Note that each word in cache is shared among
\[ \frac{2^{32}}{4 : N} = 2^{30} / N \text{ addresses} \]
Of course, not all \( 2^{30} \) addresses will be in use!

How do we know which address an entry in the cache currently corresponds to?
- Need a tag to identify the address
- Just use the upper \( 32 - 2 \cdot \log(N) \) bits as the tag
  - The rest of the bits are implicit in the cache index

---

\[ \text{CACHE} \]

- How do we know that a cache entry contains anything at all?
  - Need a "valid" bit

So, a cache entry consumes \( (32 - \log(N) + 1) \) extra bits as compared to memory.
Logic circuit for cache!

What happens in the case of a cache miss?

**Data Read:**
- Stall pipeline
- Initiate memory fetch
- Write into cache memory - include tag!
  - Continue execution

**Data Write:**
- Just write to cache - overwriting cache entry
  - Assumes "write-through" cache
  - Data must be sent to memory too
    - Need "write buffer"
  - Alternative: "write back" cache
    - Only write to memory when cache entry is removed.
Instruction Read:

- need to send original PC (PC-4) to memory on read request
- wait for data
- write into instruction cache
- restart execution

One word per cache entry exploits temporal locality but not spatial locality.

- to exploit spatial locality, need to bring in multiple words at a time
- each cache entry would then contain multiple words (e.g., 4, 8, 16 words) - "cache line"
- new addressing is

So, use bits: $2^{-(\log(M)+1)}$ to index into cache block

$\log(M)+2 - (\log(N)+1)$ to index into cache to find cache block.
Example:

512KB cache w/ 8 word cache blocks
(word offset: 3 bits
cache block index: 14 bits
tag: 13 bits
data: 8 words = 32 bits = 256 bits)

Cache logic for multword cache block:

In a mult-word cache block case, write misses require loading the cache block from memory into the cache.
- only writing one word, but need to have entire block in the cache.
- otherwise, other words in the block will be wrong!
Finding the write cache block size:
- too small: doesn't take advantage of spatial locality
- too large: too few blocks in the cache, space within blocks is wasted

Typical block size: 4-32 words (section 4 is 16 words)

Also, the larger the cache block, the more data that must be transferred upon a miss.
- wider bus, more complicated logic (multiplex)

For 4-word block:

Possibilities:

1. Bus 1-word connection:
   access time = cycles + cycles + cycles

Multi-word bus connection:
  access time = cycles + cycles

Single memory:
  access time = cycles + cycles

Single-word bus, interleaved memories
  access time = cycles + cycles + 4 cycles

- almost as fast, but simpler logic
Cache Performance

- Perfect Cache vs Imperfect Cache

Example: Assume CPI of 100% cache hit rate
How much slower is the processor with a 2% I-cache and a 4% data cache miss rate?

- Assume cache miss is 100 cycles & stall
- Instruction distribution includes 35% memory instructions

If a program executes I instructions:

perfect cache: \( I \times \text{CPI}_{\text{perfect}} = I \) cycles

imperfect cache = \( I \times \text{CPI}_{\text{perfect}} + (0.02 \times 100) + (0.04 \times 35 \times 100) \)

= \( I + 2I + 14I \)

= 4.4I cycles

So, program runs 4.4 times as slow as above cache performance.

- Doubling speed of processor at cache but not main memory

  - compare all same cache hit rates as a base

  old (above): \( \text{CPI}_{\text{slow}} = 4.4 \) cycles

  new: miss penalty is 200 cycles, so

  \( \text{CPI}_{\text{fast}} = I \times \text{CPI}_{\text{profile}} + (I \times 0.02 \times 200) + (I \times 0.04 \times 35 \times 200) \)

  = \( I + 4I + 2.8I = 7.8I \) cycles fast

  Since cycles = \( 2 \times \) cycles slow

  \( \frac{\text{Time slow}}{\text{Time fast}} = \frac{4.4I \times 2}{7.8} = \frac{8.8}{7.8} = 1.128 \)
Cache Placement

Associative caches

- "n-way set associative" - each block of memory can be mapped into n different possible locations in cache.
- 1-way - direct mapped
- n-way, where n = # blocks in cache - fully associative

Associative caches reduce the number of "conflict misses" - where a block in use must be removed from cache because another block in use maps to same cache block location.
- even though there might be plenty of room in the cache.
- blocks are N words apart, where N is number of words in the cache.
- Example: summing two arrays that use N words apart.
- Other kinds of misses are "compulsory misses" - occur the first time a block is needed and "capacity misses" - where the number of blocks in use exceed the cache size.
- Associativity doesn't help these last two.
For n-way associative caches, the larger the n,
the 1) more tag bits needed in the cache
2) the increased number of comparators
   (comparing tags)
3) the larger the max.

\[
\begin{array}{c}
\text{tag} \\
\text{set select} \\
\text{word offset} \\
\text{byte offset} \\
\text{so } \frac{N}{n} \text{ sets/cache}
\end{array}
\]

\[
30 - (\log(N) - \log(n)) - \log(M) = (30 - \log(N) - \log(M)) + \log(n)
\]
This, extra \( \log(n) \) bits per block.
- In fully associative case, that is \( n\log(N) \) bits.

You also need \( n \) comparators and an n-to-1
mux.
Example: 4-way set associative cache, 1-word cache block, 256 sets/cache (1Kw cache)
Cache Replacement Policy

- Where should an incoming block be placed in cache?
- Might have to remove block already there.

Direct mapped - no choice.
Fully associative - anywhere in cache, preferably an empty location.
Set associative - anywhere in the set.

Assuming no empty locations, practical possibilities are:
  random
  LRU - remove least recently used candidate block.

In cache's, decision must be made fast.
  LRU is only practical for 2- and 4-way set associative cache.

2-way - we need an extra bit per set, indicating which block was accessed most recently.
  On every access, set bit appropriately.

4-way: divide set in half.
  Use 1 bit to indicate recently used block in each half.
  Use other bit to indicate recently used half.