Introduction to caching

Kevin Wang

UNIVERSITY OF MICHIGAN

7 December 2023

I assume you already know…

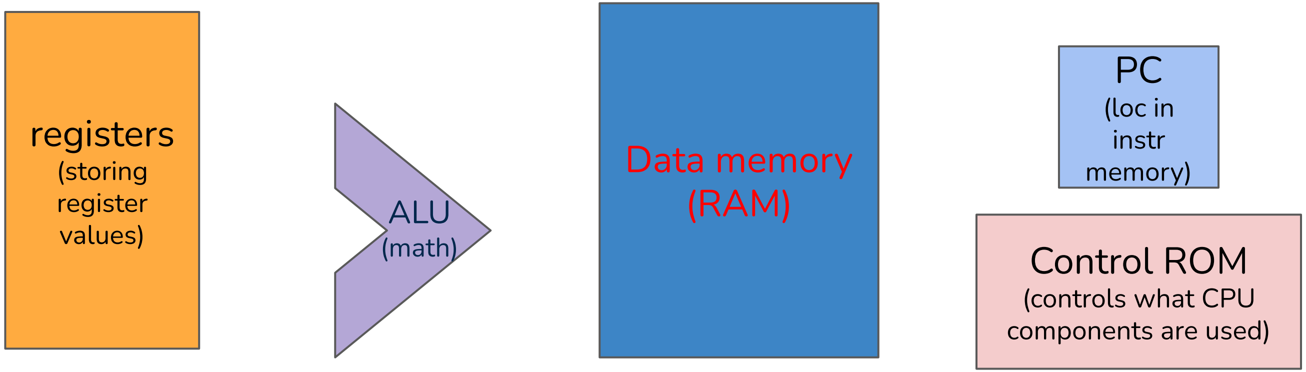
- …the basics of how memory in a CPU works
  - i.e. what happens on a load / store
- …what components in the CPU handle memory operations
  - RAM, registers, and maybe you've seen caches already
- …how to read hexadecimal :)

- There will be a brief ungraded assignment after this to help you practice the concepts

Motivation and basics

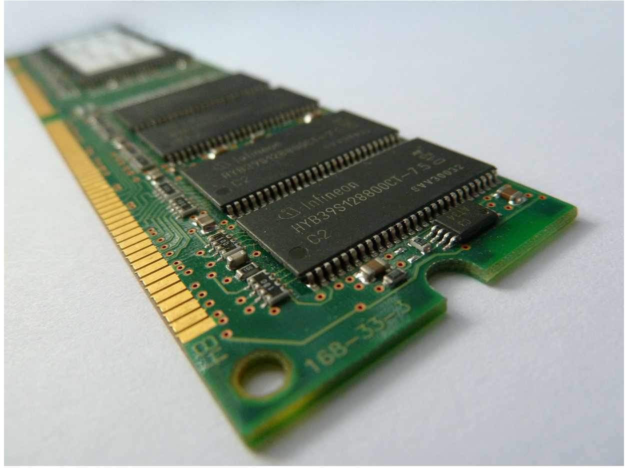
Memory operations are slow

Loading and storing are typically the slowest operations in a CPU (compared to register operations / ALU)



However, most programs will use a lot of memory operations

Is there a way to make memory faster?



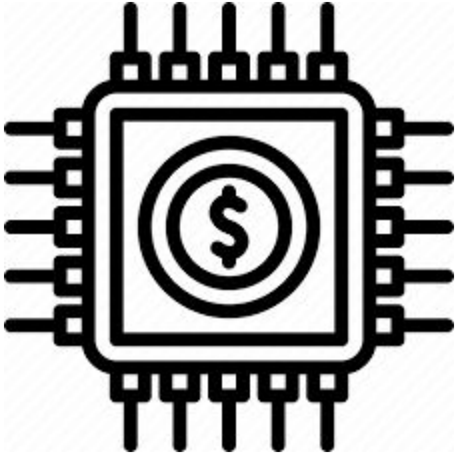
Solution 1: say no, memory is always slow

…well, that's boring

Solution 2: add more registers

This could work…

…but registers are very expensive and can't store lots of data



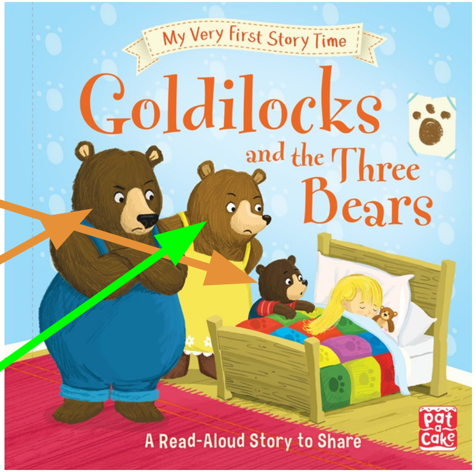
Solution 3: use something in between

The **cache** is a memory component that is much quicker to access than RAM, but a little slower than registers

registers (too small)

RAM (too slow)

cache (just right!)



Computer Memory Hierarchy

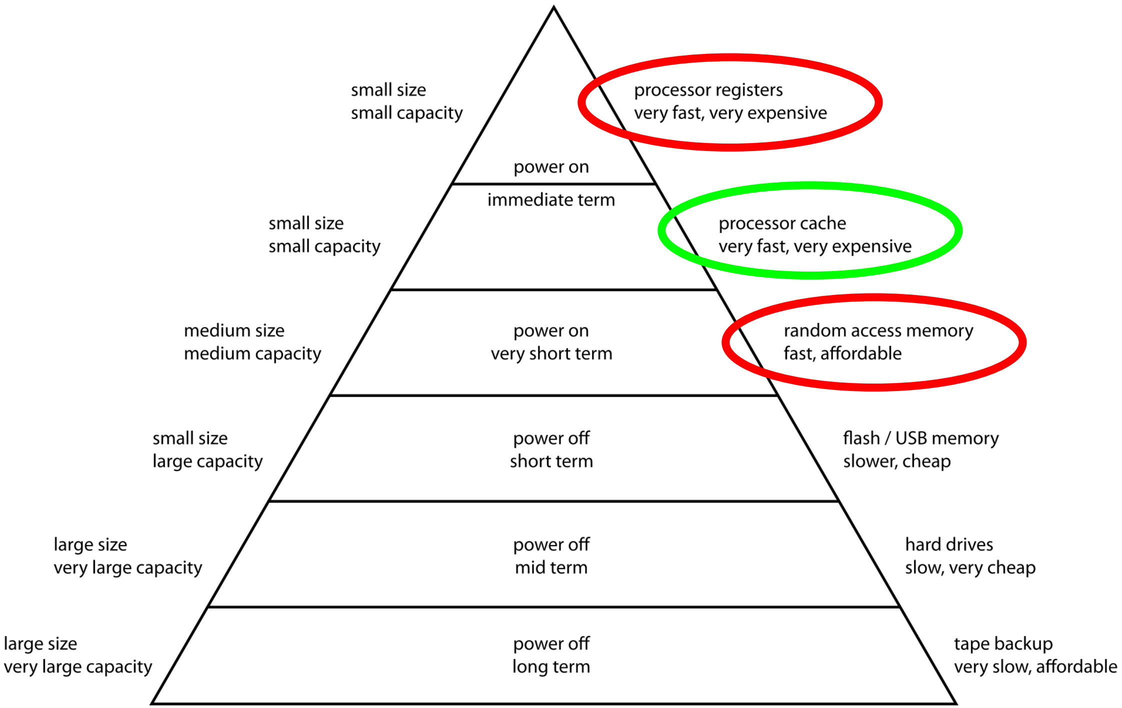


Diagram of the basic idea

Next instruction is a memory access

```
...
add     1    2    3      ⇐
load    4    0    addr
...
load    5    0    addr
```

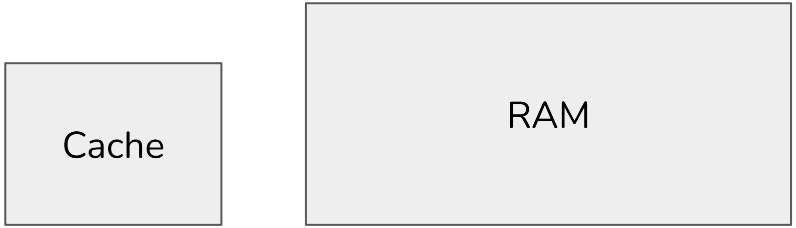


Diagram of the basic idea

When we do a memory access, we check if it's already in the cache

```
...
add     1    2    3
load    4    0    addr  ⇐
...
load    5    0    addr
```

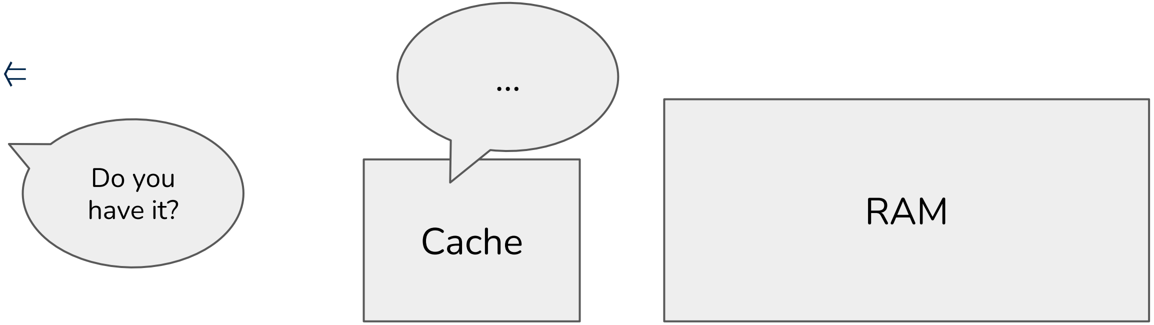
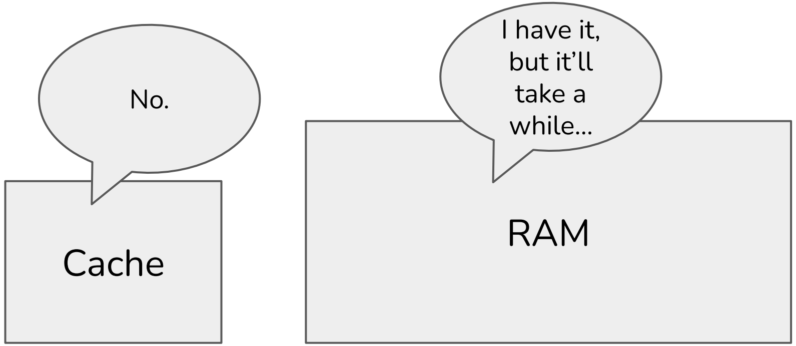


Diagram of the basic idea

Here, we have a **cache miss** because the data we are accessing was not in the cache already

We perform the memory operation and also put some data into the cache

```
...
add    1   2   3
load   4   0   addr  ⇐
...
load   5   0   addr
```



UNIVERSITY OF MICHIGAN

Diagram of the basic idea

When we do a memory access, we check if it’s already in the cache

```
...
add     1    2    3
load    4    0    addr
...
load    5    0    addr
```

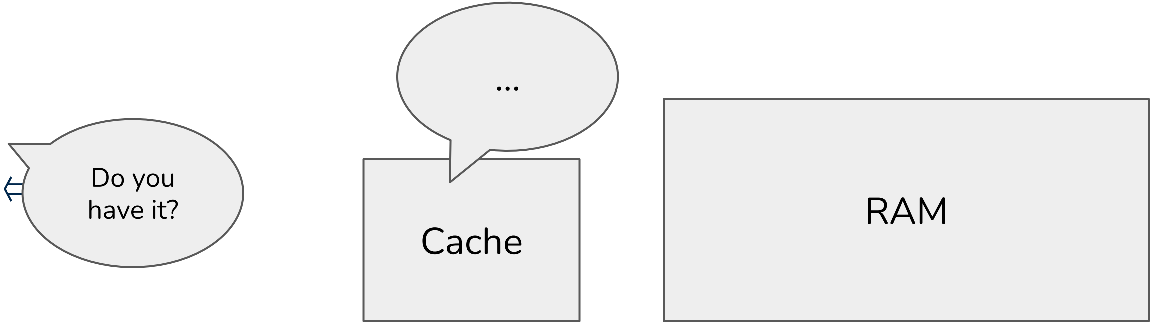
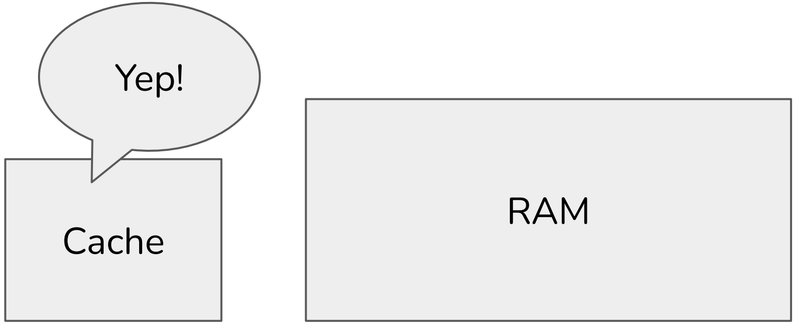


Diagram of the basic idea

Here, we have a **cache hit** because the data we are accessing is in the cache already

We perform the memory operation, but use cache latency instead of RAM latency

```
...
add     1   2   3
load    4   0   addr
...
load    5   0   addr  ⇐
```



UNIVERSITY OF MICHIGAN

Details of cache design

Cache is divided into **cache blocks / lines**

- Every block in the cache corresponds to a block in memory
  - Memory block = contiguous range of addresses, e.g. 0x1000 - 0x100F is a 16-byte block
  - We can choose the size of our blocks
- The more blocks in the cache, the more data we can store
  - Which also makes it so that we can hit more frequently!

represents one cache line

Ideally, we want to avoid cache misses as much as possible

- If we miss too frequently, caching doesn't help a lot
- We are happy if we find the design that misses the least on most programs

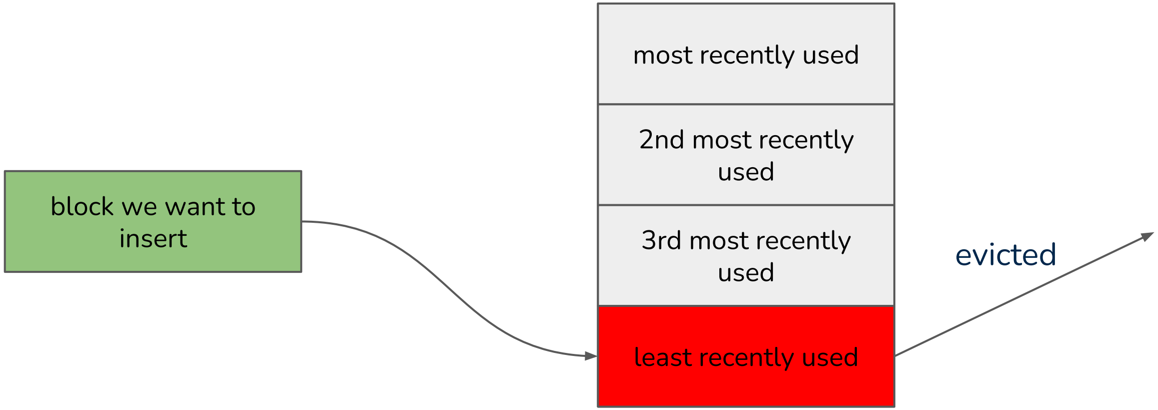
How do misses happen?

- Compulsory miss: has to happen
  - The cache cannot hit on a memory block that has not been seen before!
- Capacity miss: happens because the cache is full at some point
  - More on this next!
- Conflict miss: happens because the cache is not “associative” enough
  - More on this later….

What happens when the cache is full?

Typical way to solve this is to **evict** (remove) the least recently used block

Idea: if it hasn't been used recently, then it won't be used anytime soon



With two 8B blocks, this code will cache miss every two accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```

[EMPTY]
[EMPTY]

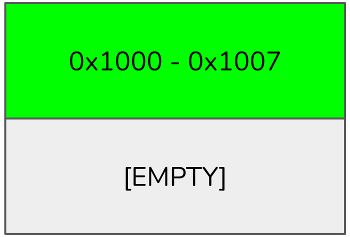
With two 8B blocks, this code will cache miss every two accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```

0x1000 - 0x1007
[EMPTY]

With two 8B blocks, this code will cache miss every two accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



With two 8B blocks, this code will cache miss every two accesses

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```

0x1000 - 0x1007
0x1008 - 0x100F

With two 8B blocks, this code will cache miss every two accesses

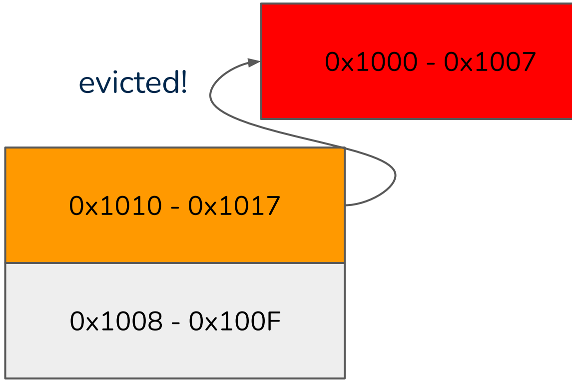
```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```

0x1000 - 0x1007
0x1008 - 0x100F

UNIVERSITY OF MICHIGAN

With two 8B blocks, this code will cache miss every two accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



With two 8B blocks, this code will cache miss every two accesses

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```

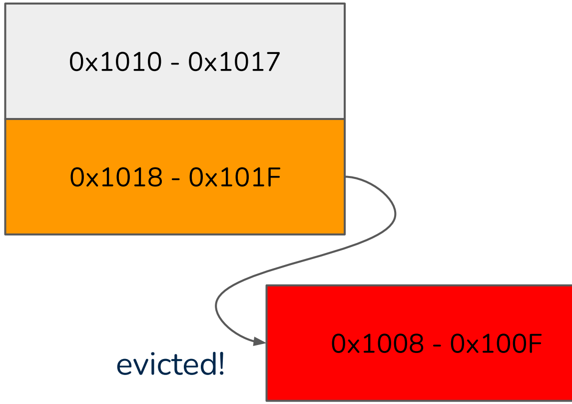
0x1010 - 0x1017

0x1008 - 0x100F

UNIVERSITY OF MICHIGAN

With two 8B blocks, this code will cache miss every two accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



With two 8B blocks, this code will cache miss every two accesses

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```

0x1010 - 0x1017
0x1018 - 0x101F

What happens if we increase the block size?

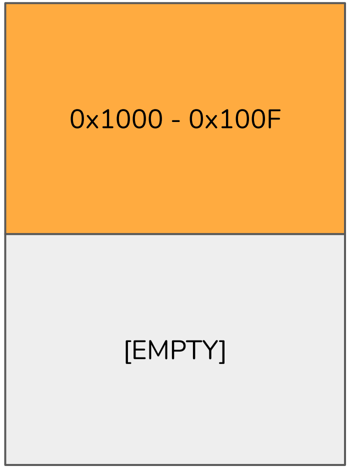
With two 32B blocks, this code will cache miss once every four accesses

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```

[EMPTY]
[EMPTY]

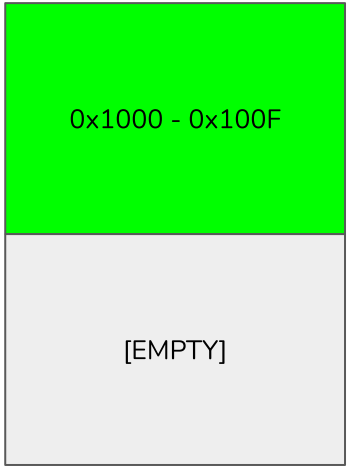
With two 32B blocks, this code will cache miss once every four accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



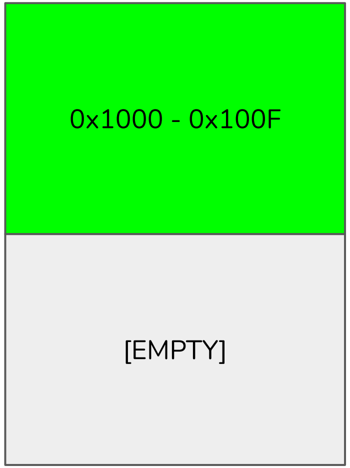
With two 32B blocks, this code will cache miss once every four accesses

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



With two 32B blocks, this code will cache miss once every four accesses

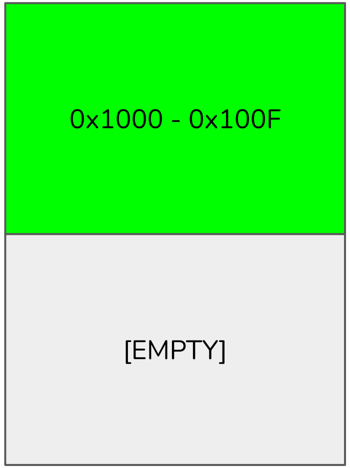
```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



UNIVERSITY OF MICHIGAN

With two 32B blocks, this code will cache miss once every four accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



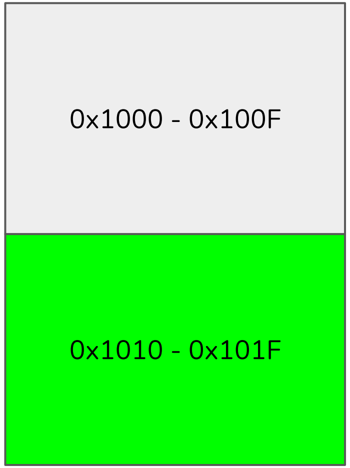
With two 32B blocks, this code will cache miss once every four accesses

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```

0x1000 - 0x100F
0x1010 - 0x101F

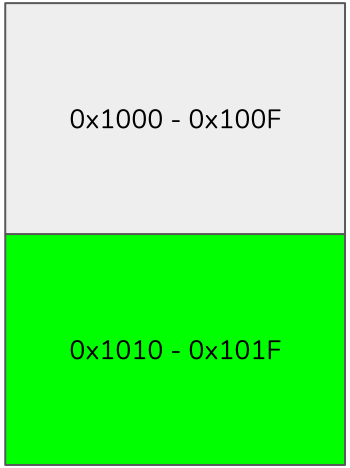
With two 32B blocks, this code will cache miss once every four accesses

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



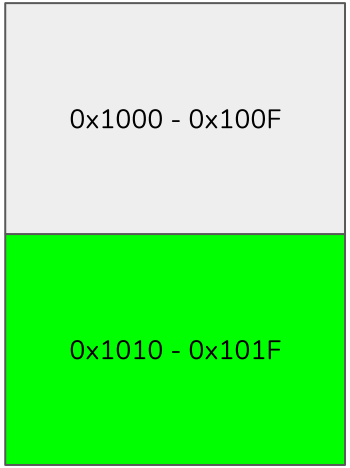
With two 32B blocks, this code will cache miss once every four accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



With two 32B blocks, this code will cache miss once every four accesses

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



Increasing block size reduces **<u>compulsory misses</u>** because memory is divided into less blocks.

These examples aren’t really taking advantage of caching, though

- Caches store memory so that it can be used again quickly
  - But our example never reuses accessed data!
- To use this, we should be accessing the same memory multiple times

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; ++inner) {
        arr[inner] += 1;
    }
}
```

Consider a small cache: two 4B blocks

What will happen?

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; ++inner) {
        arr[inner] += 1;
    }
}
```

Answer: all blocks will always be evicted before being reused again

On this program, the cache always misses!

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; ++inner) {
        arr[inner] += 1;
    }
}
```

How about this cache?

Increase number of blocks to 8

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; ++inner) {
        arr[inner] += 1;
    }
}
```

Answer: no block gets evicted because we always have space now

Cache will hit every time after compulsory misses

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; ++inner) {
        arr[inner] += 1;
    }
}
```

Increasing number of blocks reduces **capacity misses** because the cache can hold more previously accessed memory.

Is the best cache just a really big cache?

First, let's note that small differences in cost matter a lot

- If you create 100 million ICs, a $0.20 difference in cost for one IC becomes a $20 million total difference for all ICs
    - 427 billion ICs shipped worldwide in 2022
- You can solve a lot of problems by adding more wires / other hardware, but that's not necessarily a *good* solution

How do we know which blocks are in the cache?

- We can’t store this information for free
  - **Overhead:** additional information about the data stored in the cache
  - Overhead takes space ⇒ costs additional money
- We have to know what range of addresses is represented by one block
  - This introduces overhead!

<u>Tag:</u> part of the address used to identify memory block in cache

- Tag is calculated as follows (for block size of 16B)

$$0b\ \underbrace{0001\ 0000\ 0000}_{\text{Tag bits}}\ \underbrace{1101}_{\substack{\text{Block offset (not part in tag,}\\ \text{based on block size)}}}$$

- It can help to write the address in binary instead of hex
- $\log_2$(block size) is the number of block offset bits
  - These are the bits at the end, and they tell us where “in” the block we want to access
- Tag bits are just the remaining bits

What is the tag for this address...

- ...if the block size is 4B?
- ...if the block size is 16B?
- ...if the block size is 64B?

```
0b 0001 0000 0000 1101
```

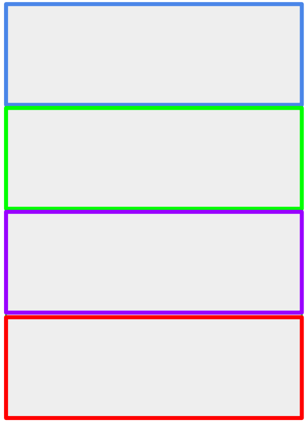
How can we reduce overhead from tag?

- Remember, all these additional bits cost money!
- By introducing **cache sets**
  - In cache design, a set is a group of blocks
- Blocks from memory belong to exactly one set
  - This is predetermined by the starting address of the block
- Address is now split into tag, set index, and block offset

$$\texttt{0b}\ \underbrace{\texttt{0001 0000 00}}_{\text{Tag bits}}\underbrace{\texttt{00}}_{\text{Set index (for 4 sets)}}\ \underbrace{\texttt{1101}}_{\text{Block offset (for block size 16B)}}$$

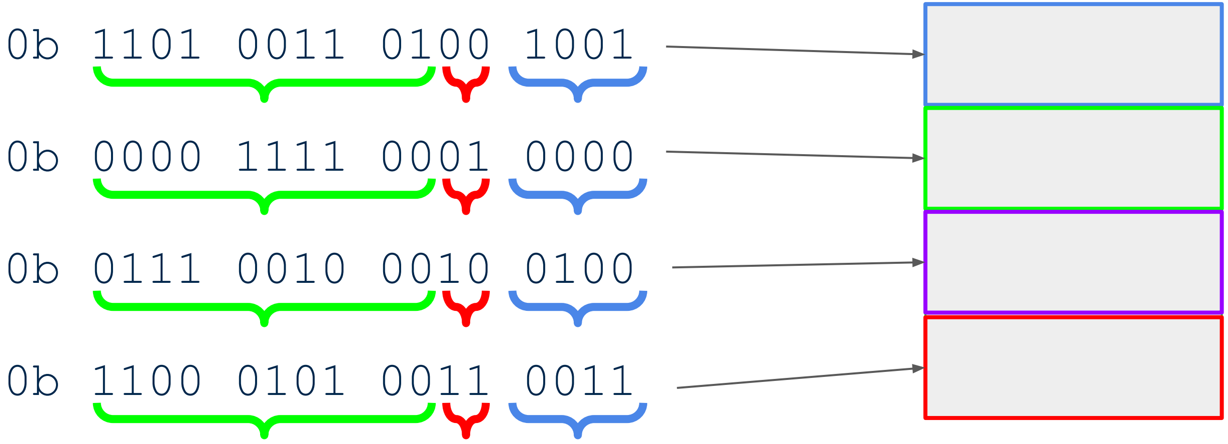
Extreme case: direct-mapped cache

- Every cache line is its own set
- This cache has 4 blocks, 4 sets



To determine which set a block goes into, look at the set bits

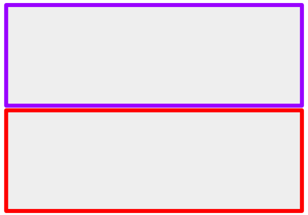
- Suppose block size is 16B



Where does this address belong?

Consider a 2-block direct mapped cache with 8 B block size

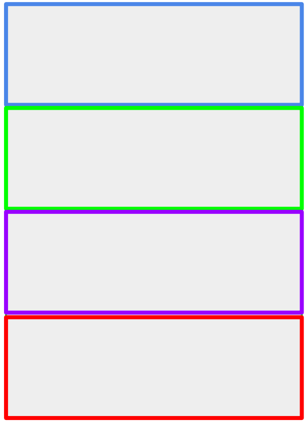
`0b 0101 0110 1011 0001`



If we run this program on this cache design, we see more or less the same behavior as a cache without sets

assuming 8B block size

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {2, 4, 6, 8, 10, 12, 14, 16};
for (int i = 0; i < 8; ++i) {
    arr[i] += 1;
}
```



Why is this? (Do any of the blocks map to the same set?)

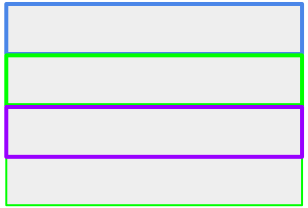
Let's change the code slightly

```
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {100, 0, 100, 0, 100, 0, 100, 0};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; inner += 2) {
        arr[inner] += 1;
    }
}
```

UNIVERSITY OF MICHIGAN

Let's use a different cache that is 4B block size

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {100, 0, 100, 0, 100, 0, 100, 0};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; inner += 2) {
        arr[inner] += 1;
    }
}
```



Without sets, how many misses / hits will we have?

Let's look at which addresses we're accessing

```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {100, 0, 100, 0, 100, 0, 100, 0};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; inner += 2) {
        arr[inner] += 1;
    }
}
```

0b 0001 0000 0000 0000

0b 0001 0000 0000 1000

0b 0001 0000 0001 0000

0b 0001 0000 0001 1000

What do you notice about the sets?

So what happens?

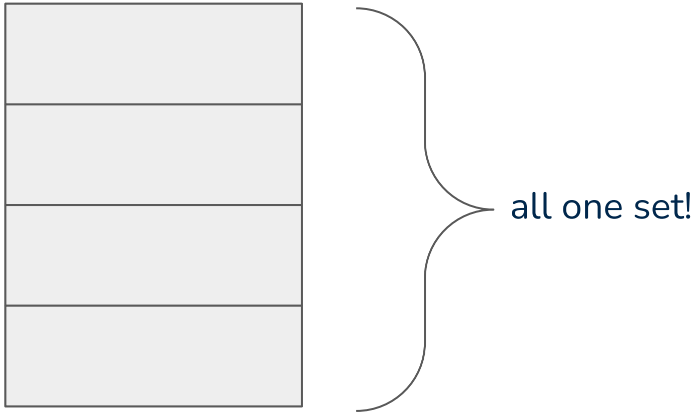
```c
// ints are 4 bytes here. Assume arr starts at 0x1000.
int arr[8] = {100, 0, 100, 0, 100, 0, 100, 0};
for (int outer = 0; outer < 3; ++outer) {
    for (int inner = 0; inner < 8; inner += 2) {
        arr[inner] += 1;
    }
}
```

always misses
never used
always misses
never used

Only set 0 and set 2 are used, and they miss every time.

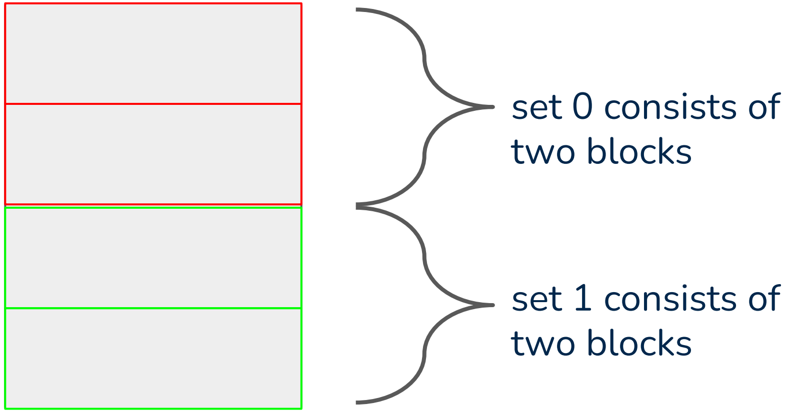
If you have only one set, this gives us our original design

Formally, this is called a **fully-associative cache** (all blocks **fully associate** with one set)



Or you can go in-between

- This is a **<u>set-associative cache</u>** (blocks associate with sets)
    - I *think* this is how most modern caches are designed
- Tradeoff:
    - The more sets, the less overhead and therefore the less cost to make the component
    - The less sets, the less cache misses and therefore the better the performance
        - Why would you miss less with less sets?



For the sake of time, I’m skipping a number of more advanced details

- What happens on memory write?
    - Do you modify the data in the cache or the data in RAM?
- Additional overhead
-

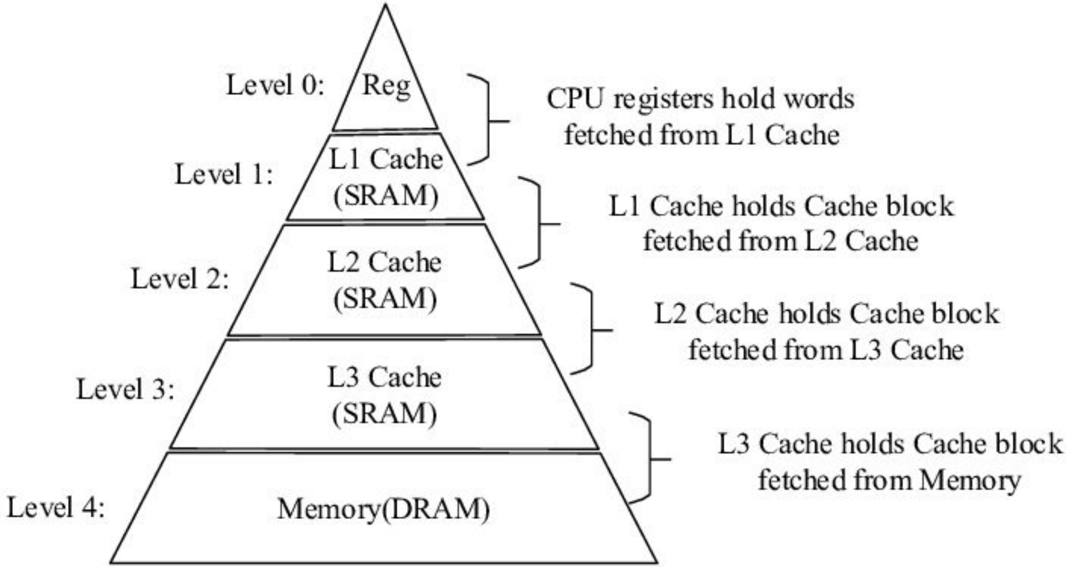
Caches in the real world

It is very rare that caching causes worse performance

- It is possible with poorly designed caches and adversarial access patterns
- However, it is possible – we had examples where we missed the cache every time

Modern processors usually have several levels of caches

Some levels are faster and smaller



UNIVERSITY OF MICHIGAN

If you are curious (and want to practice your English), I recommend checking out research papers on the subject

- Lots of interesting questions you can look into!
  - Overall theme: how do we save the most amount of time and money with cache designs?
- Conferences of interest:
  - ACM (Association for Computing Machinery)
  - IEEE (Institute of Electrical and Electronics Engineers) and subsidiary conferences
  - PACT (International Conference on Parallel Architectures and Compilation Techniques)
- Check to make sure that papers you are reading have high citations and are coming from top-tier conferences – don’t read garbage

Cache simulator and assignment

Go to GitHub ⇒ michigan-musicer ⇒ al.architecture ⇒ caching

Visit https://eecs370.github.io/simulators/cache/ and let's simulate some caches!

Use Chrome browser to avoid functionality issues

Simulate the programs provided and answer the questions. I will walk around and provide assistance as necessary