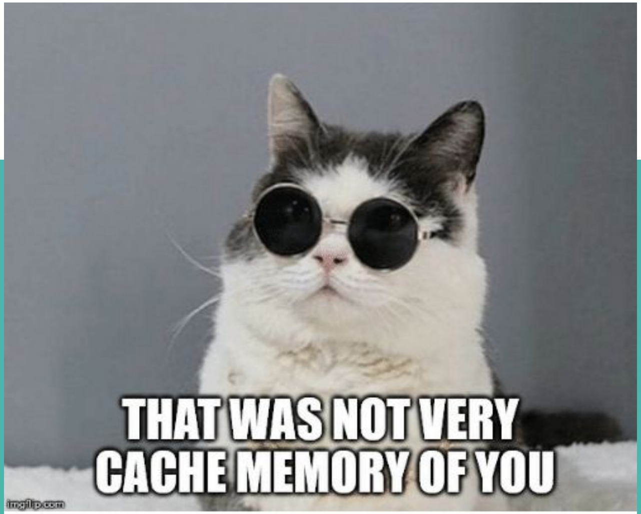
CSE 351 Section 7

Caches

Administrivia

- Lab 3
  - Due Friday, November 11
- Homework 17
  - Due Wednesday, November 16

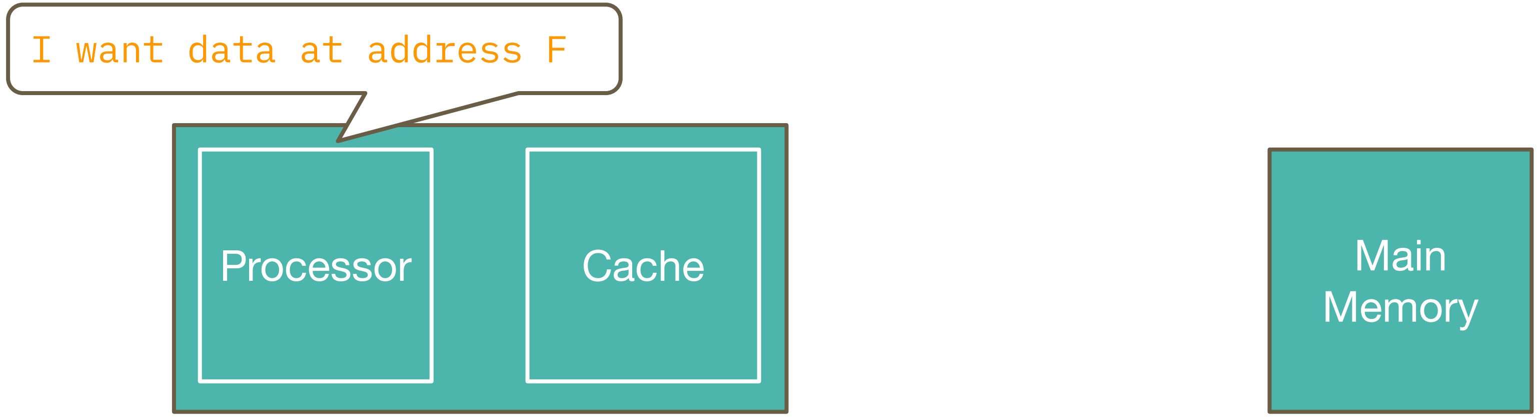
Caches



When your CPU fetches from RAM

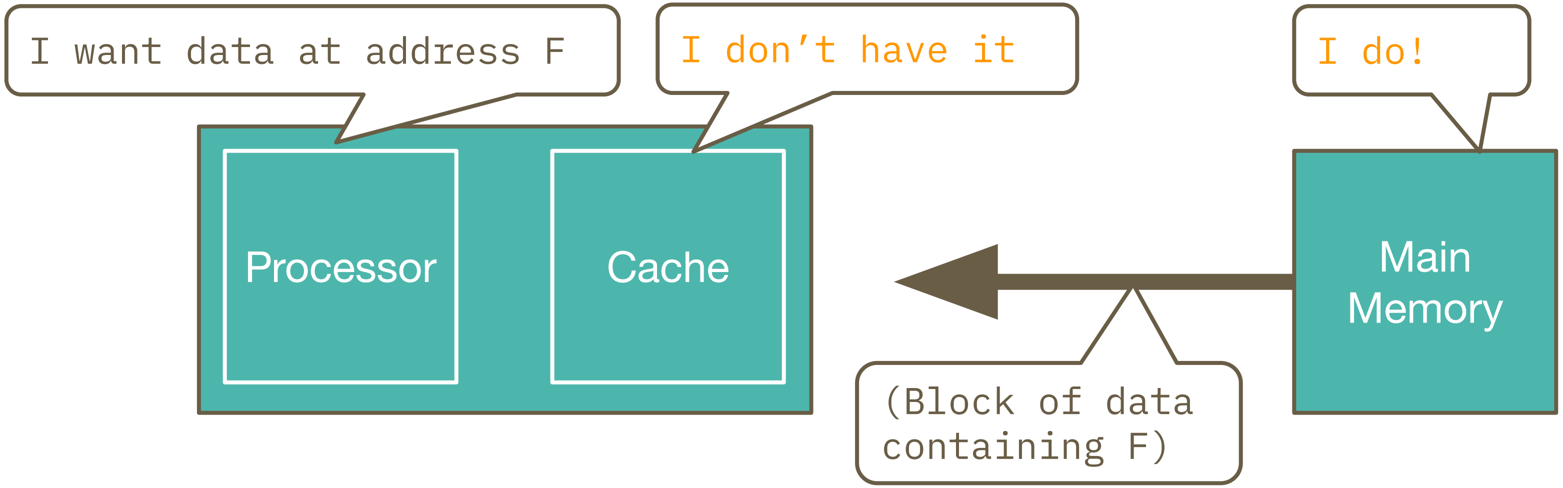
Cache Review

The cache stores a subset of main memory with much faster access time! It is located much closer to the processor, often on the same chip. When we access memory, we check the cache(s) first.



Cache Review

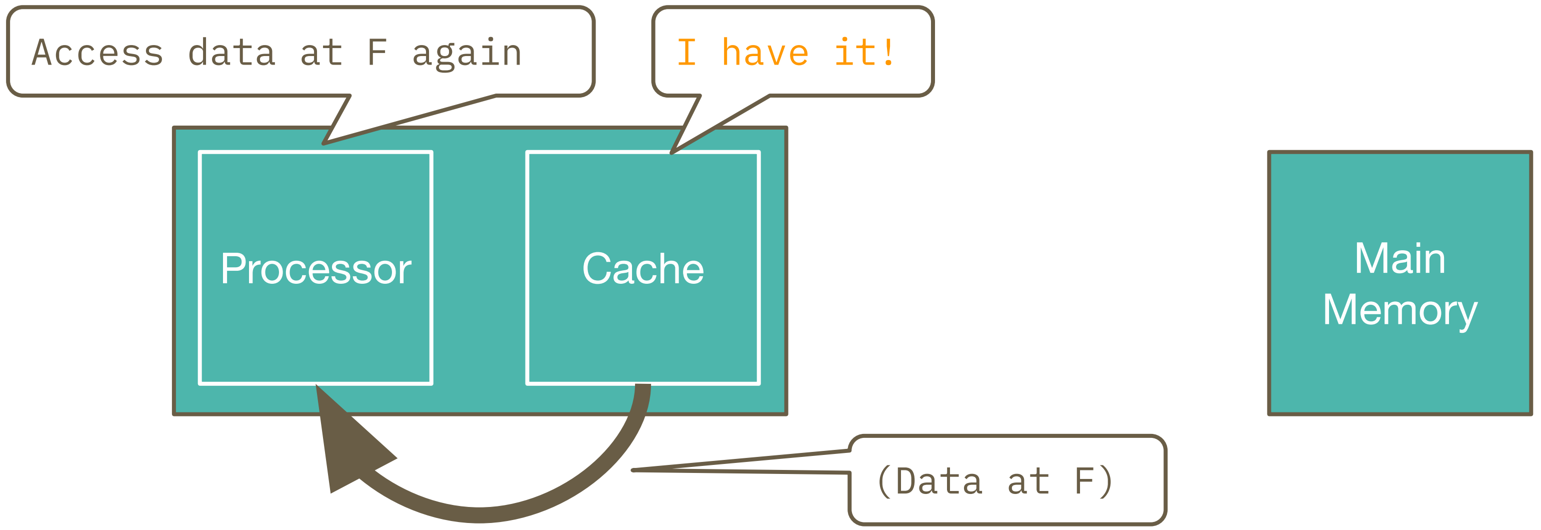
If the data we want isn't in the cache, that's a ***cache miss***. We have to go to main memory, and then we'll save that data in the cache. By transferring entire blocks of data at a time, we take advantage of spatial locality.



Cache Review

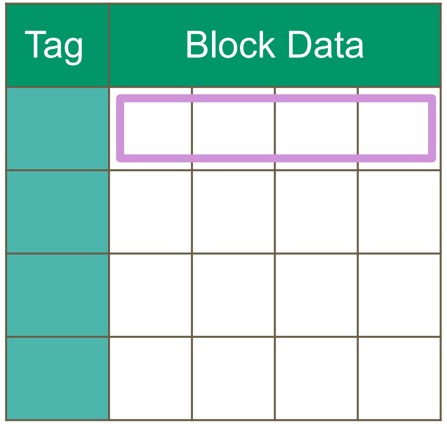
If the data we want is in the cache and valid, that's a ***cache hit***.

We don't go to memory, which saves us a lot of time!



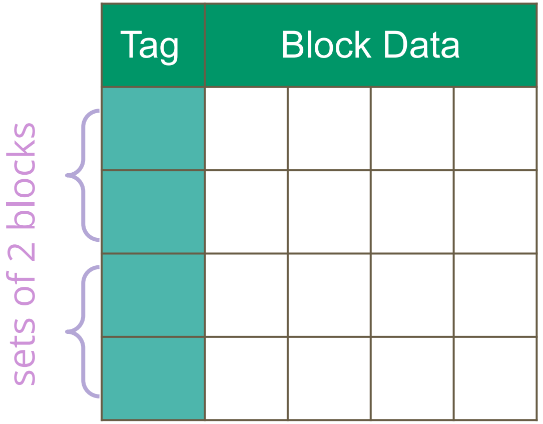
Cache Organization

- Caches have “slots” that each store *blocks* containing **Block Size** bytes of consecutive data.



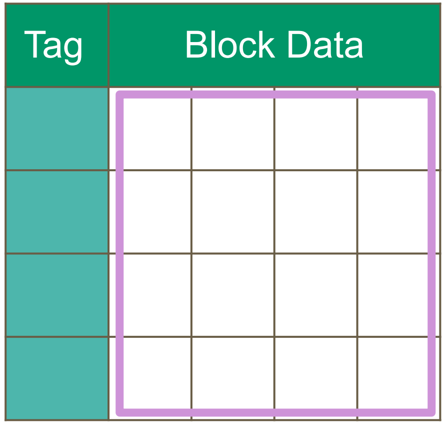
Cache Organization

- Caches have “slots” that each store *blocks* containing **Block Size** bytes of consecutive data.
- The **Associativity** is how many slots we group together in each *set*.



Cache Organization

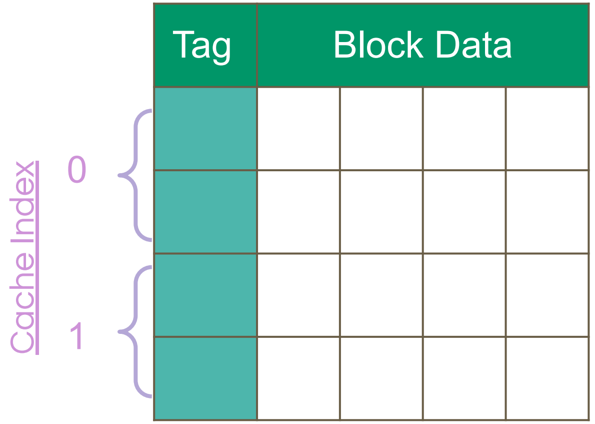
- Caches have “slots” that each store *blocks* containing **Block Size** bytes of consecutive data.
- The **Associativity** is how many slots we group together in each *set*.
- The total program data held in the cache (*i.e.*, the sum of all blocks) gives the **Cache Size**.



Analogy:

Cache Organization

- Caches have “slots” that each store *blocks* containing **Block Size** bytes of consecutive data.
- The **Associativity** is how many slots we group together in each *set*.
- The total program data held in the cache (*i.e.*, the sum of all blocks) gives the **Cache Size**.

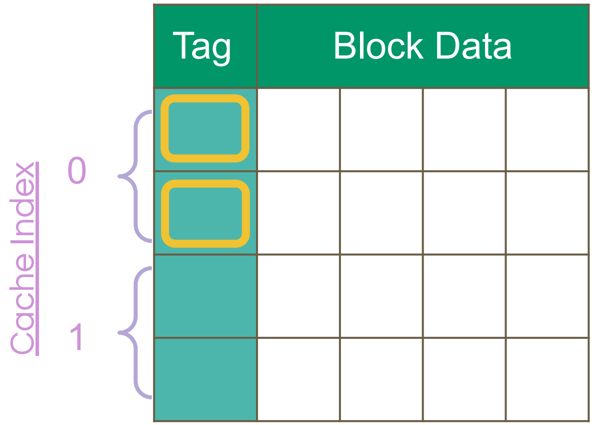


Accessing the Cache

- Each set is given a unique ***index*** encoding. Every address will map to a predetermined index; we will only search this set.

Cache Organization

- Caches have “slots” that each store *blocks* containing **Block Size** bytes of consecutive data.
- The **Associativity** is how many slots we group together in each *set*.
- The total program data held in the cache (*i.e.*, the sum of all blocks) gives the **Cache Size**.

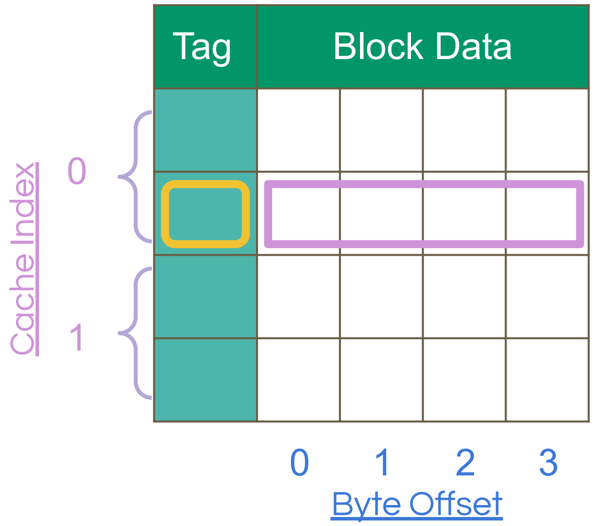


Accessing the Cache

- Each set is given a unique ***index*** encoding. Every address will map to a predetermined index; we will only search this set.
- Each block that maps to the same set can be uniquely identified by its ***tag***. We check for a tag match with each block in the set.

Cache Organization

- Caches have “slots” that each store *blocks* containing **Block Size** bytes of consecutive data.
- The **Associativity** is how many slots we group together in each *set*.
- The total program data held in the cache (*i.e.*, the sum of all blocks) gives the **Cache Size**.

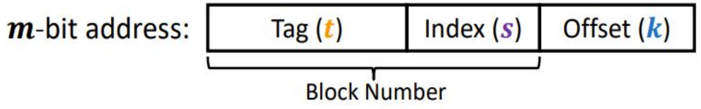


Accessing the Cache

- Each set is given a unique ***index*** encoding. Every address will map to a predetermined index; we will only search this set.
- Each block that maps to the same set can be uniquely identified by its ***tag***. We check for a tag match with each block in the set.
- The data’s starting position is given within the block by the ***offset*** byte numbering.

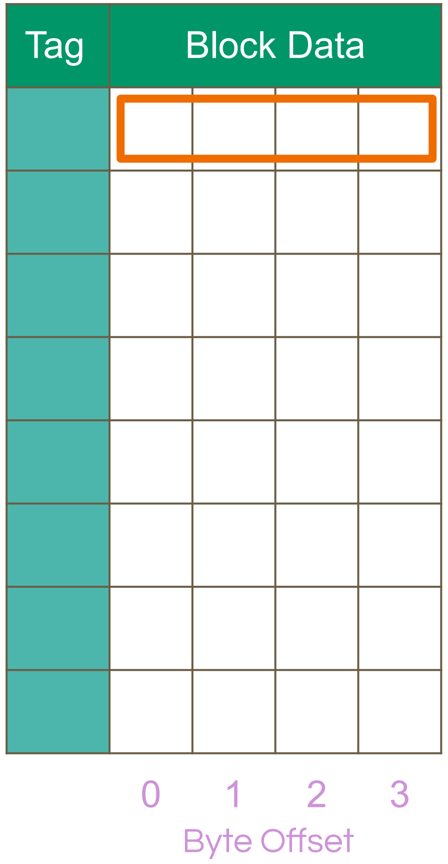
Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s



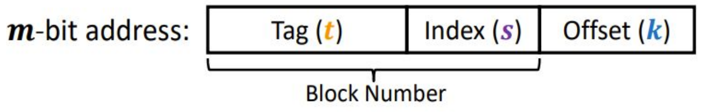
- Each block contains 4 bytes, so K = 4 bytes.

Example Cache

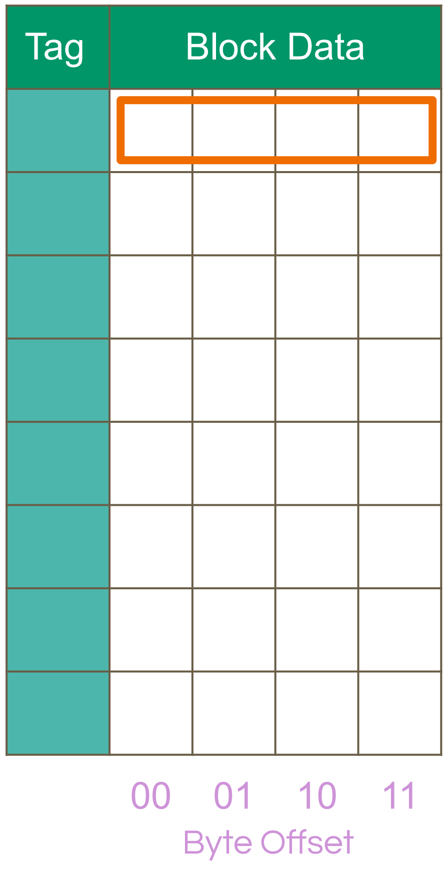


Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s

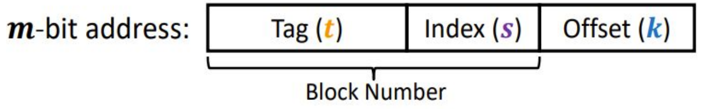


- Each block contains 4 bytes, so K = 4 bytes.
  - To label each byte, we need $k = \log_2(K) = 2$ bits.

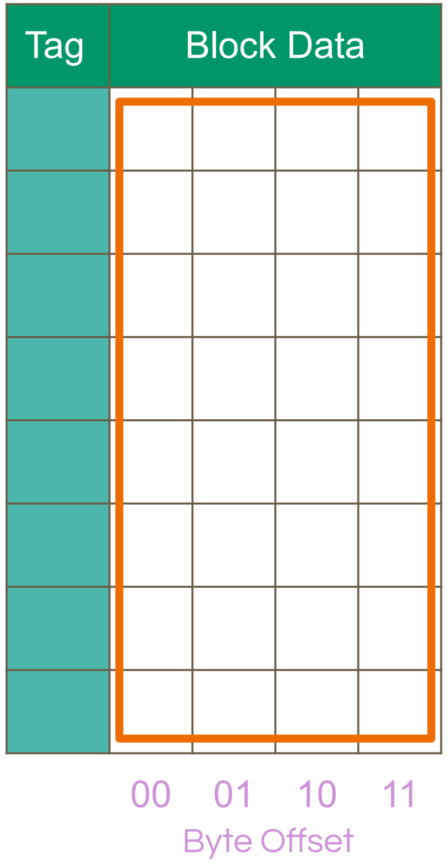


Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s

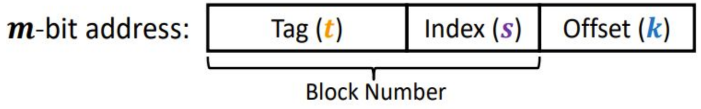


- Each block contains 4 bytes, so K = 4 bytes.
    - To label each byte, we need $k = \log_2(K) = 2$ bits.
- The cache size is (bytes in a block) * (number of blocks), so C = K * 8 = 32 bytes.

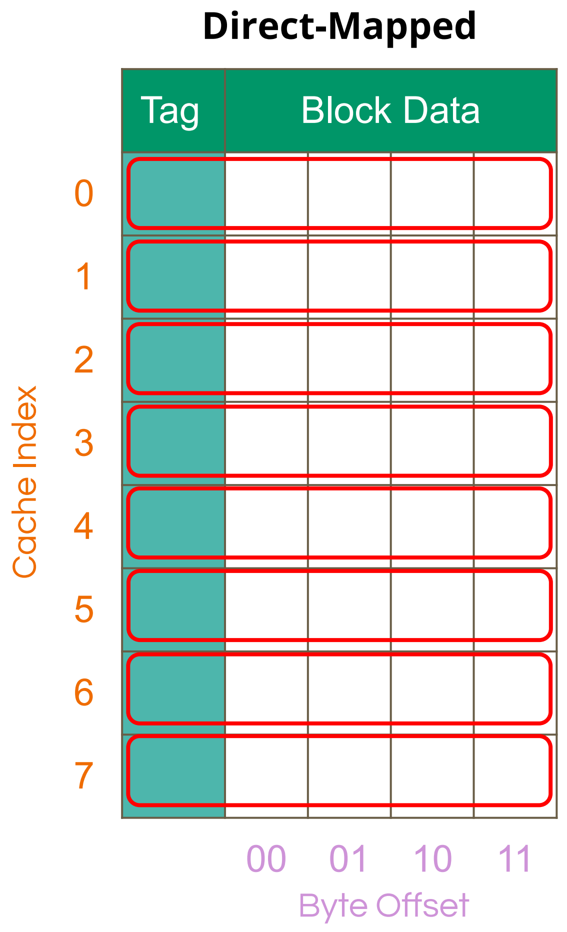


Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s

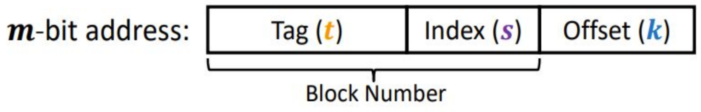


- Each block contains 4 bytes, so K = 4 bytes.
  - To label each byte, we need k = $\log_2(K)$ = 2 bits.
- The cache size is (bytes in a block) * (number of blocks), so C = K * 8 = 32 bytes.
- Assuming **Direct-Mapped**, E = 1 and S = (C/K)/E = 8 sets.



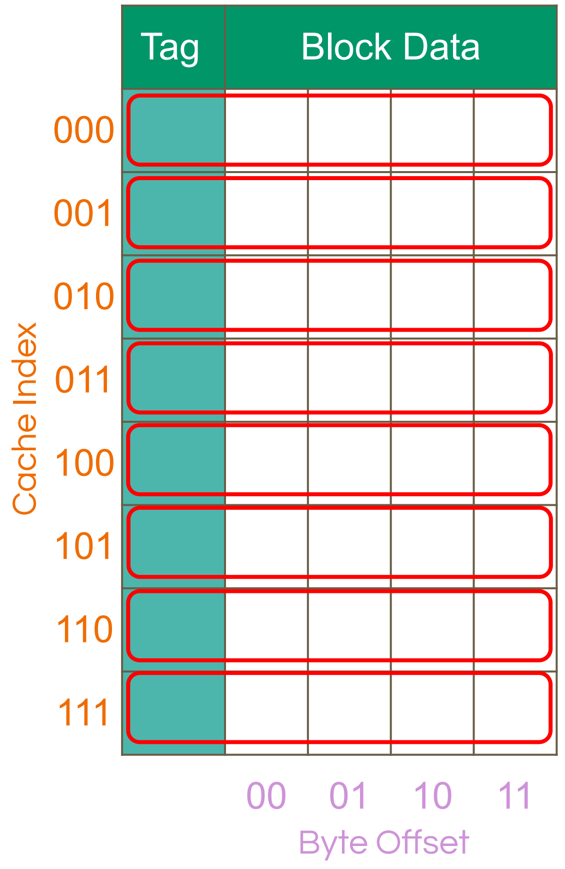
Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s



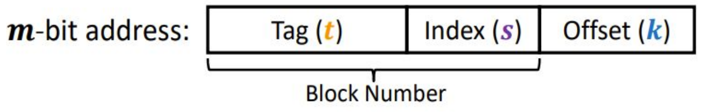
- Each block contains 4 bytes, so K = 4 bytes.
  - To label each byte, we need k = $\log_2(K)$ = 2 bits.
- The cache size is (bytes in a block) * (number of blocks), so C = K * 8 = 32 bytes.
- Assuming **Direct-Mapped**, E = 1 and S = (C/K)/E = 8 sets.
  - To label each slot, we need s = $\log_2(S)$ = 3 bits.

Direct-Mapped



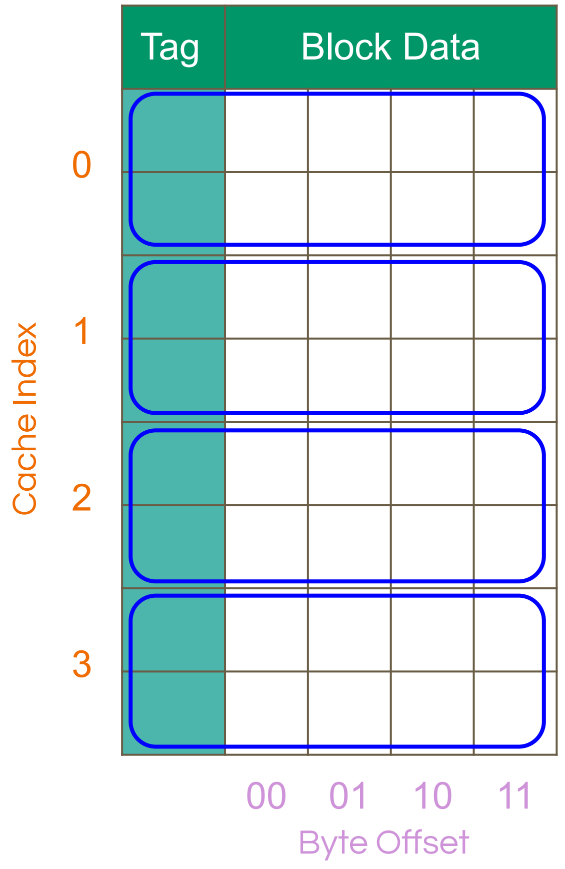
Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s



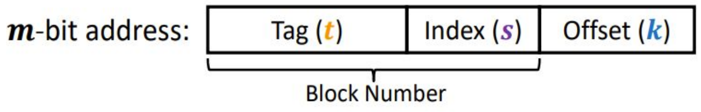
- Each block contains 4 bytes, so K = 4 bytes.
  - To label each byte, we need $k = \log_2(K) = 2$ bits.
- The cache size is (bytes in a block) * (number of blocks), so C = K * 8 = 32 bytes.
- Assuming **2-way Set Assoc**, E = 2 and S = (C/K)/E = 4 sets.

2-way Set Assoc

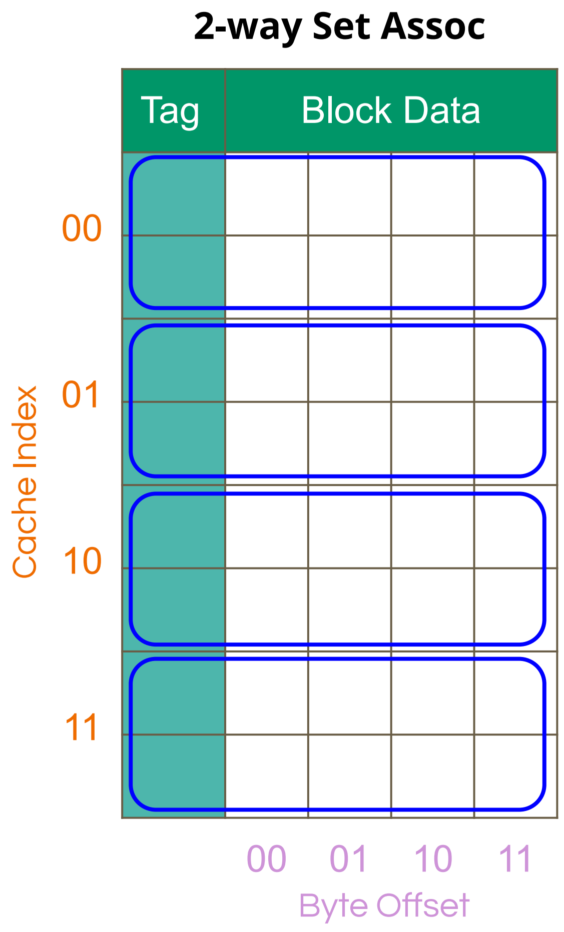


Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s

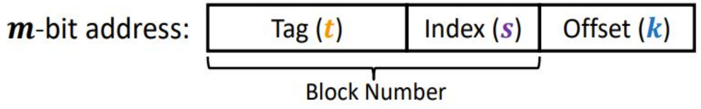


- Each block contains 4 bytes, so K = 4 bytes.
  - To label each byte, we need $k = \log_2(K) = 2$ bits.
- The cache size is (bytes in a block) * (number of blocks), so C = K * 8 = 32 bytes.
- Assuming **2-way Set Assoc**, E = 2 and S = (C/K)/E = 4 sets.
  - To label each slot, we need $s = \log_2(S) = 2$ bits.

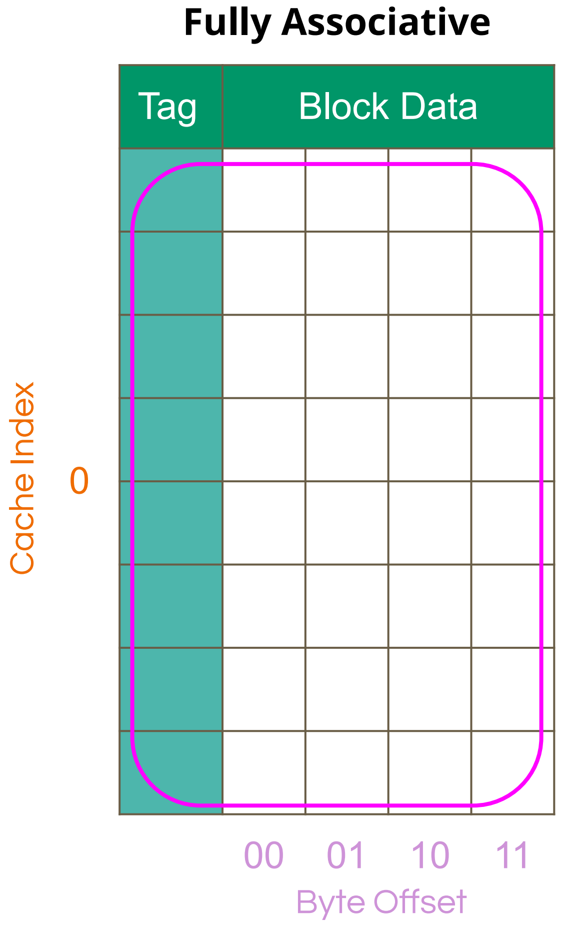


Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s

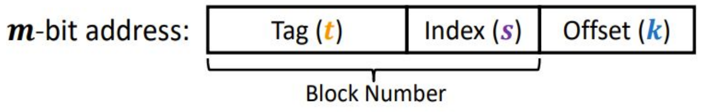


- Each block contains 4 bytes, so K = 4 bytes.
  - To label each byte, we need k = $\log_2(K)$ = 2 bits.
- The cache size is (bytes in a block) * (number of blocks), so C = K * 8 = 32 bytes.
- Assuming **Fully Associative**, E = 8 and S = (C/K)/E = 1 set.

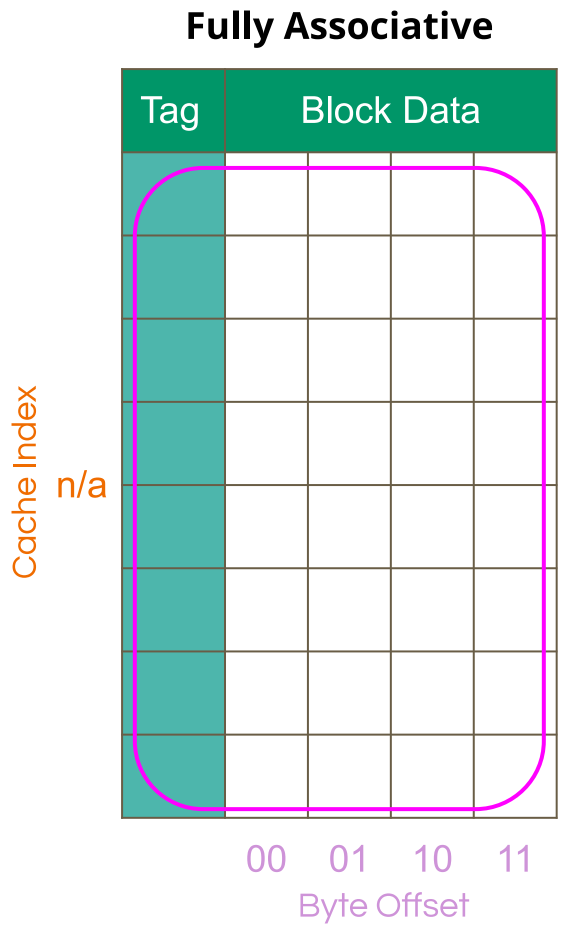


Cache Parameters

Symbol	Meaning
K	Block Size
C	Cache Size
E	Associativity
S	# Sets = (C / K) / E
m	Address Width
k	# Offset Bits = $\log_2(K)$
s	# Index Bits = $\log_2(S)$
t	# Tag Bits = m - k - s



- Each block contains 4 bytes, so K = 4 bytes.
  - To label each byte, we need k = $\log_2(K)$ = 2 bits.
- The cache size is (bytes in a block) * (number of blocks), so C = K * 8 = 32 bytes.
- Assuming **Fully Associative**, E = 8 and S = (C/K)/E = 1 set.
  - To label each slot, we need s = $\log_2(S)$ = 0 bits.



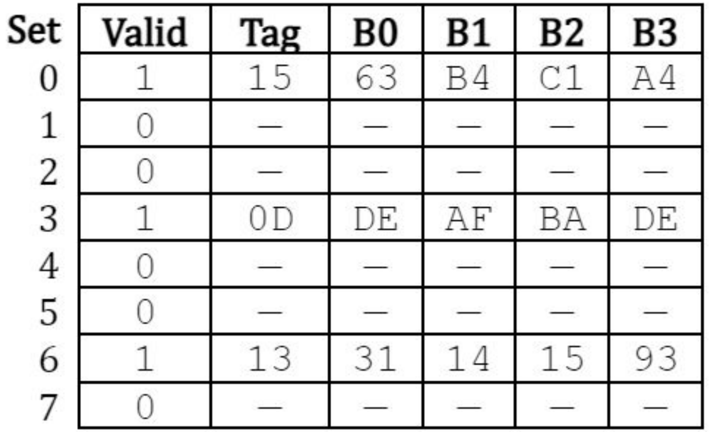
Exercises

Example

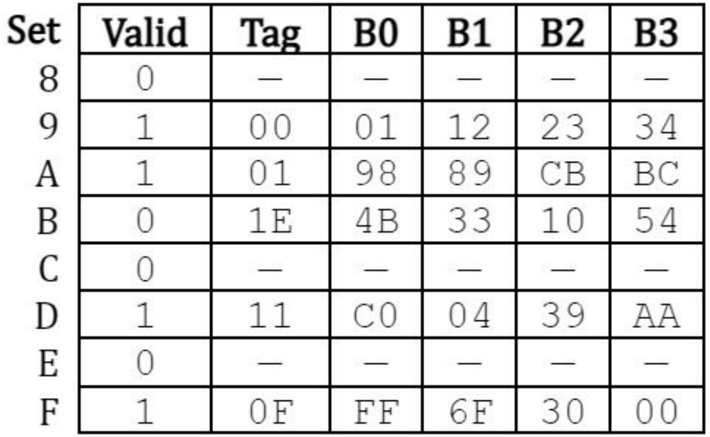
64 B capacity cache, 4 B block size, direct-mapped, 12 bit address length.

What's the TIO address breakdown?

- #bits for offset:
- #bits for index:
- #bits for tag:



Set	Valid	Tag	B0	B1	B2	B3
0	1	15	63	B4	C1	A4
1	0	—	—	—	—	—
2	0	—	—	—	—	—
3	1	0D	DE	AF	BA	DE
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	13	31	14	15	93
7	0	—	—	—	—	—



Set	Valid	Tag	B0	B1	B2	B3
8	0	—	—	—	—	—
9	1	00	01	12	23	34
A	1	01	98	89	CB	BC
B	0	1E	4B	33	10	54
C	0	—	—	—	—	—
D	1	11	C0	04	39	AA
E	0	—	—	—	—	—
F	1	0F	FF	6F	30	00

Example

Read 1 byte from address `0x024`

1. Translate to Binary:
   a. `0x024 =`
2. Split into TIO
   a. Tag =
   b. Index =
   c. Offset =

Set	Valid	Tag	B0	B1	B2	B3
0	1	15	63	B4	C1	A4
1	0	—	—	—	—	—
2	0	—	—	—	—	—
3	1	0D	DE	AF	BA	DE
4	0	—	—	—	—	—
5	0	—	—	—	—	—
6	1	13	31	14	15	93
7	0	—	—	—	—	—

Set	Valid	Tag	B0	B1	B2	B3
8	0	—	—	—	—	—
9	1	00	01	12	23	34
A	1	01	98	89	CB	BC
B	0	1E	4B	33	10	54
C	0	—	—	—	—	—
D	1	11	C0	04	39	AA
E	0	—	—	—	—	—
F	1	0F	FF	6F	30	00

Locality & Code Analysis

Temporal Locality

If your program used some data recently, it will likely **use it again** in the near future.

- **Examples: Loops**
  - Variables are likely be accessed multiple times.
  - Instructions are stored in memory too! Loops iterate over the same set of instructions in a short span of time.

```
char val = 0;

for (int i = 0; i < 8; i++)
    val += a[i];

for (i = 0; i < 8; i++)
    val ^= a[i];
```

- **Your Goal:** Make sure the data doesn't get kicked out of the cache in-between accesses.

Spatial Locality

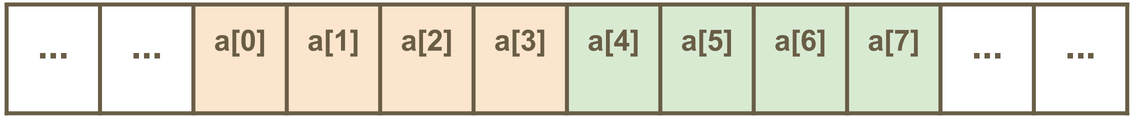
If your program used some data recently, it will likely **use nearby data** in memory in the near future.

- **Examples:**
  - Machine code for sequential instructions are placed next to each other in memory.
  - Arrays place neighboring elements in consecutive chunks of memory.

```
char val = 0;

for (int i = 0; i < 8; i++)
    val += a[i];

for (i = 0; i < 8; i++)
    val ^= a[i];
```



Sample array cut into two cache blocks in memory, ... is mystery data!

Spatial Locality

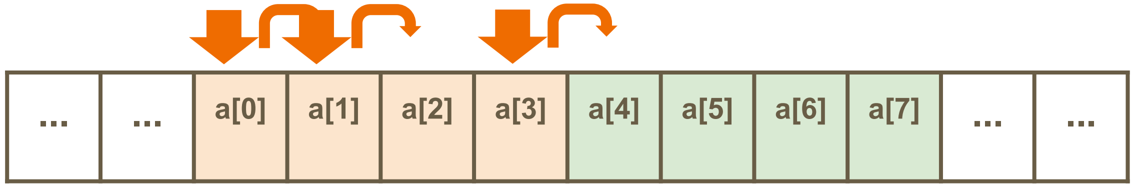
If your program used some data recently, it will likely use nearby data in memory in the near future.

- **Your Goal:** Use small strides/leaps when traversing data.
  - Notice that when the code accesses `a[i]`, it will next access `a[i+1]`, which is nearby!!
  - Array data can span multiple blocks, so even with stride-1, there will still be **cache misses**.

```
char val = 0;

for (int i = 0; i < 8; i++)
    val += a[i];

for (i = 0; i < 8; i++)
    val ^= a[i];
```



Sample array cut into two cache blocks in memory, ... is mystery data!

Miss Rate

The cache is mostly invisible to programmers. But we can still make some optimizations by keeping it in mind!

The *miss rate* is the ratio of cache misses to total memory accesses. If we can analyze when cache misses occur (usually by considering locality), we may be able to make our code more cache-friendly and improve performance.

Average Memory Access Time (AMAT) = (Hit Time) + (Miss Penalty)*(Miss Rate)

Example

What's the Miss Rate?

- First loop
  - Note array starts at beginning of a block
    - 0x600 -> 0b 011000 | 0000 | **00**
  - First access misses (cold cache)
    - Loads a[0] through a[3] into cache
    - a[1] through a[3] are hits

Index	Block Offset			
	00	**01**	**10**	**11**
00	🟥 `a[0]`	🟩 `a[1]`	🟩 `a[2]`	🟩 `a[3]`
01	?	?	?	?
10	?	?	?	?
...		...		

```
char val = 0;

for (int i = 0; i < 8; i++)
    val += a[i];

for (i = 0; i < 8; i++)
    val ^= a[i];
```

a is a `char` array of size 8.
Its address is 0x600, and the cache starts cold.
Assume i and `val` are stored in registers.

<u>Cache Parameters</u>

C = 64 bytes   K = 4 bytes   E = 1

What's the Miss Rate?

- First loop (continued)
  - Next miss on a[4]
    - Loads a[4] through a[7] into cache
    - a[5] through a[7] are hits
  - **8 accesses, 2 misses**

	Block Offset			
Index	00	01	10	11
00	a[0]	a[1]	a[2]	a[3]
01	a[4]	a[5]	a[6]	a[7]
10	?	?	?	?
...		...		

```
char val = 0;

for (int i = 0; i < 8; i++)
    val += a[i];

for (i = 0; i < 8; i++)
    val ^= a[i];
```

a is a `char` array of size 8.
Its address is 0x600, and the cache starts cold.
Assume i and `val` are stored in registers.

Cache Parameters

C = 64 bytes K = 4 bytes E = 1

What's the Miss Rate?

- Second loop
  - Entire array is still in the cache!
  - **8 accesses, 0 misses**
- Overall miss rate
  - 16 accesses, 2 misses
  - 2 / 16 = **12.5%**

	Block Offset			
Index	00	01	10	11
00	a[0]	a[1]	a[2]	a[3]
01	a[4]	a[5]	a[6]	a[7]
10	?	?	?	?
...	...			

```
char val = 0;

for (int i = 0; i < 8; i++)
    val += a[i];

for (i = 0; i < 8; i++)
    val ^= a[i];
```

a is a char array of size 8.
Its address is 0x600, and the cache starts cold.
Assume i and val are stored in registers.

Cache Parameters

C = 64 bytes   K = 4 bytes   E = 1

Cache Simulator

Cache Simulator!

Link:
https://courses.cs.washington.edu/courses/cse351/cachesim/

The cache simulator can be a helpful tool for reasoning through cache problems and mechanisms, particularly on homework and in lab 4.

That’s All, Folks!

Thanks for attending section! Feel free to stick around for a bit if you have quick questions (otherwise post on Ed or go to office hours).

See you all on Friday!