Caches IV


The cache when you ask for something that was just evicted:
I've never met this man in my life.

Code Analysis

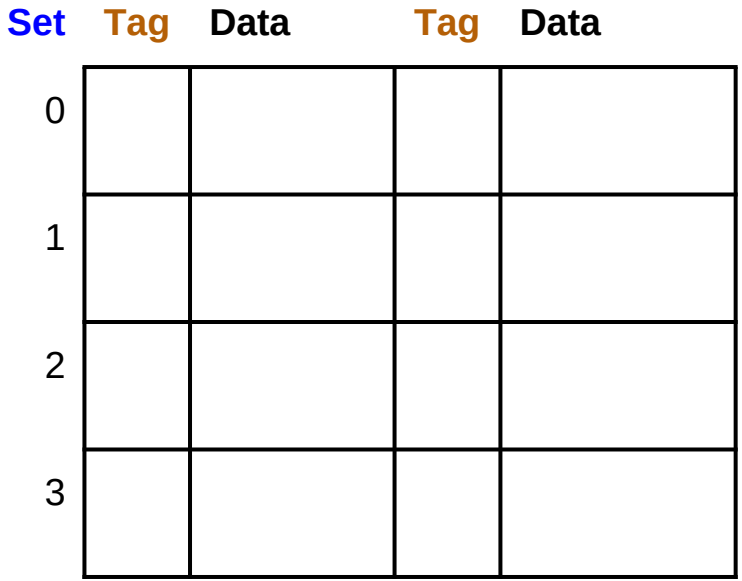
- Assuming cache starts **cold** (i.e. all blocks invalid), and `sum`, `i`, and `j` are all stored in registers, calculate the **miss rate**.
  - ***m*** = 10 bits, ***C*** = 64B, ***K*** = 8B, ***E*** = 2

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Code Analysis: relevant values

- ***m*** = 10 bits, ***C*** = 64B, ***K*** = 8B, ***E*** = 2
  - ***k*** = 3, ***s*** = 2
- 8B blocks = 4 `shorts` per block
- Starting address = `0b10000 00 000`
  - Block stored in set 0, tag = `0b10000` = `0x10`

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```



Code Analysis: step 1

`i = 0, j = 0`

Misses: 1
Hits: 0

- Access `ar[0][0]`
  - `0b10000 00 000`
    - Miss!
  - Load block into set 0 with tag 0x10

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]		
1				
2				
3				

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Code Analysis: step 2

`i = 0, j = 1`

Misses: 2
Hits: 0

- Access `ar[1][0]`
  - `0b10000 10 000`
    - Miss!
  - Load block into set 2 with tag 0x10

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]		
1				
2	10	a[1][0] … a[1][3]		
3				

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Code Analysis: step 3

`i = 0, j = 2`

Misses: 3
Hits: 0

- Access `ar[2][0]`
  - `0b10001 00 000`
    - Miss!
  - Load block into set 0 with tag 0x11
    - Can store both blocks in set 0 because of associativity

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]	11	a[2][0] … a[2][3]
1				
2	10	a[1][0] … a[1][3]		
3				

Code Analysis: step 4

`i = 0, j = 3`

Misses: 4
Hits: 0

- Access `ar[3][0]`
  - `0b10001 10 000`
    - Miss!
  - Load block into set 1 with tag 0x11

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]	11	a[2][0] … a[2][3]
1				
2	10	a[1][0] … a[1][3]	11	a[3][0] … a[3][3]
3				

Code Analysis: step 5

`i = 0, j = 4`

Misses: 5
Hits: 0

- Access `ar[4][0]`
  - `0b10010 00 000`
    - Miss!
  - Load block into set 0 with tag 0x12
    - Evicts least recently used block

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Set	Tag	Data	Tag	Data
0	12	a[4][0] … a[4][3]	11	a[2][0] … a[2][3]
1				
2	10	a[1][0] … a[1][3]	11	a[3][0] … a[3][3]
3				

Code Analysis: step 6-8

`i = 0, j = 5…`

Misses: 8
Hits: 0

- Same as step 5
  - Accesses to `a[5][0]`, `a[6][0]`, and `a[7][0]` will kick out the old blocks in the cache
- So for i = 0:
  - 8 accesses total (j = 0…7), 8 misses

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Set	Tag	Data	Tag	Data
0	12	a[4][0] … a[4][3]	13	a[6][0] … a[6][3]
1				
2	12	a[5][0] … a[5][3]	13	a[7][0] … a[7][3]
3				

Code Analysis: step 9

`i = 1, j = 0`

Misses: 9
Hits: 0

- Access `ar[0][1]`
  - `0b10000 00 010`
    - Same block that we loaded in in step 1, but it got evicted in step 5!
    - Miss!
  - Load block back into set 0

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Set	Tag	Data	Tag	Data
0	10	a[0][0] … a[0][3]	13	a[6][0] … a[6][3]
1				
2	12	a[5][0] … a[5][3]	13	a[7][0] … a[7][3]
3				

Code Analysis: step 10+

- All future accesses will continue to follow this pattern
  - Each block is loaded in, then kicked out of the cache before it's accessed again
- **Miss rate 100%!**
- How can we fix this?

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[j][i];
}
```

Improving Cache Performance

- Reduce **stride**
  - I.e. access data that's closer together

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[i][j];
}
```

Improving Cache Performance Example: step 1

Misses: 1
Hits: 3

- First 4 accesses:
  - `ar[0][0]`: miss, load block into the cache
  - `ar[0][1]`: hit!
  - `ar[0][2]`: hit!
  - `ar[0][3]`: hit!

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[i][j];
}
```

Set	Tag	Data	Tag	Data
0	10	ar[0][0] … ar[0][3]		
1				
2				
3				

Improving Cache Performance Example: step 2

Misses: 2
Hits: 6

- Next 4 accesses:
  - `ar[0][4]`: miss, load block into the cache
  - `ar[0][5]`: hit!
  - `ar[0][6]`: hit!
  - `ar[0][7]`: hit!

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[i][j];
}
```

Set	Tag	Data	Tag	Data
0	10	ar[0][0] … ar[0][3]		
1	10	ar[0][4] … ar[0][7]		
2				
3				

Improving Cache Performance Example: step 3+

- All accesses follow this pattern
  - Because we use a whole block before moving on, we will miss 1 out of every 4 accesses
- **Miss rate: 25%**

```
#define SIZE 8
short ar[SIZE][SIZE], sum = 0; // &ar=0x200
for (int i = 0; i < SIZE; i++) {
    for (int j = 0; j < SIZE; j++)
        sum += ar[i][j];
}
```

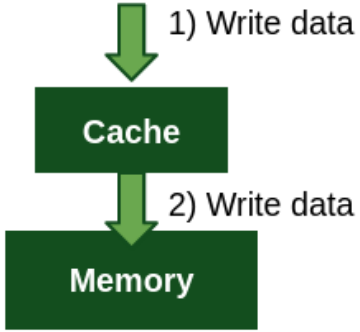
Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- **Cache organization**
  - Direct-mapped (sets; index + tag)
  - Associativity (ways)
  - Replacement policy
  - **Handling writes**
- Program optimizations that consider caches

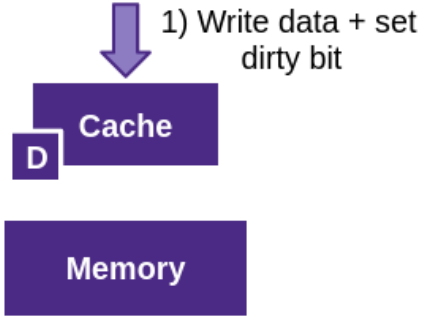
Write-Hit Policies

What to do if the data is already in the cache?

- **Write-through**: immediately write to the next level
- **Write-back**: don't write to next level until we have to
  - Keep track of **dirty bit** for each block
  - On eviction, if dirty bit is set, write contents back to memory



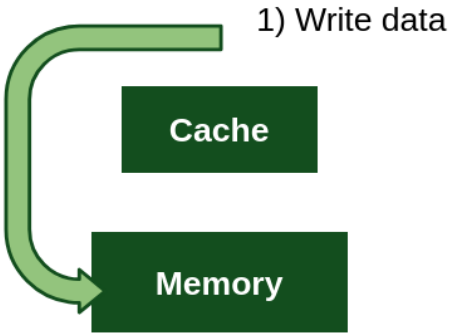
Write-through



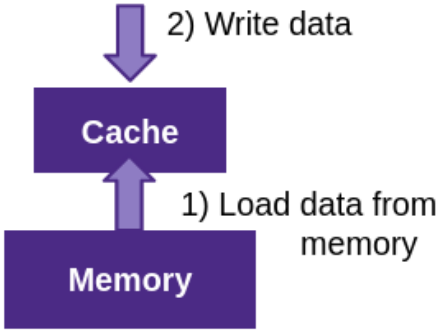
Write-back

Write-Miss Policies

- What to do if the block we want to write to is not in the cache?
- **No-write-allocate** (“write around”): don’t load into the cache, just write to the next level
- **Write-allocate** (“fetch on write”) load data into the cache before writing



No-write-allocate

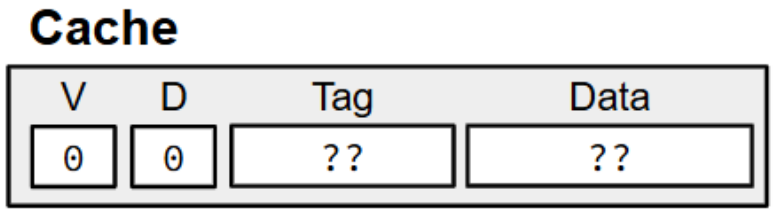


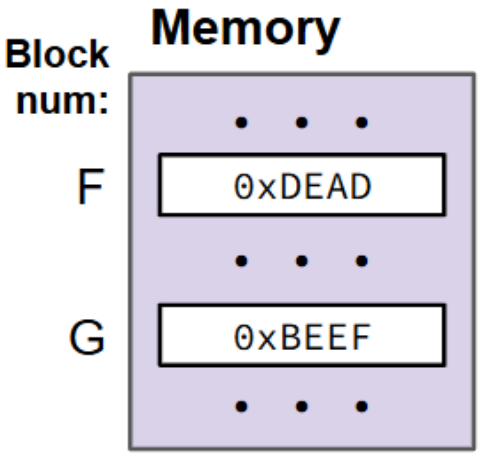
Write-allocate

Ex: Write-Back, Write-Allocate

- Single-block mini cache
  - Tag includes the entire block number
  - *Not a realistic example*

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache





Ex: Write-Back, Write-Allocate (pt 2)

1. `mov $0xB0BA, (F)`

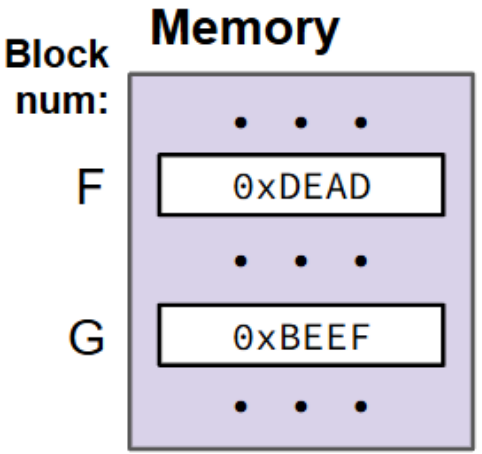
Write miss

Not valid x86. Assume we mean an address within block F.

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

Cache

V	D	Tag	Data
0	0	??	??



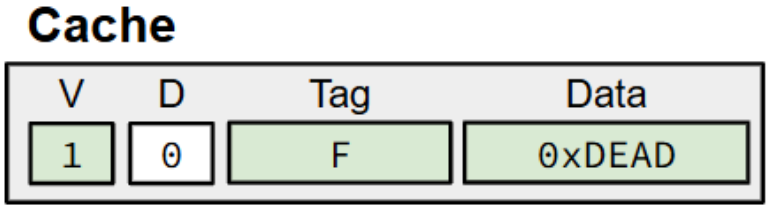
Ex: Write-Back, Write-Allocate (pt 3)

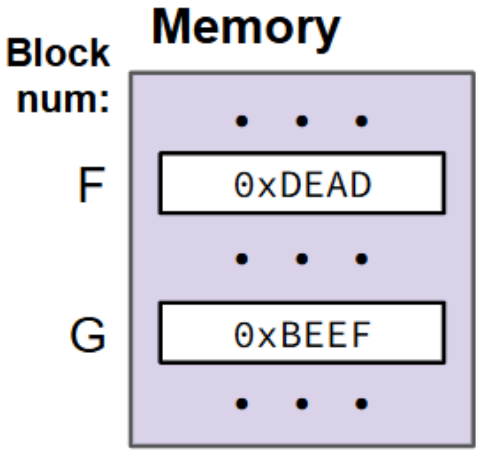
Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

1. `mov $0xB0BA, (F)`

Write miss

1. Bring F into cache





Ex: Write-Back, Write-Allocate (pt 4)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

1. `mov $0xB0BA, (F)`

Write miss

1. Bring F into cache

Cache

V	D	Tag	Data
1	1	F	0xB0BA

Memory

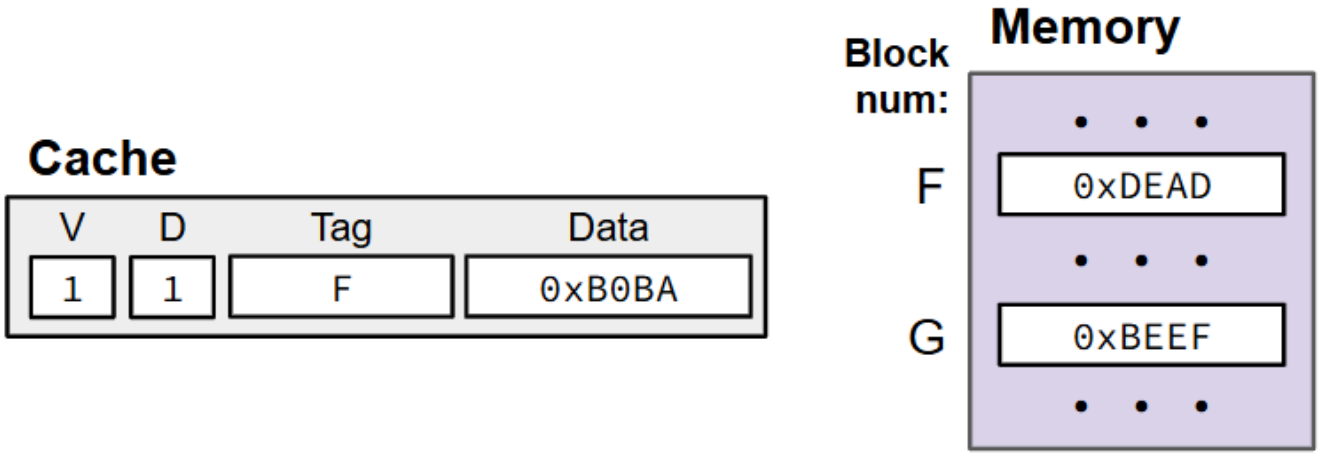
Block num:	
	• • •
F	0xDEAD
	• • •
G	0xBEEF
	• • •

Ex: Write-Back, Write-Allocate (pt 5)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

2. `mov $0xF00D, (F)`

Write hit



Ex: Write-Back, Write-Allocate (pt 6)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

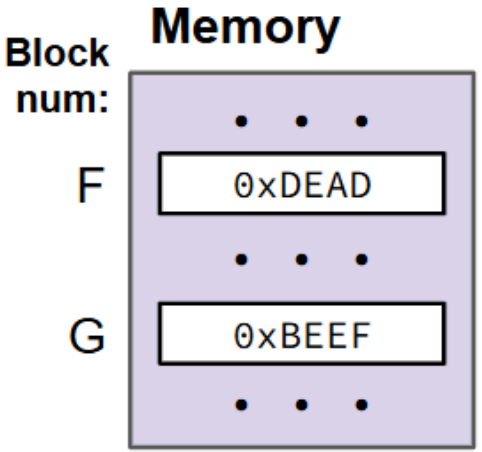
2. `mov $0xF00D, (F)`

Write hit

1. Write `0xF00D` into cache *only*
   a. Set dirty bit

Cache

V	D	Tag	Data
1	1	F	0xF00D

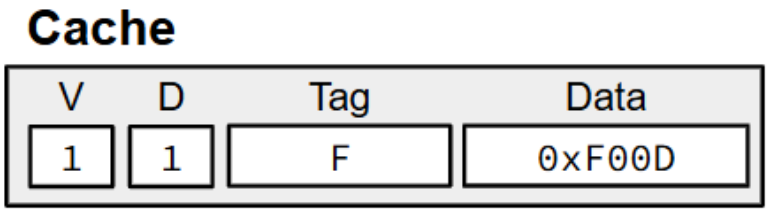


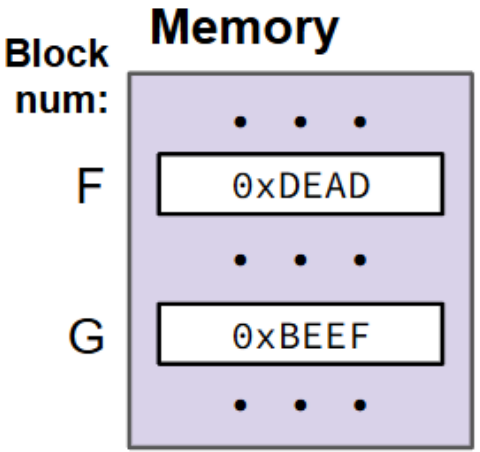
Ex: Write-Back, Write-Allocate (pt 7)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

3. `mov (G), %ax`

Read miss





Ex: Write-Back, Write-Allocate (pt 8)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

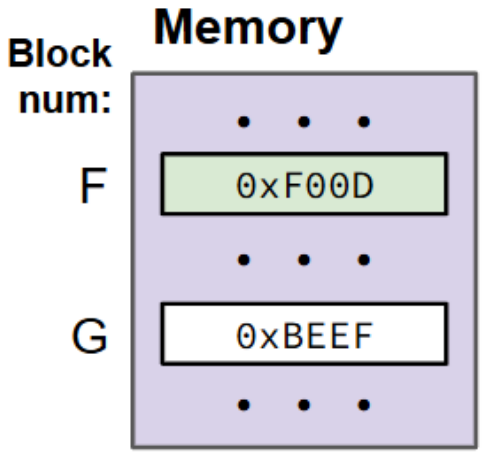
3. `mov (G), %ax`

Read miss

1. Write F back to memory since it is dirty

Cache

V	D	Tag	Data
1	1	F	0xF00D



Ex: Write-Back, Write-Allocate (pt 9)

Write-back: defer write to next level until line is evicted
Write-allocate: on a miss, bring the data into cache

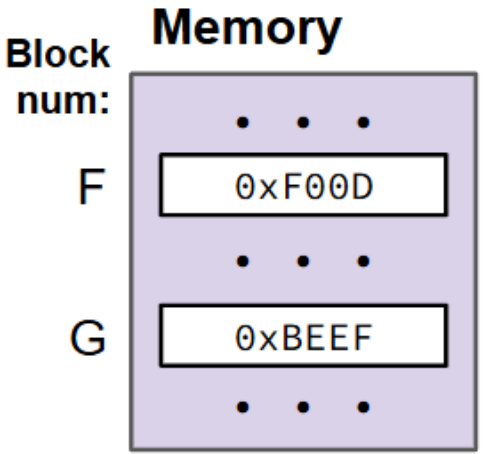
3. `mov (G), %ax`

Read miss

1. Write F back to memory since it is dirty
2. Bring G into cache

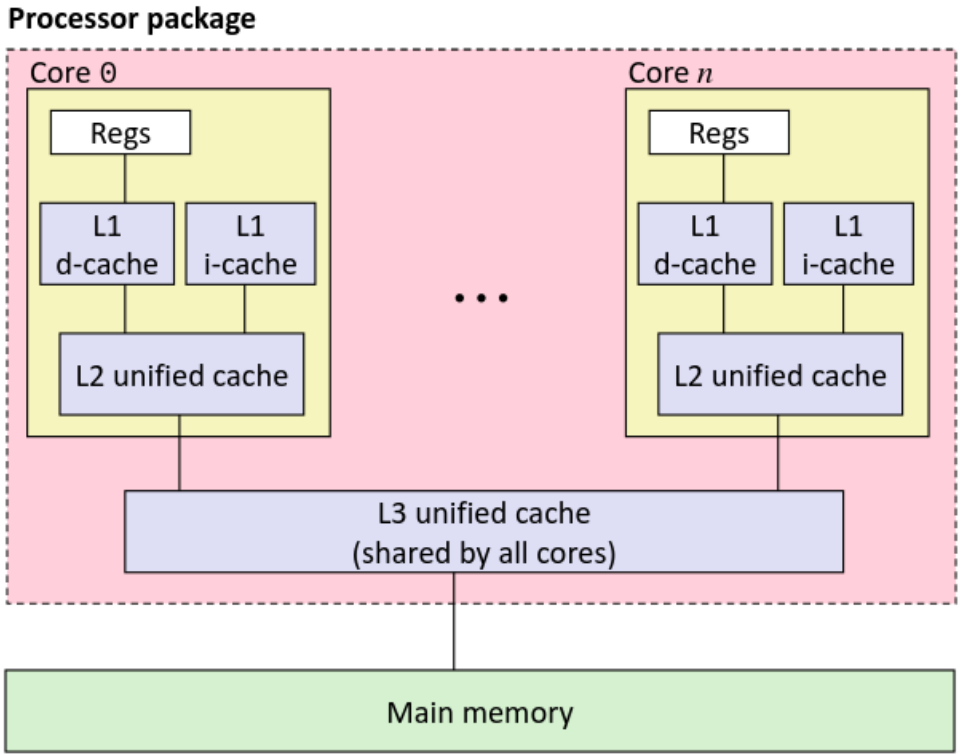
Cache

V	D	Tag	Data
1	0	G	0xBEEF



Common Policies

- **Write-back + Write-allocate**
  - (most common)
- Write-through + No-write-allocate
  - When would this be used?



Caches

- Cache basics
- Principle of locality
- Memory hierarchies
- Cache organization
  - Direct-mapped (sets; index + tag)
  - Associativity (ways)
  - Replacement policy
  - Handling writes
- **Program optimizations that consider caches**

Optimizations for the Memory Hierarchy

- Write code that has locality
  - **Spatial**: access data contiguously
  - **Temporal**: make sure access to the same data is not too far apart in time
- How can you achieve locality?
  - Adjust memory accesses in code to improve miss rate (MR)
    - Requires knowledge of **both** how caches work as well as your system's parameters
  - Proper choice of algorithm
  - Loop transformations