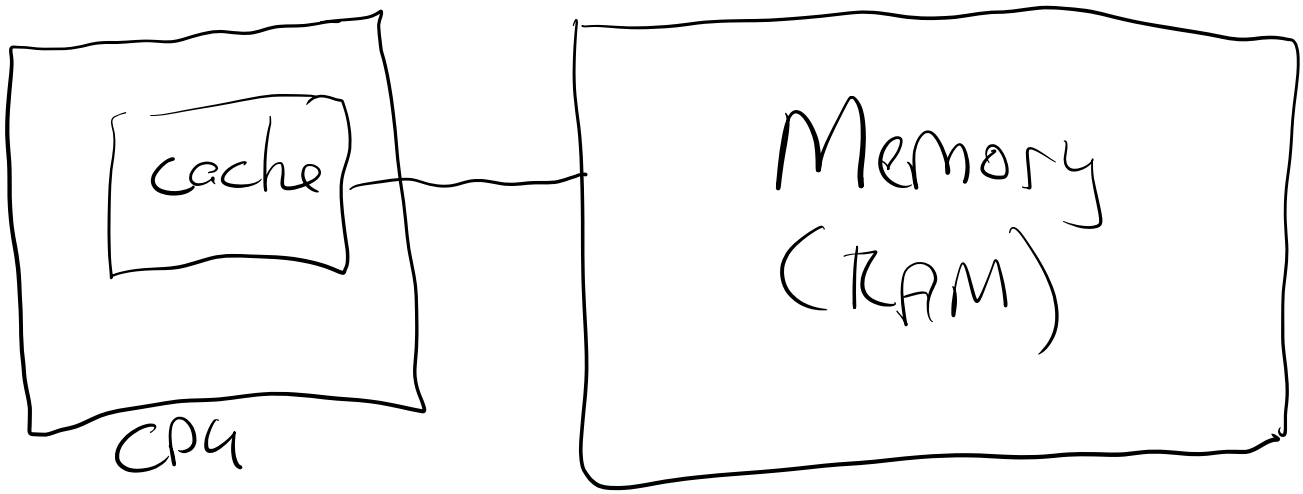


Cache stores copies of some of the data in memory.

- small fraction of the size of memory ($1/2^{20}$)

So, why do we think that keeping a tiny fraction of the data in cache helps speed things up?

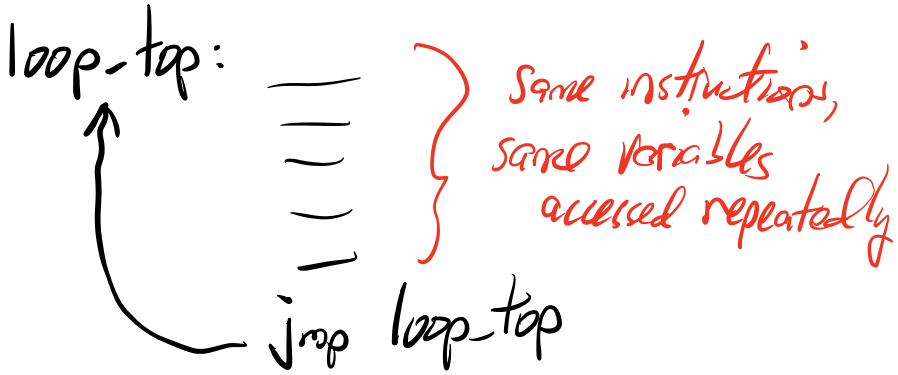
Because programs don't access memory randomly.

Programs exhibit:

① Temporal Locality: A location in memory that was accessed recently will be accessed again soon.

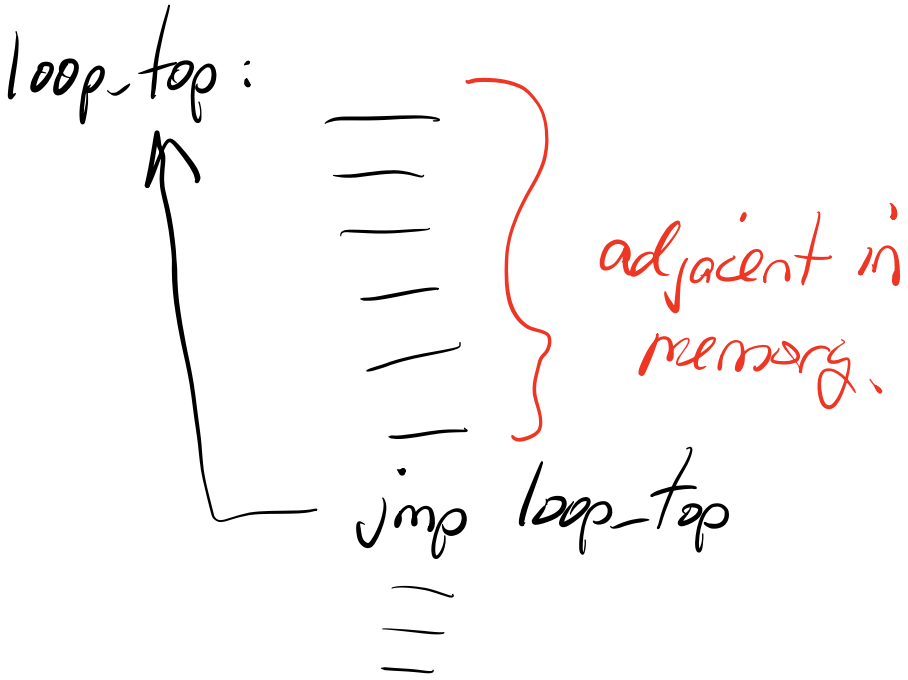
- That is, within a short period of time, the same memory locations will be accessed over and over again.
  - for data and instructions
- Why?

  Because of loops!

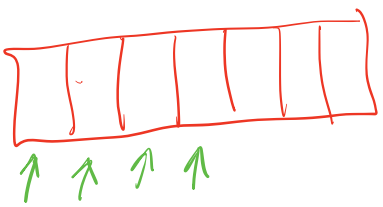


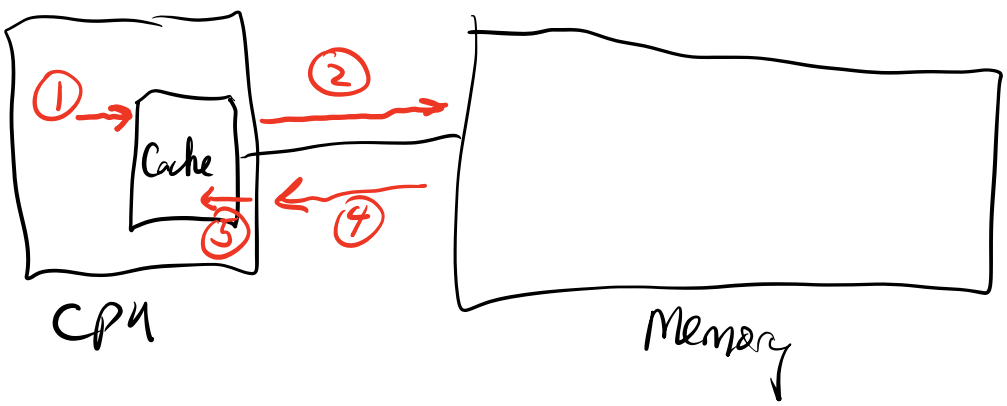
② Spatial locality: If a memory location has been accessed, then a neighboring location will be accessed soon.

- instructions and data
- Consecutive instructions are in consecutive memory locations (words)



Data: Adjacent elements of an array are in adjacent memory locations.





The CPU issues a request for data or an instruction at some address in memory.

1. The cache is checked to see if the data or instruction has already been copied into the cache.
   - if so, "cache hit", the data or instruction is provided immediately to the CPU.

2. Otherwise, "cache miss", a request for the instruction or data is issued to the memory.

3. CPU waits for the memory to send the requested data or instruction.
   "stall"

4. Memory sends back the requested data or instruction.

5. The data or instruction is written into the cache and provided to the CPU.

Cache Placement:

Where do we put an instruction or data in the cache, when it is sent from memory?

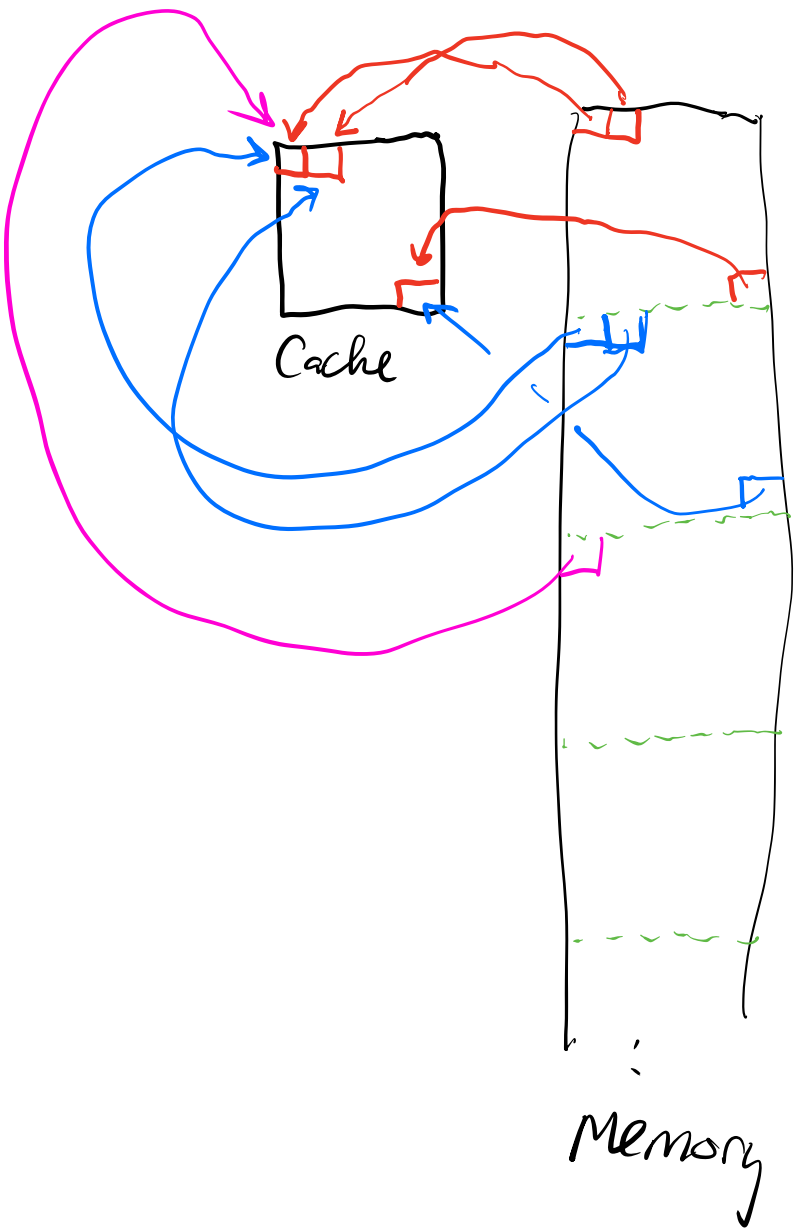
- retrieval from cache has to be fast.

Direct-Mapped Cache:

Where data is put depends only on the address of that data

- there is only one place in the cache corresponding to each memory address.

- Important: the memory always sends a whole word, even if only a byte is requested.



Assuming N words in the cache, the first N words in RAM are mapped to the N words in the cache.

The next N words in RAM are mapped in the same way to the N words of the cache, and so on.

So,

Cache index = address Mod N

- Cache index: which word in cache
- address: which word in RAM
- N: size in words of the cache.

Modulo is very fast if N is a power of two!

- Example: 3 bits

$$10101101_{11} \bmod 2^3$$

$$= 011$$

Taking mod 2^x just gives you the lowest x bits.

size of cache in words

So, if $N = 2^x$, then address mod N is just the x lowest bits, where x is $\log N$.

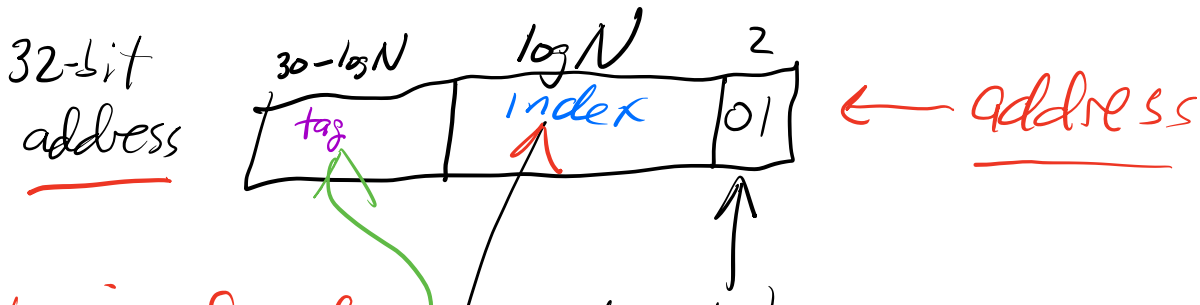
An address is given in bytes, but the memory sends only whole words

32 bits

Important:

assume words are 4-bytes

- the memory doesn't care about individual bytes within a word.



N = size of cache in words

lowest two bits are ignored by memory.

The logN bits are the result of the MOD operation above.

- gives us the cache index.

These bits will be different for different address that map to the same index in the cache.

- the "tag" bits

When data from some address in RAM is written into the cache, the tag bits are also written into the cache.

- Each cache entry needs to store the 30 - log N extra tag bits.

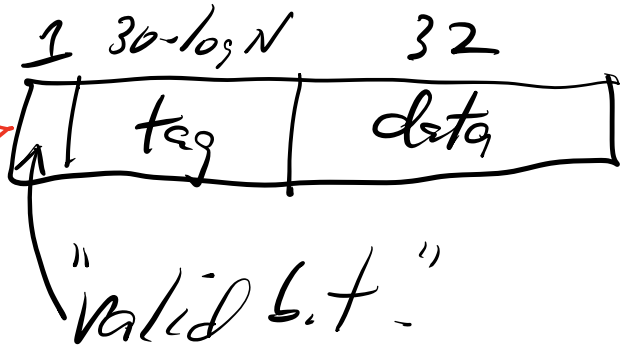
When the CPU issues an address, the index bits are used to determine the cache index and the tag bits of the address are compared to the tag bits stored at that index in the cache.

- If the tags match, it's a cache hit!

Also need an extra bit to indicate whether a cache entry holds valid data or just garbage.

- don't want your tag bits in the address to match random bits in the cache.

cache entry (in the cache)



- so, a cache hit occurs only if the valid bit is 1 and the tag in the address matches the tag in the cache entry.