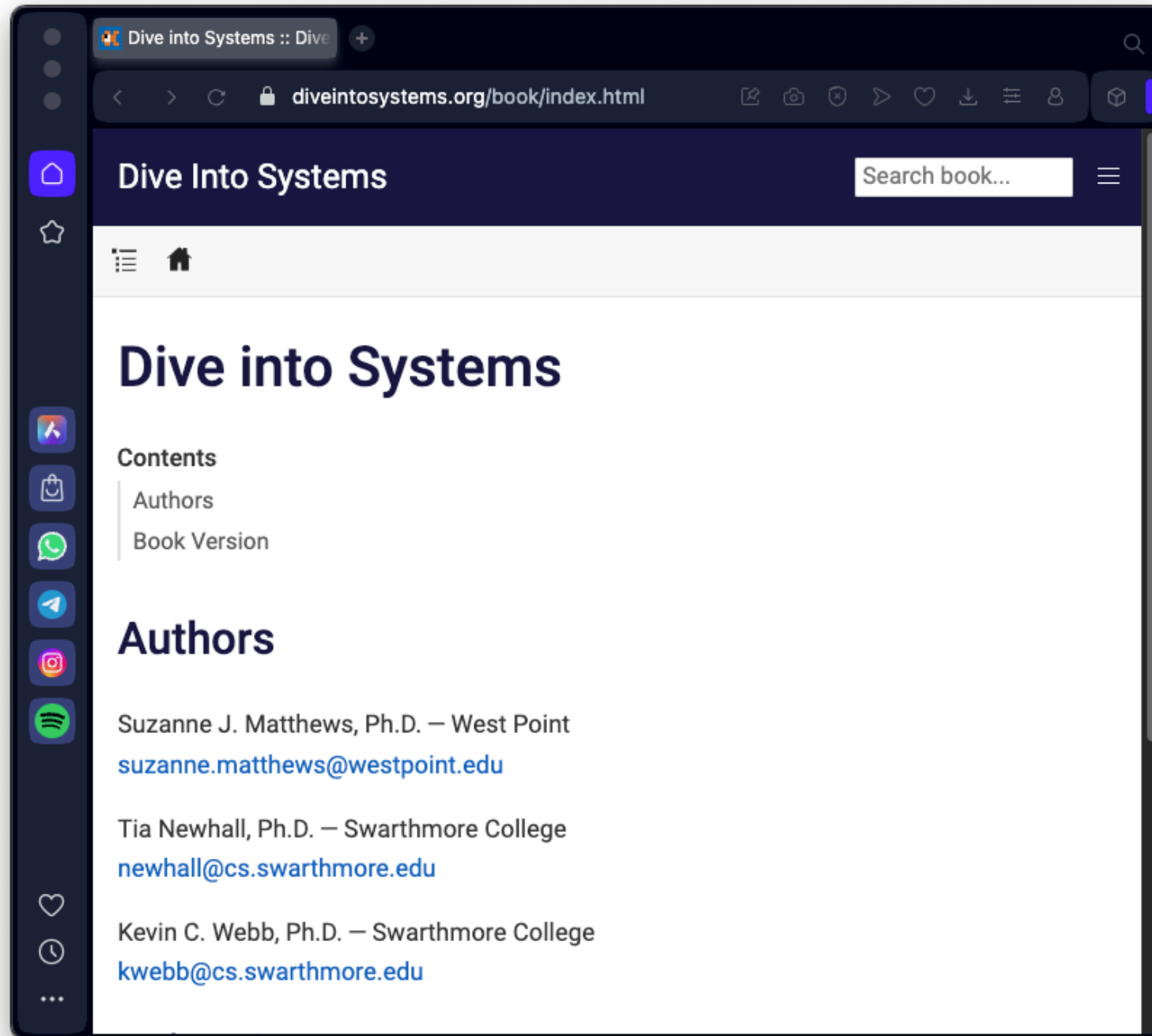


11. Storage and the Memory Hierarchy

For COMSC 142

Free online textbook



- <https://diveintosystems.org/book/index.html>

Topics

11.1. The Memory Hierarchy

11.2. Storage Devices

11.3. Locality

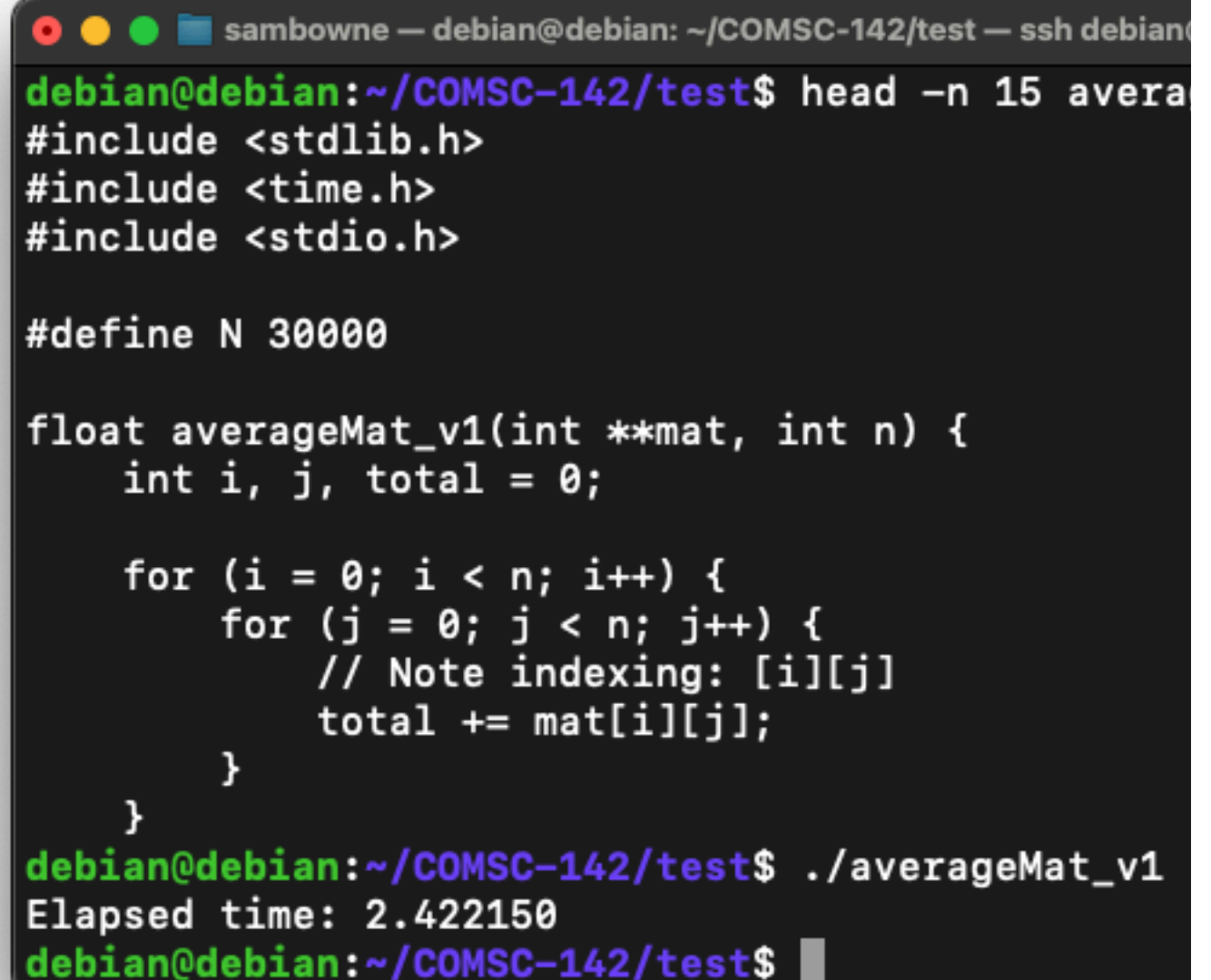
11.4. Caching

11.5. Cache Analysis and Cachegrind

11.6. Looking Ahead: Caching on Multicore Processors

Demonstration of Timing

- `wget https://samsclass.info/COMSC-142/proj/averageMat_v1.c`
- `gcc -o averageMat_v1 averageMat_v1.c`
- `head -n 15 averageMat_v1.c`
- `./averageMat_v1`



A terminal window titled "sambowne — debian@debian: ~/COMSC-142/test — ssh debian" displays the following content:

```
debian@debian:~/COMSC-142/test$ head -n 15 averageMat_v1.c
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

#define N 30000

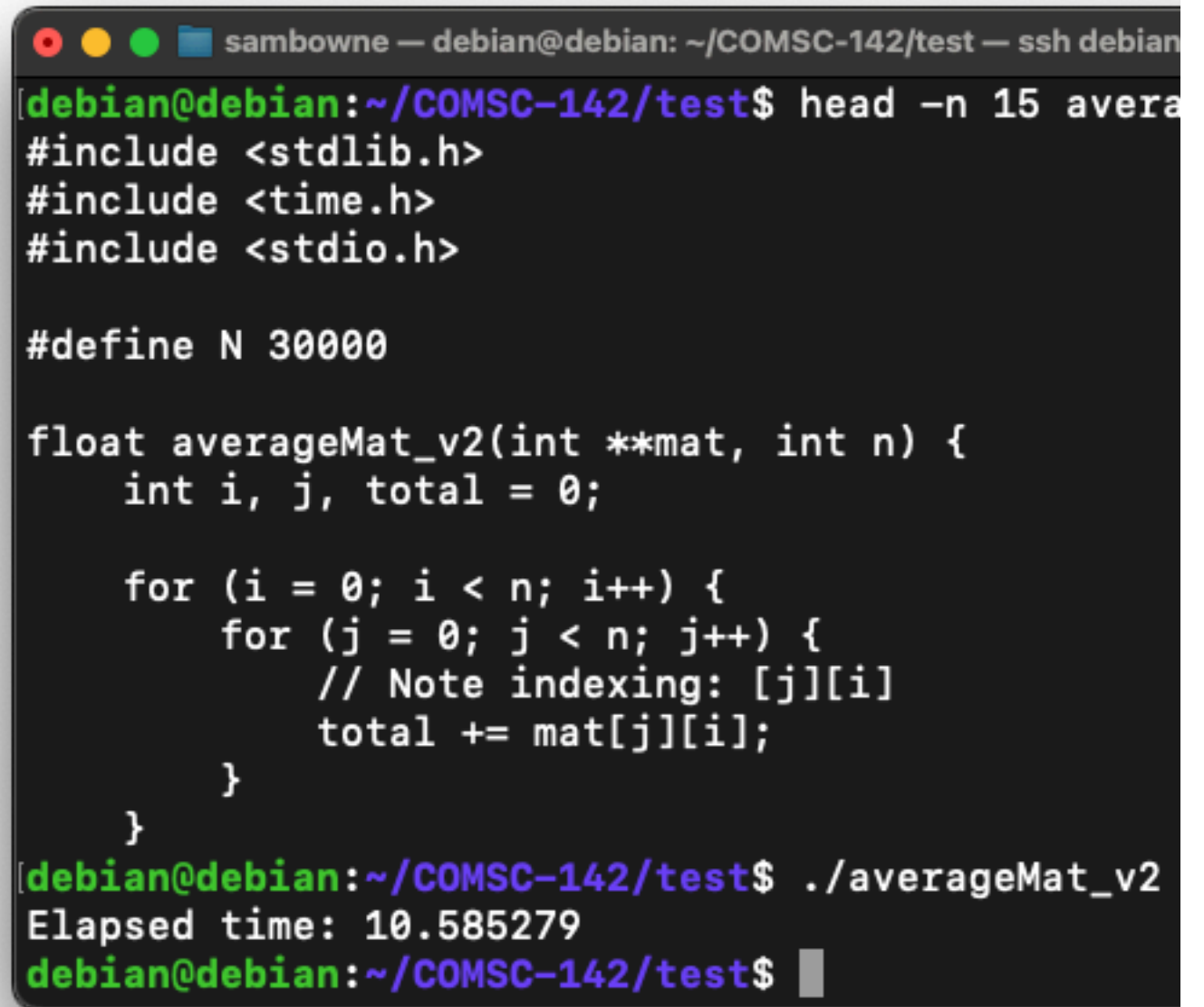
float averageMat_v1(int **mat, int n) {
    int i, j, total = 0;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            // Note indexing: [i][j]
            total += mat[i][j];
        }
    }
}

debian@debian:~/COMSC-142/test$ ./averageMat_v1
Elapsed time: 2.422150
debian@debian:~/COMSC-142/test$
```

Demonstration of Timing

- `wget https://samsclass.info/COMSC-142/proj/averageMat_v2.c`
- `gcc -o averageMat_v2 averageMat_v2.c`
- `head -n 15 averageMat_v2.c`
- `./averageMat_v2`



A terminal window titled "sambowne — debian@debian: ~/COMSC-142/test — ssh debian" displays the following content:

```
debian@debian:~/COMSC-142/test$ head -n 15 averageMat_v2.c
#include <stdlib.h>
#include <time.h>
#include <stdio.h>

#define N 30000

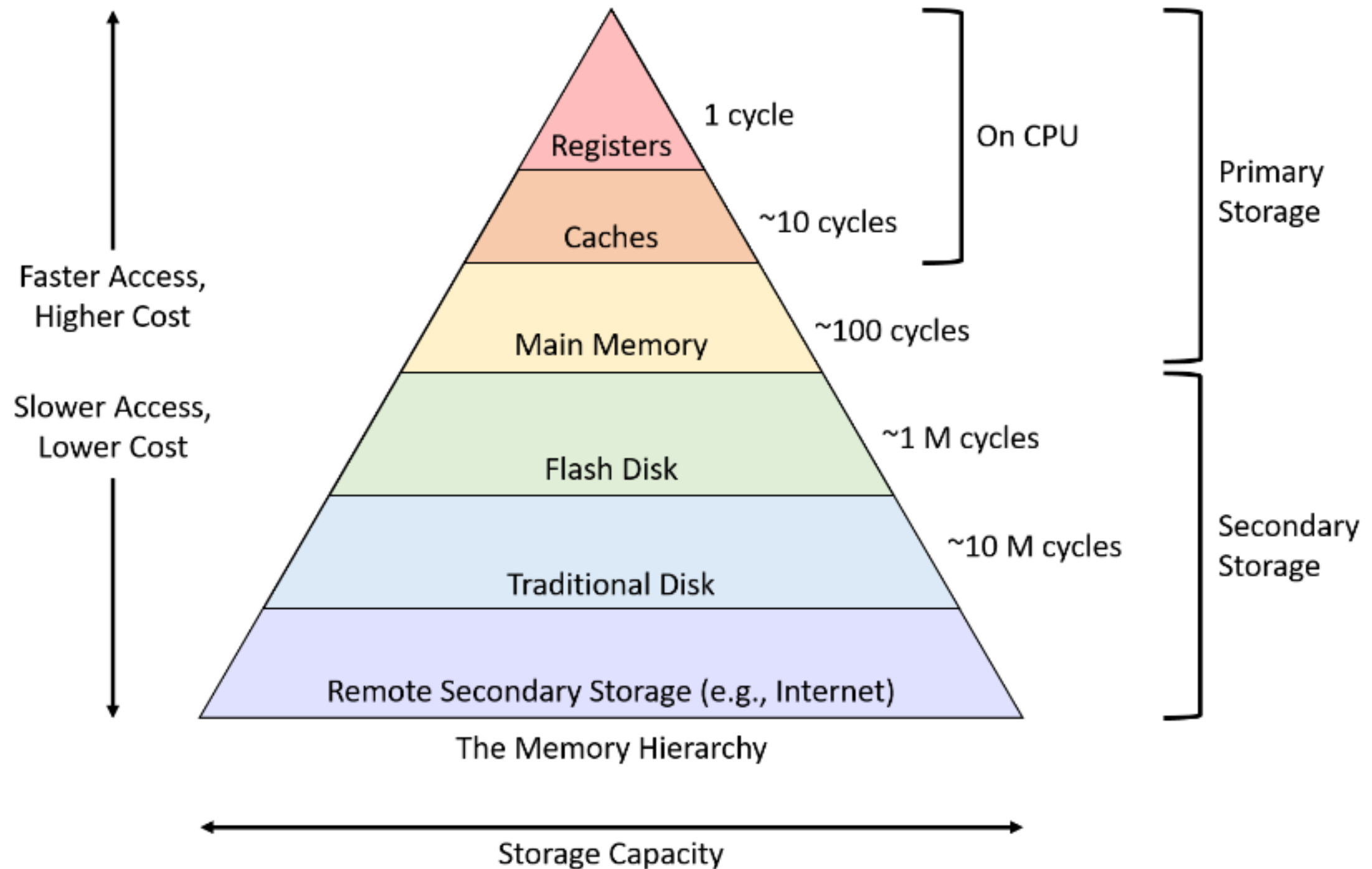
float averageMat_v2(int **mat, int n) {
    int i, j, total = 0;

    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            // Note indexing: [j][i]
            total += mat[j][i];
        }
    }
}

debian@debian:~/COMSC-142/test$ ./averageMat_v2
Elapsed time: 10.585279
debian@debian:~/COMSC-142/test$
```

11.1. The Memory Hierarchy

The Memory Hierarchy



Cache Levels

- Level 1 (L1)
 - Sits close to the ALU
- L2
 - Slower, further from the ALU
- L3
 - Used to share data between cores in a multicore CPU

11.2. Storage Devices

Primary and Secondary Storage

- Primary Storage
 - Can be accessed directly by the CPU
 - Registers and RAM
 - Examples: %rax, (%rax)
- Secondary Storage
 - Cannot be accessed directly by the CPU
 - Examples: Hard disk, SSD, Floppy disk, remote file servers, etc.
 - CPU must first request the device to copy data into primary storage to access it

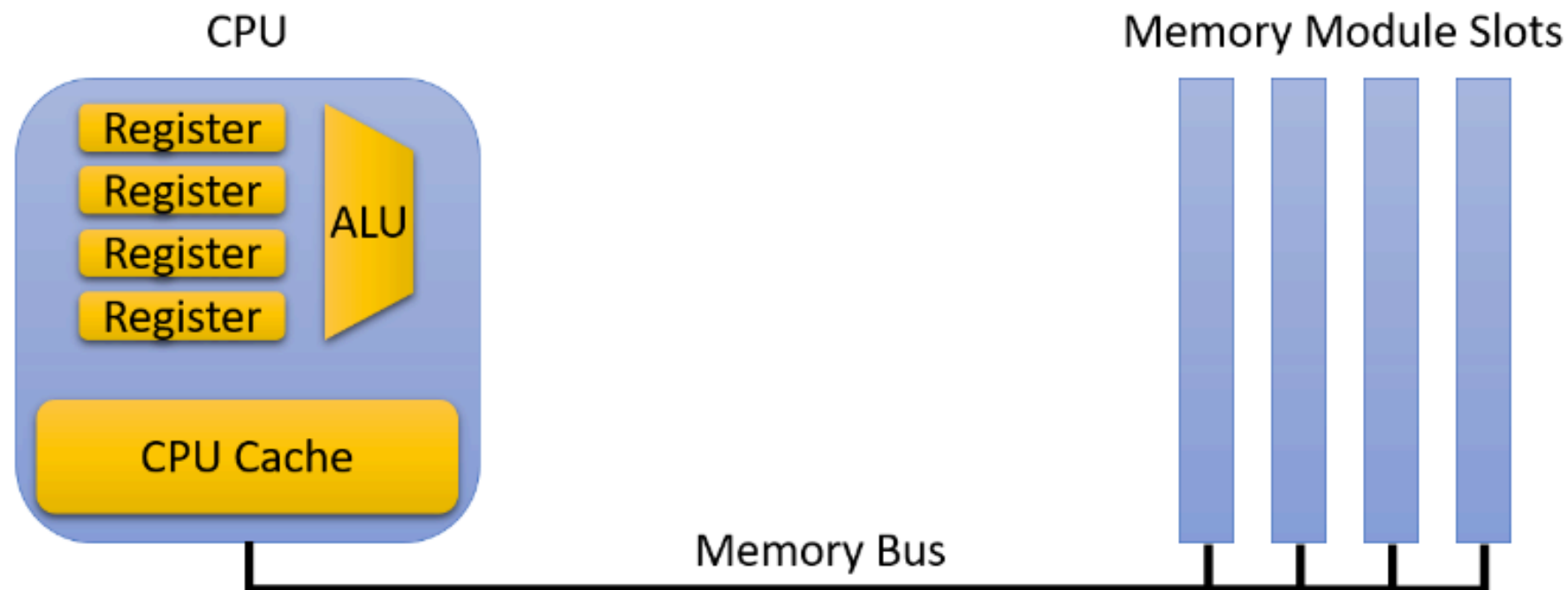
Criteria for Memory Devices

- Capacity
- Latency
 - Time from a request for data to the data being retrieved
- Transfer rate
 - Also called **throughput**
 - Amount of bytes per second retrieved

11.2.1. Primary Storage

Table 1. Primary Storage Device Characteristics of a Typical 2020 Workstation

Device	Capacity	Approx. latency	RAM type
Register	4 - 8 bytes	< 1 ns	SRAM
CPU cache	1 - 32 megabytes	5 ns	SRAM
Main memory	4 - 64 gigabytes	100 ns	DRAM



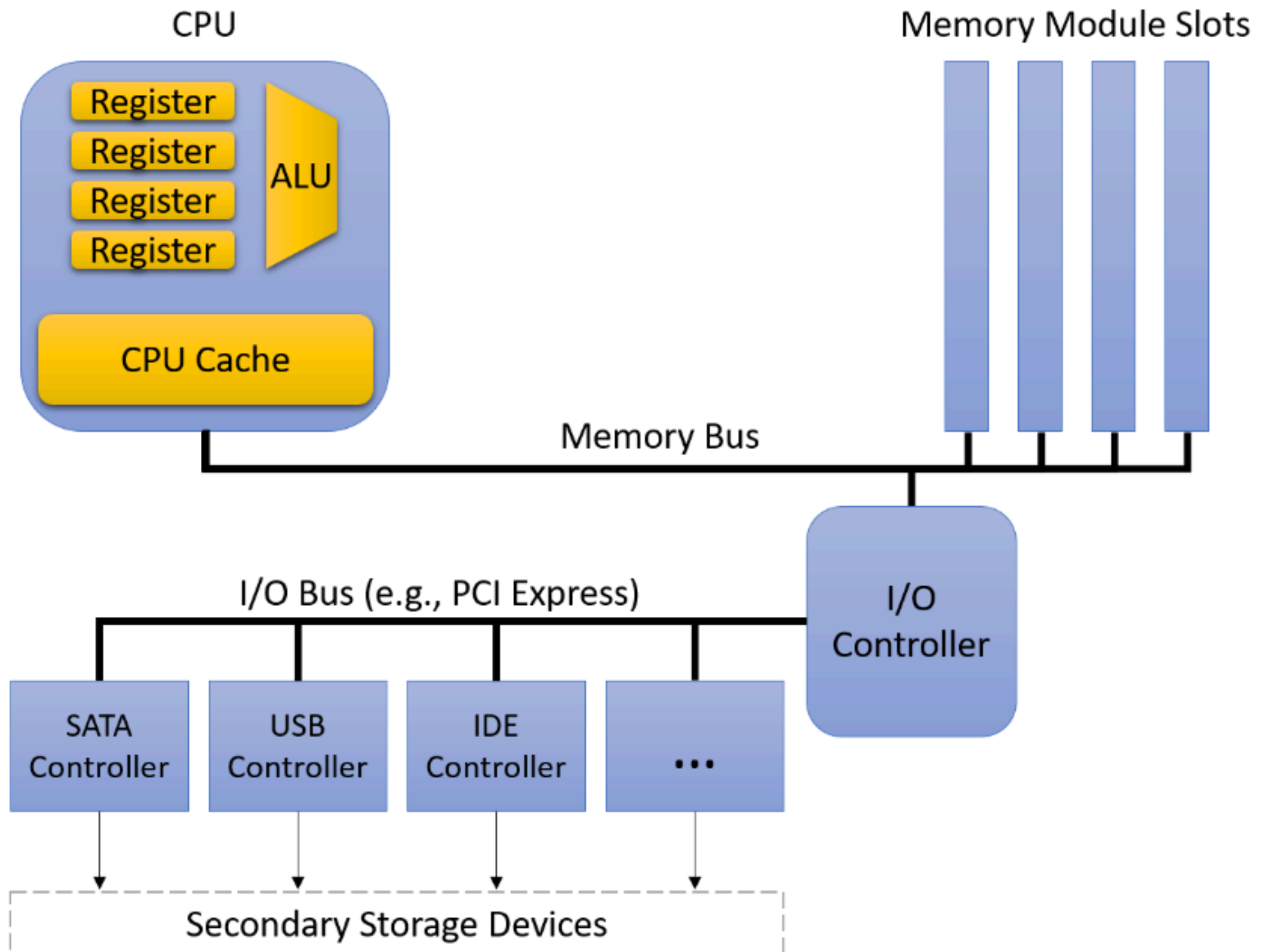
Cache

- Capacity is a few KB or MB
- Programmer doesn't explicitly load the cache
- CPU automatically loads it from RAM

```
sambowne — debian@debian: ~/COMSC-142/test — ssh debian@172.16.123.130 — 85x7
[debian@debian:~/COMSC-142/test$ lscpu -C
NAME  ONE-SIZE  ALL-SIZE  WAYS  TYPE          LEVEL  SETS  PHY-LINE  COHERENCY-SIZE
L1d   32K       64K       8     Data          1      64    1         64
L1i   32K       64K       8     Instruction    1      64    1         64
L2    256K      512K      4     Unified       2     1024   1         64
L3    12M       24M      16     Unified       3    12288   1         64
debian@debian:~/COMSC-142/test$
```

11.2.2. Secondary Storage

Device	Capacity	Latency	Transfer rate
Flash disk (SSD)	0.5 - 2 terabytes	0.1 - 1 ms	200 - 3,000 megabytes / second
Traditional hard disk	0.5 - 10 terabytes	5 - 10 ms	100 - 200 megabytes / second
Remote network server	Varies considerably	20 - 200 ms	Varies considerably



11.3. Locality

Two Types of Locality

- Temporal locality
 - If a program has used a variable recently, it's likely to use that variable again soon.
- Spatial locality
 - If a program accesses data at addresses N and $N+4$, it's likely to access $N+8$ soon

```
/* Sum up the elements in an integer array of length len. */
int sum_array(int *array, int len) {
    int i;
    int sum = 0;

    for (i = 0; i < len; i++) {
        sum += array[i];
    }

    return sum;
}
```

Two Types of Locality

- Temporal locality
 - **i**, **len**, **array**, and **sum** are accessed repeatedly
 - They'll be loaded into cache memory only once
- Spatial locality
 - Many elements of **array** are used in sequence
 - Modern systems will load a block of data into the cache at once
 - Including several integer values
 - A 16-byte **block size** will load 4 integers at a time

```
for (i = 0; i < len; i++) {  
    sum += array[i];  
}
```

Demonstration of Timing

- The first version loads elements in order
 - So a single read of RAM loads several elements at once
- The second version loads elements out of sequence
 - Only loads one element per read of RAM

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        // Note indexing: [i][j]
        total += mat[i][j];
    }
}
```

debian@debian:~/COMSC-142/test\$./aver

Elapsed time: 2.422150

debian@debian:~/COMSC-142/test\$

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        // Note indexing: [j][i]
        total += mat[j][i];
    }
}
```

debian@debian:~/COMSC-142/test\$./aver

Elapsed time: 10.585279

debian@debian:~/COMSC-142/test\$

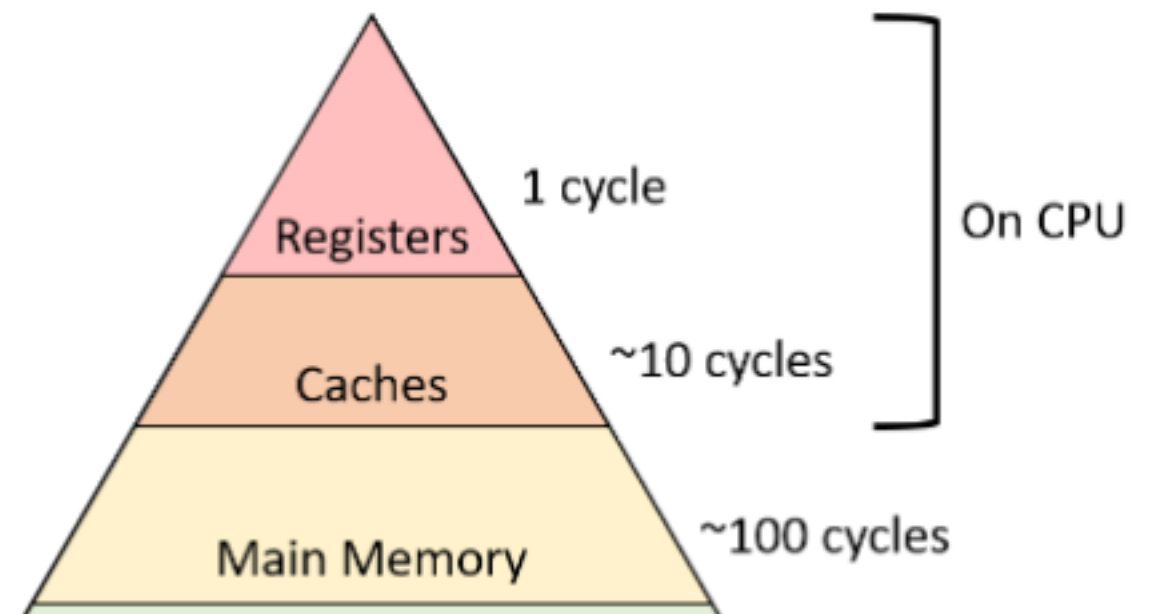
Kahoot!

Ch 11a

11.4. Caching

11.4. CPU Caches

- When the CPU needs data, it firsts calculates the address of the desired data
- It then sends the address to both the cache and main memory, with two possible results
- **Cache hit**
 - The data is found in the cache
 - Memory access is cancelled
- **Cache miss**
 - The data is not found in the cache
 - CPU waits for main memory to respond



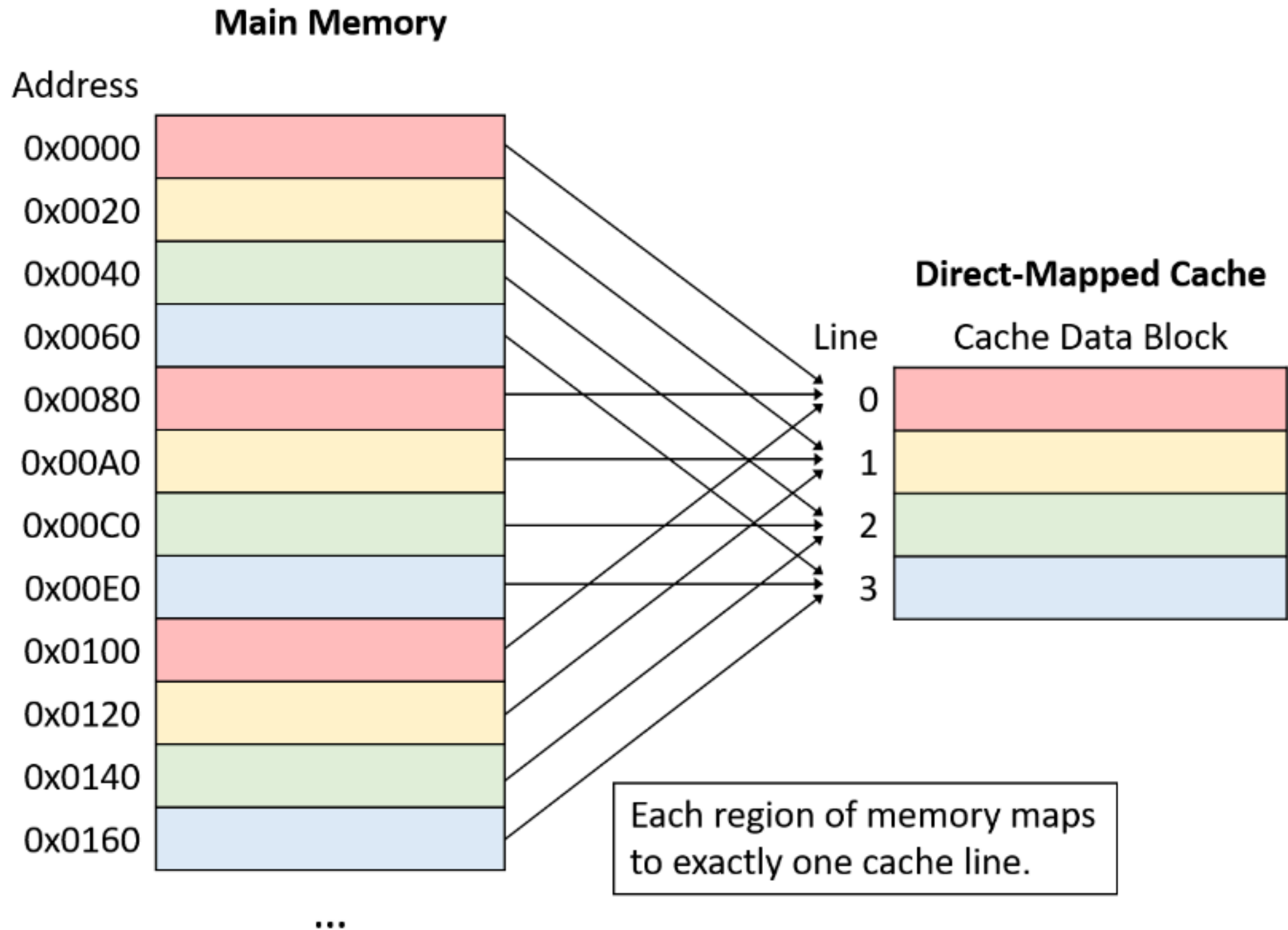
Cache Eviction

- After a **cache miss**
 - Data from RAM is loaded into the cache
- The cache is often full
 - So some resident data must be **evicted**
 - Requiring a write to RAM (in case it's been updated)
- There are three cache designs:
 - **Direct-Mapped**
 - **Fully Associative**
 - **Set Associative**

11.4.1. Direct-Mapped Caches

- Cache storage is divided into **cache lines**
- Each cache line is independent
 - Contains two types of information:
 - **Cache data block** or **cache block**
 - A block of program data from main memory
 - Larger block size is best for programs with good spatial locality
 - Typically 16-64 bytes
 - **Metadata**
 - Information about the contents of the cache line's data block
 - Identifying which subset of memory the data block holds

Locating Cached Data



Locating Cached Data

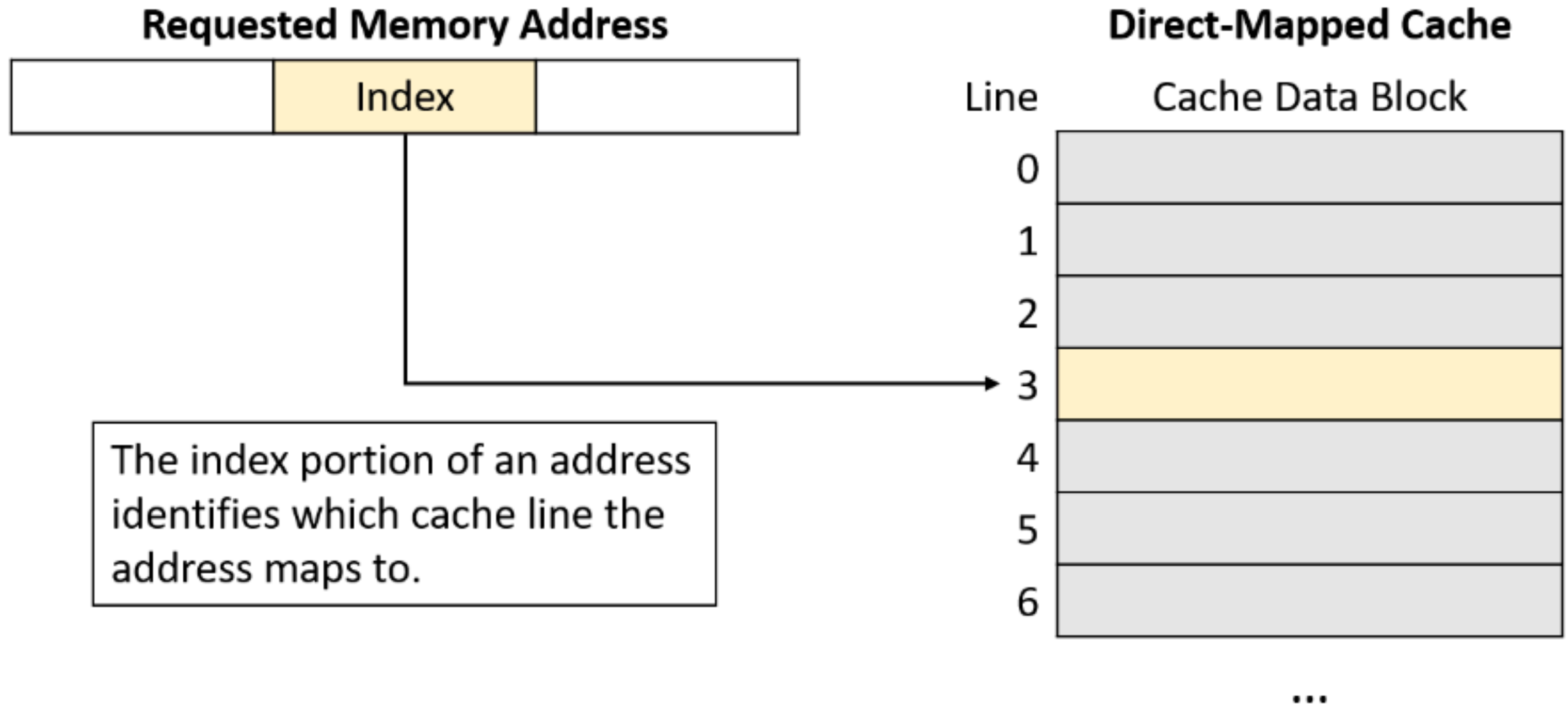


Figure 2. The middle index portion of a memory address identifies a cache line.

Identifying Cache Contents

- Cache metadata answers:
 - Does this cache line hold a valid subset of memory?
 - **Valid bit** is 1 if it does
 - If so, which of the many subsets of memory that map to this cache line does it currently hold?
 - **Tag** stores the higher-order bits of the address range stored in the cache line

Identifying Cache Contents

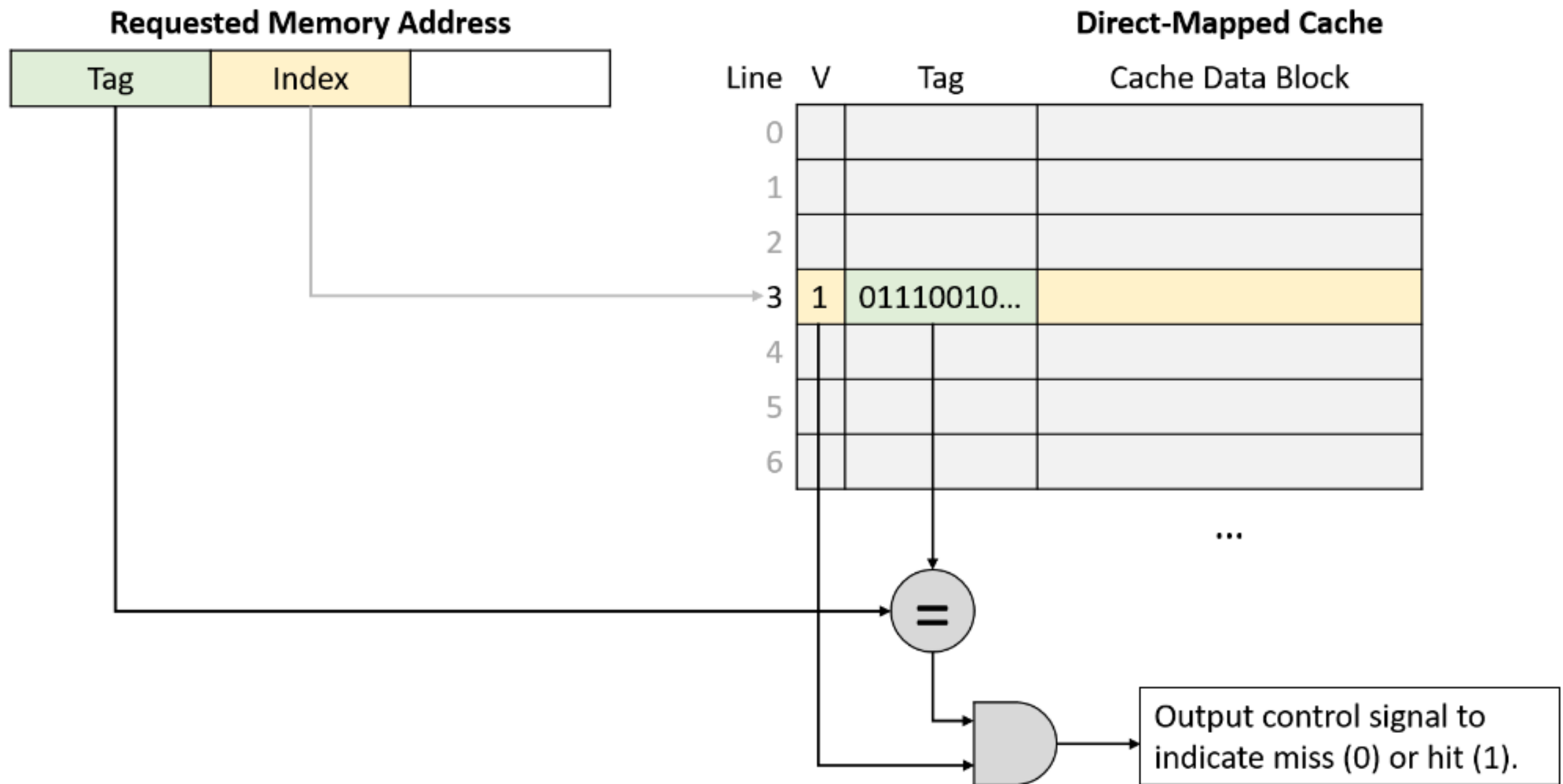
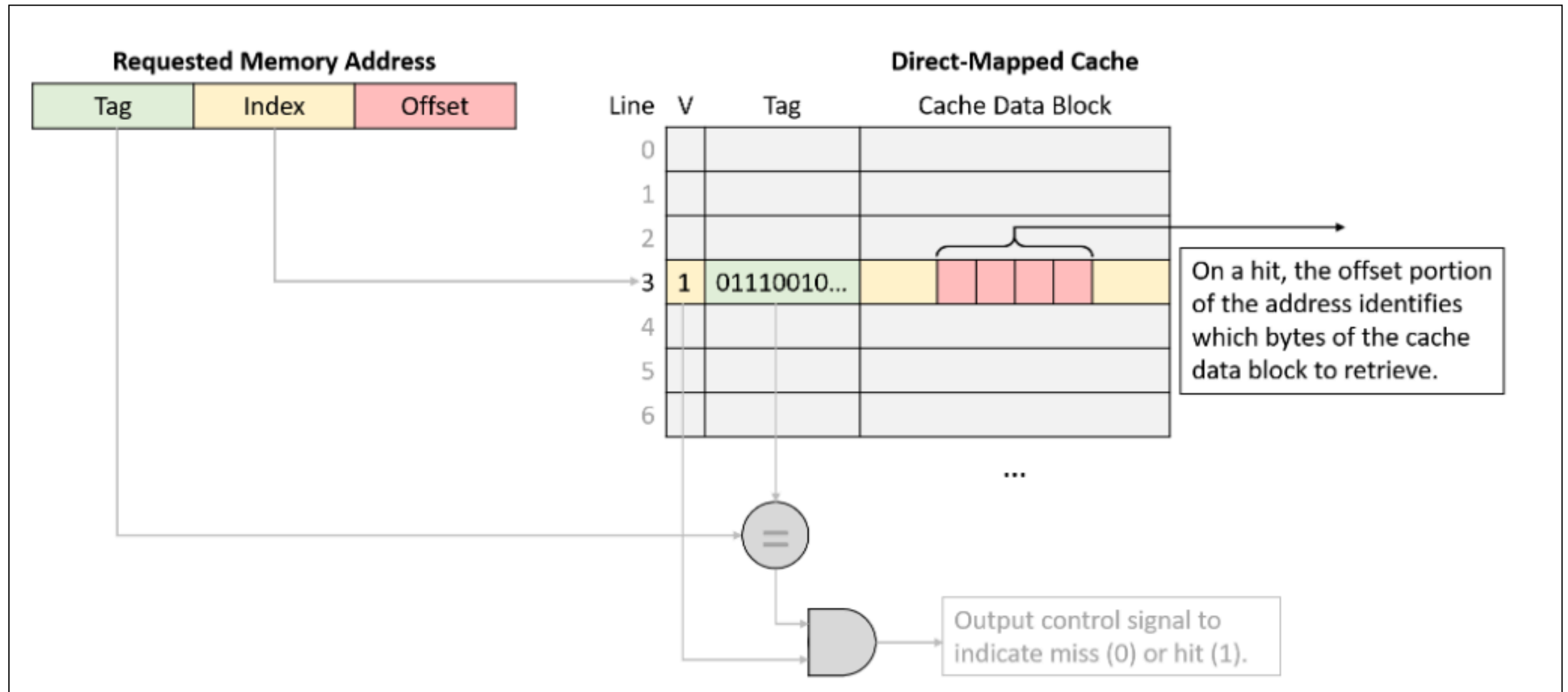


Figure 3. After using the requested memory address's index bits to locate the proper cache line, the cache simultaneously verifies the line's valid bit and checks its tag against the requested address's tag. If the line is valid with a matching tag, the lookup succeeds as a hit.

Retrieving Cached Data

- **Offset** is the lower-order bits of the requested address
- Identifies the required bytes in the cache block



Memory Address Division

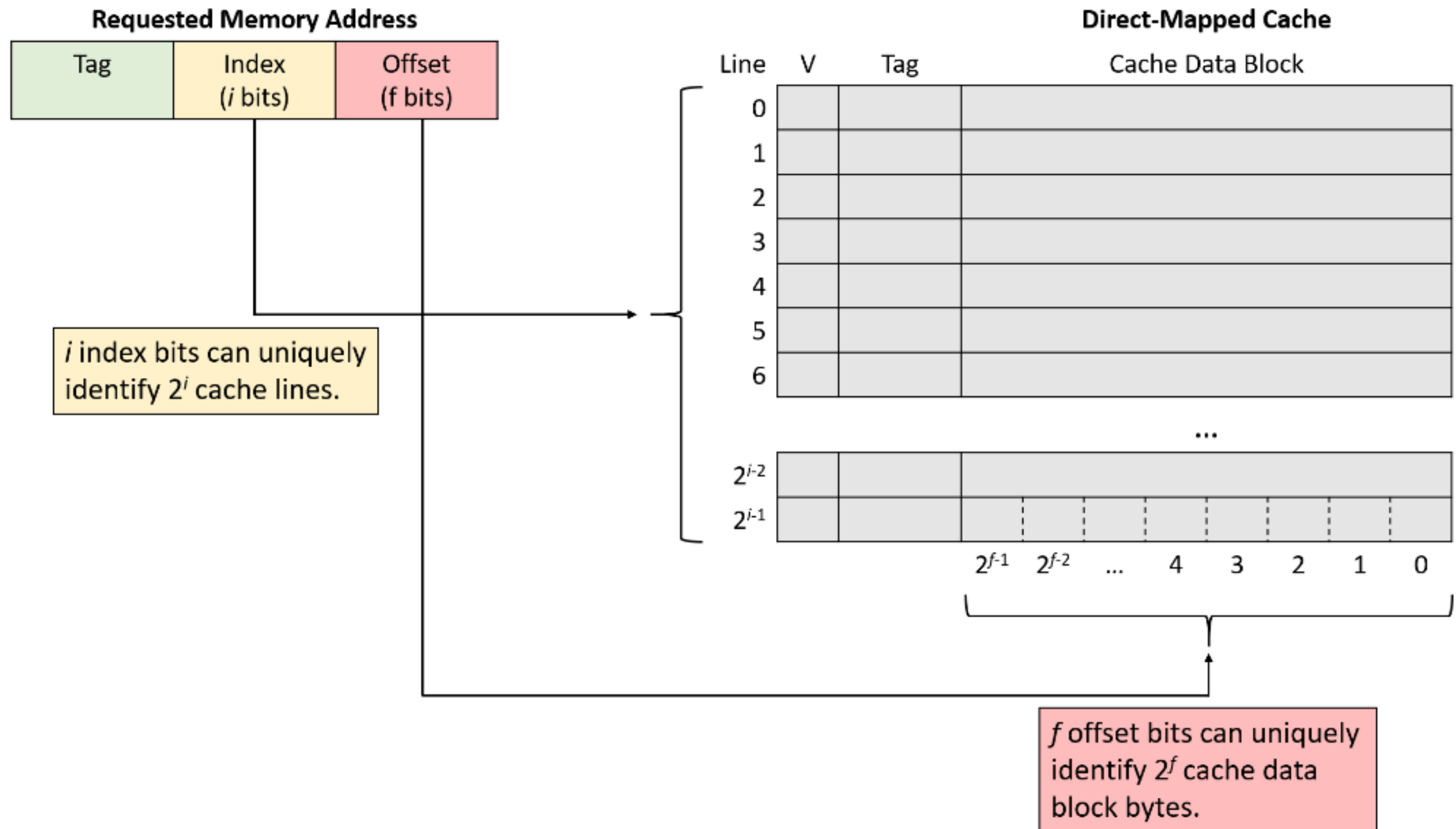


Figure 5. The index portion of an address uniquely identifies a cache line, and the offset portion uniquely identifies a position in the line's data block.

Direct-Mapped Read Example

- Consider a CPU with the following characteristics:
 - 16-bit memory addresses
 - a direct-mapped cache with 128 cache lines
 - 32-byte cache data blocks.

Direct-Mapped Cache			
Line	V	Tag	Cache Data Block (32 bytes)
0	0		
1	0		
2	0		
3	0		
4	0		
...			
127	0		

Figure 6. An empty direct-mapped example cache

First Read: Cache Miss

Read from address 1010000001100100:

Tag	Index	Offset
1010	0000011	00100

Result: miss, the line was invalid prior to this access.

Direct-Mapped Cache			
Line	V	Tag	Cache Data Block (32 bytes)
0	0		
1	0		
2	0		
3	1	1010	← Load data from Memory
4	0		
...			
127	0		

Figure 7. Read from address 1010000001100100. Index 0000011 (line 3) is invalid, so the request misses and the cache loads data from main memory.

Next Read: Cache Hit

Read from address 1010000001100111:

Tag	Index	Offset
1010	0000011	00111

Result: hit, the line is valid,
and the tag matches.

Direct-Mapped Cache			
Line	V	Tag	Cache Data Block (32 bytes)
0	0		
1	0		
2	0		
3	1	1010	
4	0		
...			
127	0		

Figure 8. Read from address 1010000001100111. Index 0000011 (line 3) is valid, and the tag (1010) matches, so the request hits. The cache yields data beginning at byte 7 (offset 0b00111) of its data block.

Next Read: Cache Miss

Read from address 1001000000100000:

Tag	Index	Offset
1001	0000001	00000

Direct-Mapped Cache

Line	V	Tag	Cache Data Block (32 bytes)
0	0		
1	1	1001	← Load data from Memory
2	0		
3	1	1010	
4	0		
...			
127	0		

Result: miss, the line was invalid prior to this access.

Figure 9. Read from address 1001000000100000. Index 0000001 (line 1) is invalid, so the request misses and the cache loads data from main memory.

Next Read: Cache Miss

Read from address 1111000001100101:

Tag	Index	Offset
1111	0000011	00101

Result: miss, the line is valid,
but the tag doesn't match.

Direct-Mapped Cache			
Line	V	Tag	Cache Data Block (32 bytes)
0	0		
1	1	1001	
2	0		
3	1	1111	← Load data from Memory
4	0		
...			
127	0		

Figure 10. Read from address 1111000001100101. Index 0000011 (line 3) is valid, but the tag doesn't match, so the request misses and the cache loads data from main memory.

Writing to Cached Data

- Two strategies.
 - **Write-through cache**
 - Modify the value in the cache and simultaneously update the contents of main memory
 - **Write-back cache**
 - Modify the value stored in the cache's data block, but don't update main memory
 - After updating the cache's data, the cache's contents differ from the corresponding data in main memory
 - Store a **dirty bit** as additional metadata

Dirty Bit

Dirty: a one-bit flag that indicates whether the data stored in a cache line has been modified. When set, the data in the cache line is out of sync with main memory and must be written back to memory before eviction.

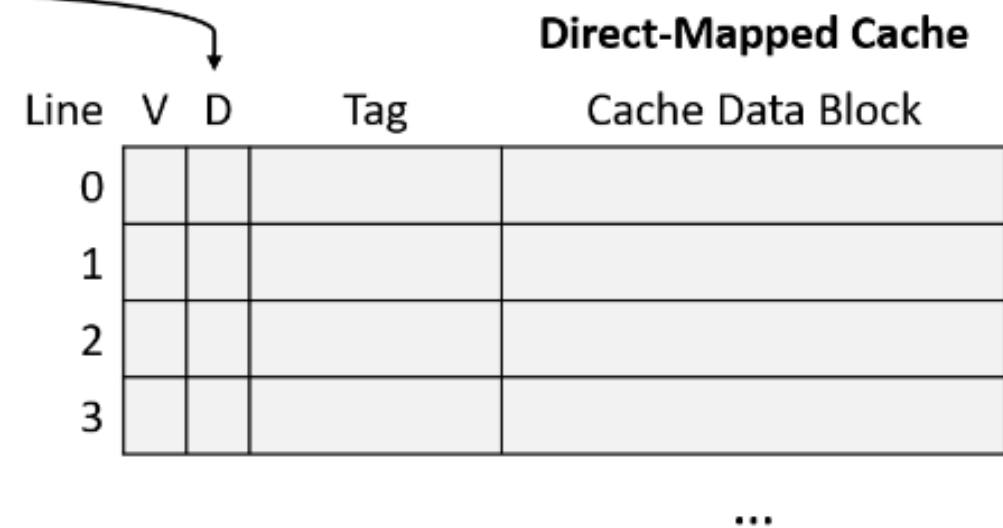


Figure 11. Cache extended with a dirty bit

- Write-back caches are more complex than write-through caches
- But they reduce the cost of repeated writes to the same location in memory

Direct-Mapped Write Examples (Write-Back)

Write to address 1111000001100000:

Tag	Index	Offset
1111	0000011	00000

Result: hit, the line is valid,
and the tag matches.

Set *dirty* bit to 1 on write.

Direct-Mapped Cache				
Line	V	D	Tag	Cache Data Block (32 bytes)
0	0	0		
1	1	0	1001	
2	0	0		
3	1	1	1111	
4	0	0		
...				
127	0	0		

Figure 12. Write to address 1111000001100000. Index 0000011 (line 3) is valid, and the tag (1111) matches, so the request hits. Because this access is a write, the cache sets the line's dirty bit to 1.

Direct-Mapped Write Examples (Write-Back)

Write to address 1010000001100100:

Tag	Index	Offset
1010	0000011	00100

Direct-Mapped Cache				
Line	V	D	Tag	Cache Data Block (32 bytes)
0	0	0		
1	1	0	1001	
2	0	0		
3	1	1	1010	
4	0	0		
...				
127	0	0		

Annotations for Line 3:

- Orange arrow pointing right: Save data to Memory
- Blue arrow pointing left: Load data from Memory

Result: miss, the line is valid, but the tag doesn't match.

Save cache data block to memory before evicting it.

Set *dirty* bit to 1 on write. (again)

Figure 13. Write to address 1010000001100100. Index 0000011 (line 3) is valid, but the tag doesn't match, so the request misses. Because the target line is both valid and dirty, the cache must save the existing data block to main memory before loading the new one. This access is a write, so the cache sets the newly loaded line's dirty bit to 1.

11.4.2. Cache Misses and Associative Designs

- What causes cache misses?
 - **Compulsory misses** or **cold-start misses**
 - Program has never accessed a memory location (or any location near it)
 - **Capacity misses**
 - Program uses more memory than fits in the cache, it can't possibly find all of the data it wants in the cache, leading to misses
 - **Conflict misses**
 - Two frequently used variables map to the same cache location
 - Each access to one of those variables evicts the other from the cache as they compete for the same cache line

Associative Cache Types

- **Fully Associative**

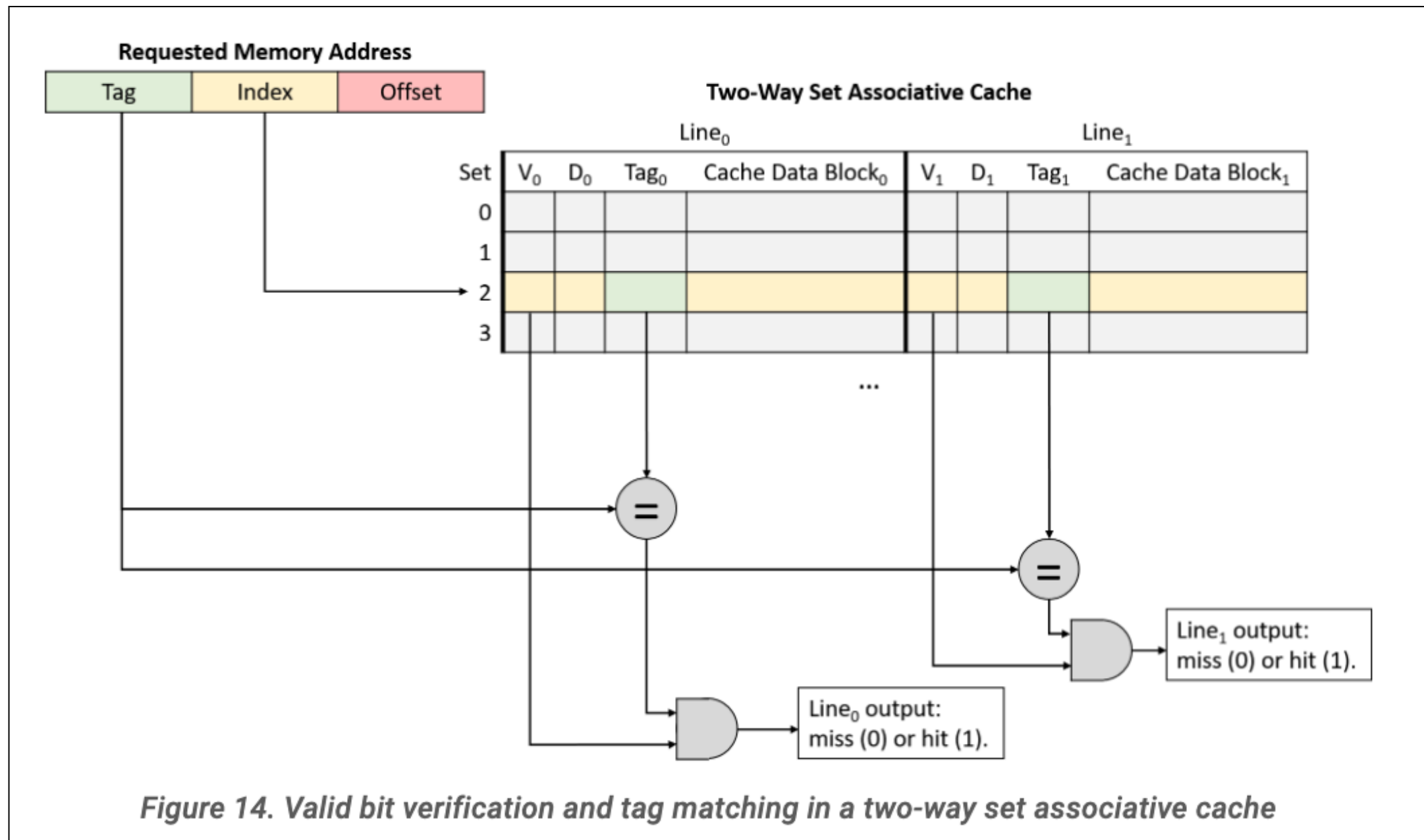
- Allows any memory region to occupy any cache location
- Maximum flexibility, but highest lookup and eviction complexity
- Every location needs to be simultaneously considered during any operation
- Valuable in some small, specialized applications (for example, translation look-aside buffers), their high complexity makes them generally unfit for a general-purpose CPU cache

Associative Cache Types

- **Set Associative**
 - Middle ground between direct-mapped and fully associative designs
 - Well suited for general-purpose CPUs
 - Every memory region maps to exactly one cache set, but each set stores multiple cache lines
 - The number of lines allowed in a set is a fixed dimension of a cache, and set associative caches typically store two to eight lines per set

11.4.3. Set Associative Caches

- Cache simultaneously checks every line in the set



Cache Replacement Policy

- When loading a value into a cache
 - (and when evicting data already resident in the cache)
- The cache must decide which of the line options to use
- It uses the LRU (Least Recently Used) line

LRU: a one-bit flag that indicates whether the leftmost line₀ of the set was least recently used (LRU = 0) or the rightmost line₁ of the set was least recently used (LRU = 1).

Two-Way Set Associative Cache

Set	LRU	Line ₀				Line ₁			
		V ₀	D ₀	Tag ₀	Cache Data Block ₀	V ₁	D ₁	Tag ₁	Cache Data Block ₁
0									
1									
2									
3									

...

Figure 15. A two-way set associative cache in which each set stores one bit of LRU metadata to inform eviction decisions

Set Associative Cache Examples

Consider a CPU with the following characteristics:

- 16-bit memory addresses.
- A two-way set associative cache with 64 sets. Note that making a cache two-way set associative doubles its storage capacity (two lines per set), so this example halves the number of sets so that it stores the same number of lines as the earlier direct-mapped example.
- 32-byte cache blocks.
- An LRU cache replacement policy that indicates whether the leftmost line of the set was least recently used ($\text{LRU} = 0$) or the rightmost line of the set was least recently used ($\text{LRU} = 1$).

Set Associative Cache Examples

Two-Way Set Associative Cache

		Line ₀				Line ₁			
Set	LRU	V ₀	D ₀	Tag ₀	Cache Data Block ₀	V ₁	D ₁	Tag ₁	Cache Data Block ₁
0	0	0	0			0	0		
1	0	0	0			0	0		
2	0	0	0			0	0		
3	0	0	0			0	0		
4	0	0	0			0	0		
...									
63	0	0	0			0	0		

Figure 16. An empty two-way set associative example cache

Set Associative Cache Examples

Read from address 1010000001100100:

Tag	Index	Offset
10100	000011	00100

Two-Way Set Associative Cache

		Line ₀				Line ₁			
Set	LRU	V ₀	D ₀	Tag ₀	Cache Data Block ₀	V ₁	D ₁	Tag ₁	Cache Data Block ₁
0	0	0	0			0	0		
1	0	0	0			0	0		
2	0	0	0			0	0		
→ 3	1	1	0	10100	<div>↑</div>	0	0		
4	0	0	0		Load Data	0	0		
...									
63	0	0	0			0	0		

Result: miss, both lines in set 3 are invalid prior to the access.

Update LRU bit to 1.

Figure 17. Read from address 1010000001100100. Both lines at index 000011 (set 3) are invalid, so the request misses, and the cache loads data from main memory. The set's LRU bit is 0, so the cache loads data into the left line and updates the LRU bit to 1.

Set Associative Cache Examples

Read from address 1010000001100111:

Tag	Index	Offset
10100	000011	00111

Two-Way Set Associative Cache

Set	LRU	Line ₀				Line ₁			
		V ₀	D ₀	Tag ₀	Cache Data Block ₀	V ₁	D ₁	Tag ₁	Cache Data Block ₁
0	0	0	0			0	0		
1	0	0	0			0	0		
2	0	0	0			0	0		
3	1	1	0	10100		0	0		
4	0	0	0			0	0		
...									
63	0	0	0			0	0		

Result: hit, one line in the set is valid and holds a matching tag.

Figure 18. Read from address 1010000001100111. The left line at index 000011 (set 3) holds a matching tag, so the request hits.

Set Associative Cache Examples

Read from address 1001000000100000:

Tag	Index	Offset
10010	000001	00000

Two-Way Set Associative Cache

		Line ₀				Line ₁			
Set	LRU	V ₀	D ₀	Tag ₀	Cache Data Block ₀	V ₁	D ₁	Tag ₁	Cache Data Block ₁
0	0	0	0			0	0		
1	1	1	0	10010	⬆ Load Data	0	0		
2	0	0	0			0	0		
3	1	1	0	10100		0	0		
4	0	0	0			0	0		
...									
63	0	0	0			0	0		

Result: miss, both lines in set 1 are invalid prior to the access.

Update LRU bit to 1.

Figure 19. Read from address 1001000000100000. Both lines at index 000001 (set 1) are invalid, so the request misses, and the cache loads data from main memory. The set's LRU bit is 0, so the cache loads data into the left line and updates the LRU bit to 1.

Set Associative Cache Examples

Read from address 1111000001100101:

Tag	Index	Offset
11110	000011	00101

Two-Way Set Associative Cache

Set	LRU	Line ₀				Line ₁			
		V ₀	D ₀	Tag ₀	Cache Data Block ₀	V ₁	D ₁	Tag ₁	Cache Data Block ₁
0	0	0	0			0	0		
1	1	1	0	10010		0	0		
2	0	0	0			0	0		
3	0	1	0	10100		1	0	11110	
4	0	0	0			0	0		
...									
63	0	0	0			0	0		

Load Data

Result: miss, one line's tag doesn't match, and the other is invalid.

Update LRU bit to 0.

Figure 20. Read from address 1111000001100101. At index 000011 (set 3), one line's tag doesn't match, and the other line is invalid, so the request misses. The set's LRU bit is 1, so the cache loads data into the right line and updates the LRU bit to 0.

Set Associative Cache Examples

Write to address 1111000001100000:

Tag	Index	Offset
11110	000011	00000

Two-Way Set Associative Cache

		Line ₀				Line ₁			
Set	LRU	V ₀	D ₀	Tag ₀	Cache Data Block ₀	V ₁	D ₁	Tag ₁	Cache Data Block ₁
0	0	0	0			0	0		
1	1	1	0	10010		0	0		
2	0	0	0			0	0		
3	0	1	0	10100		1	1	11110	
4	0	0	0			0	0		
...									
63	0	0	0			0	0		

Result: hit, one of the valid lines holds a matching tag.

Set line₁'s dirty bit.

Figure 21. Write to address 1111000001100000. The right line at index 000011 (set 3) is valid and holds a matching tag, so the request hits. Because this access is a write, the cache sets the line's dirty bit to 1. The LRU bit remains 0 to indicate that the left line remains least recently used.

Set Associative Cache Examples

Write to address 1010000001100100:

Tag	Index	Offset
10100	000011	00100

Two-Way Set Associative Cache

		Line ₀				Line ₁			
Set	LRU	V ₀	D ₀	Tag ₀	Cache Data Block ₀	V ₁	D ₁	Tag ₁	Cache Data Block ₁
0	0	0	0			0	0		
1	1	1	0	10010		0	0		
2	0	0	0			0	0		
3	1	1	1	10100		1	1	11110	
4	0	0	0			0	0		
...									
63	0	0	0			0	0		

Result: hit, one of the valid lines holds a matching tag.

Set line₀'s dirty bit.

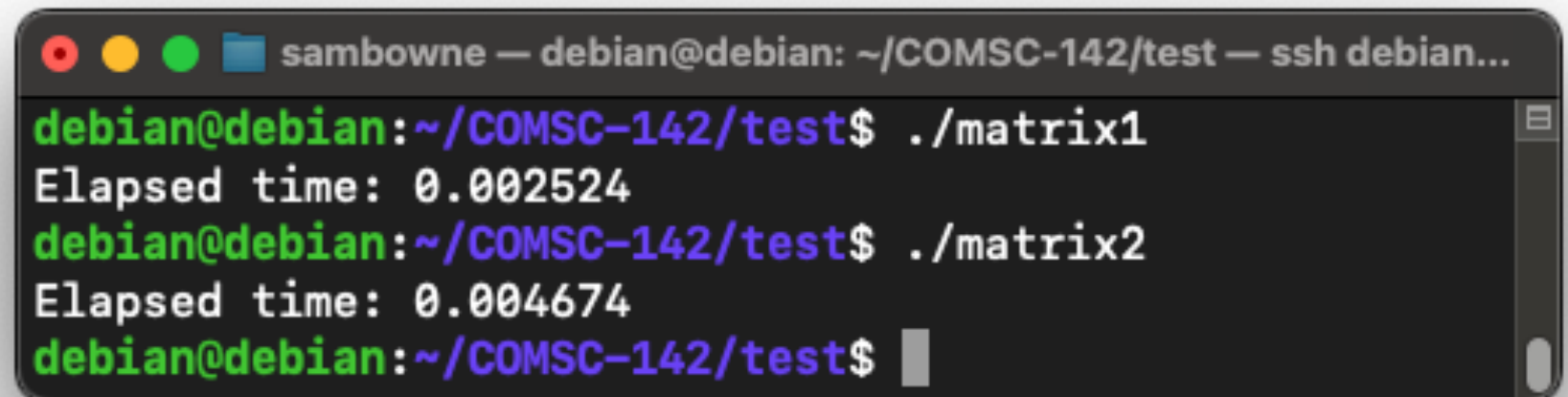
Update LRU bit to 1.

Figure 22. Write to address 1010000001100100. The left line at index 000011 (set 3) is valid and holds a matching tag, so the request hits. Because this access is a write, the cache sets the line's dirty bit to 1. After accessing the left line, the cache sets the line's LRU bit to 1.

11.5. Cache Analysis and Valgrind

Simplifying the Test Programs

- Shrink the previous two programs to 1000x1000
- `wget https://samsclass.info/COMSC-142/proj/matrix1.c`
- `gcc -o matrix1 matrix1.c`
- `wget https://samsclass.info/COMSC-142/proj/matrix2.c`
- `gcc -o matrix2 matrix2.c`
- `./matrix1`
- `./matrix2`



```
sambowne — debian@debian: ~/COMSC-142/test — ssh debian...  
debian@debian:~/COMSC-142/test$ ./matrix1  
Elapsed time: 0.002524  
debian@debian:~/COMSC-142/test$ ./matrix2  
Elapsed time: 0.004674  
debian@debian:~/COMSC-142/test$
```


Using Valgrind

- `valgrind --tool=cachegrind --cache-sim=yes ./matrix1`
- `valgrind --tool=cachegrind --cache-sim=yes ./matrix2`

```
sambowne — debian@debian: ~/COMSC-142/test$ valgrind
==5586== Cachegrind, a cache and branch-predictor simulator
==5586== Copyright (C) 2002-2017, and GNU GPL (C) 2003-2014
==5586== Using Valgrind-3.19.0 and LibVEX; rerun with --shared for a
==5586== Command: ./matrix1
==5586==
--5586-- warning: L3 cache found, using i387 cache
--5586-- warning: specified LL cache: line 1
--5586-- warning: simulated LL cache: line 1
Elapsed time: 0.040995
==5586==
==5586== I   refs:      16,303,102
==5586== I1  misses:      1,342
==5586== LLi misses:      1,320
==5586== I1  miss rate:      0.01%
==5586== LLi miss rate:      0.01%
==5586==
==5586== D   refs:      9,102,328 (9,072,000)
==5586== D1  misses:      65,781 (65,781)
==5586== LLd misses:      64,222 (64,222)
==5586== D1  miss rate:      0.7% (0.7%)
==5586== LLd miss rate:      0.7% (0.7%)
```

```
sambowne — debian@debian: ~/COMSC-142/test$ valgrind
==5588== Cachegrind, a cache and branch-predictor simulator
==5588== Copyright (C) 2002-2017, and GNU GPL (C) 2003-2014
==5588== Using Valgrind-3.19.0 and LibVEX; rerun with --shared for a
==5588== Command: ./matrix2
==5588==
--5588-- warning: L3 cache found, using i387 cache
--5588-- warning: specified LL cache: line 1
--5588-- warning: simulated LL cache: line 1
Elapsed time: 0.061840
==5588==
==5588== I   refs:      16,303,112
==5588== I1  misses:      1,342
==5588== LLi misses:      1,320
==5588== I1  miss rate:      0.01%
==5588== LLi miss rate:      0.01%
==5588==
==5588== D   refs:      9,102,331 (9,072,000)
==5588== D1  misses:      1,128,901 (1,128,901)
==5588== LLd misses:      64,222 (64,222)
==5588== D1  miss rate:     12.4% (12.4%)
==5588== LLd miss rate:      0.7% (0.7%)
```

11.6. Looking Ahead: Caching on Multicore Processors

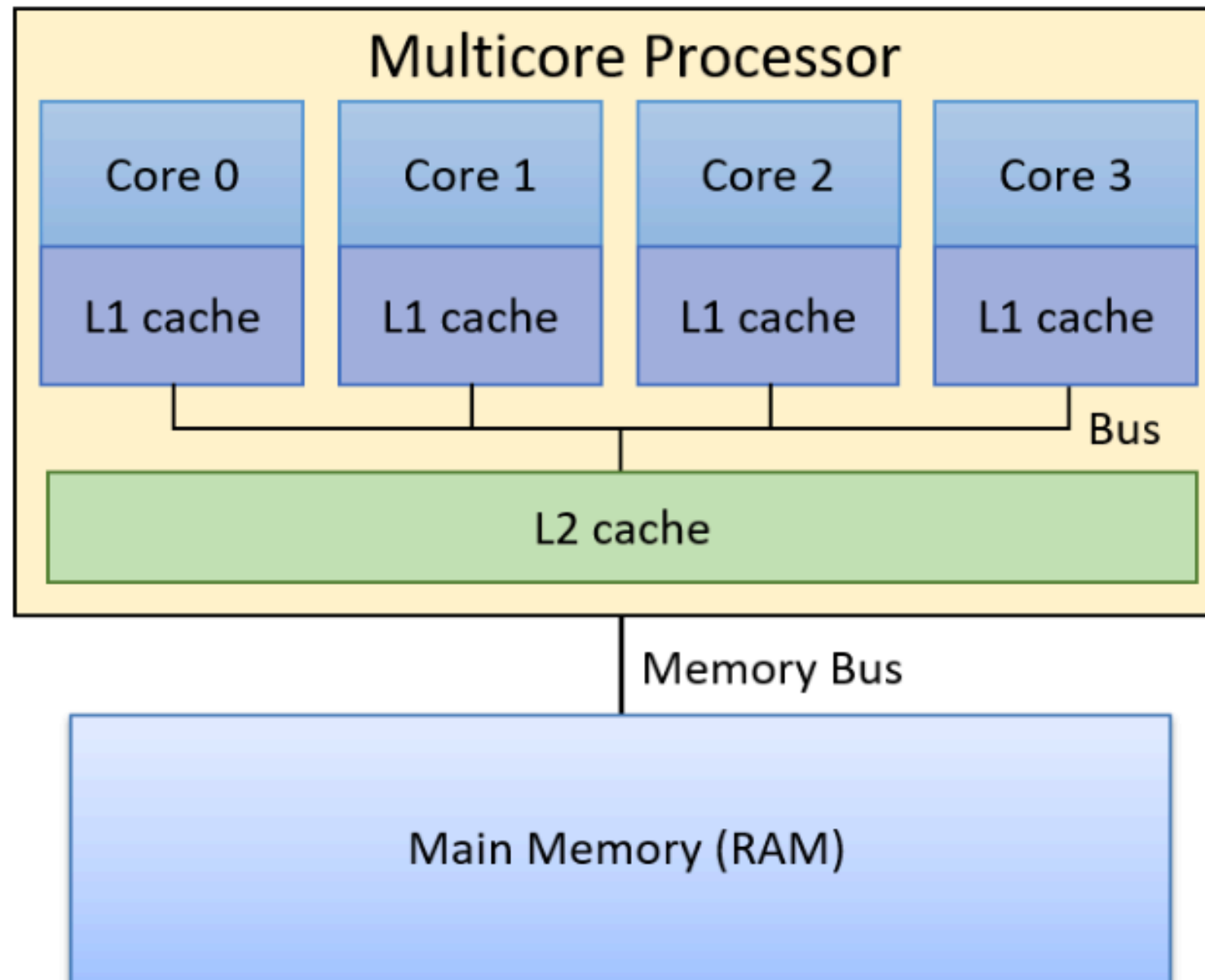


Figure 1. An example memory hierarchy on a multicore processor. Each of the four cores has its own private L1 cache, and all four cores share a single L2 cache that they access through a shared bus. The multicore processor connects to RAM via the memory bus.

11.6.1. Cache Coherency

- Two or more cores might have cached the same data in L1
 - And have different values for the same block of memory
- Multicore processors implement a **cache-coherence protocol**
 - Ensures that any core accessing a memory location
 - Sees the most recently modified value of that memory location
 - Rather than seeing an older (stale) copy of the value that may be stored in its L1 cache
- We'll describe **MSI**, one of the many cache-coherence protocols

11.6.2. The MSI Protocol

- **MSI** (Modified, Shared, Invalid)
 - Adds three flags (or bits) to each cache line
 - **M** set: the block has been **modified**
 - This core has written to its copy of the cached value
 - **S** set: the block is unmodified and can be safely shared
 - Multiple L1 caches may safely store a copy of the block and read from their copy
 - **I** set: the cached block is invalid or contains stale data
 - Is an older copy of the data that does not reflect the current value of the block of memory

MSI: Read

On a read access:

- If the cache block is in the M or S state, the cached value is used to satisfy the read (its copy's value is the most current value of the block of memory).
- If the cache block is in the I state, the cached copy is out of date with a newer version of the block, and the block's new value needs to be loaded into the cache line before the read can be satisfied.
- If another core's L1 stores a new value
 - Stores with the **M** flag set
 - It must first write to the L2 cache
 - Then write-back to its L1 cache, clearing the **M** bit
 - Sets the **S** bit to indicate that the block in this cache line is in a state that can be safely cached by other cores
 - The core that initiated the read access on an line with the I flag set can then load the new value of the block into its cache line.

MSI: Write

- If the block is in the M state, write to the cached copy of the block
 - No changes to the flags are needed (the block remains in the M state)
- If the block is in the I or the S state, notify other cores that the block is being written to (modified)
 - Other L1 caches that have the block stored in the S state, need to clear the S bit and set the I bit on their block (their copies of the block are now out of date with the copy that is being written to by the other core)
 - If another L1 cache has the block in the M state, it will write its block back to the lower level, and set its copy to I
 - The core writing will then load the new value of the block into its L1 cache, set the M flag (its copy will be modified by the write), and clear the I flags (its copy is now valid), and write to the cached block

11.6.3. Implementing Cache Coherency Protocols

- A snooping L1 cache controller listens (or **snoops**) on the bus for reads or writes to blocks that it caches
- MSI and other similar protocols such as MESI and MOESI are write-invalidate protocols; that is, protocols that invalidate copies of cached entries on writes
- Snooping can also be used by write-update cache coherency protocols, where the new value of a data is snooped from the bus and applied to update all copies stored in other L1 caches

11.6.4. More about Multicore Caching

- The benefits to performance
 - Of each core of a multicore processor having its own L1 cache
 - Is worth the added extra complexity of the cache coherency protocol
- There is another problem: **false sharing**
 - If multiple threads of a single multithreaded parallel program are running simultaneously across the multiple cores
 - And are accessing memory locations that are near to those accessed by other threads
 - In section Chapter 14.5, we discuss the false sharing problem and some solutions to it

Kahoot!

Ch 11b