Designing a Cache

What is the basic operation we want our cache to do? (What is the input to the cache, and what is the output of the cache?)

Given an address, return the byte of memory at that address if it is in the cache.

What is the simplest way we could achieve this?

Have a list of all of the addresses in our cache, with the corresponding byte at that address.

Assuming we have a 32-bit system, how memory efficient is this caching system? (For every bit of physical memory, how many bits are we caching?)

For every 32 bits (for the address) of overhead, we are storing 8 bits in our cache. So out of every 40 bits, 8 bits to hold actual memory contents.

Could we make our cache system more memory efficient? How? (Can we exploit spatial locality?) How memory efficient is the new caching system you came up with? (Do you need to use all 32 bits to specify an address?)

Instead of storing 1 byte at a time, let's store multiple bytes at a time! If we stored 2^6 = 64 bytes at a time, and always have our block of bytes start at an address that is a multiple of 64, then there are 2^32/2^6 different starting points for our block, and so we only need log() = 32-6 = 26 bits to specify our starting address. So for every 26 bits of overhead, we are storing 64*8=512 bits in our cache. So out of 528 bits, 512 bits are used to hold actual memory contents. What an improvement!
Note that you may have gotten a different numbers if you used a different block size!

Any address can be anywhere in our cache. As a result, looking through all of our starting addresses to see if there is any match can take a long time. Can we make this search time shorter, while keeping the same number of starting addresses? (Hint: What if we restricted the possible locations of each address?)

Given each address, we can assign it a smaller section of our cache through a hash. This is the notion of a set!

We'd like to keep our mechanism for the above question simple (why?), and what's simpler than using bits from the address? Which bits from the address would best fit our needs? (Hint: We want high variability!)

Simplicity is faster, and we want our cache to be fast! High order bits tend to have lower variability (think about spatial locality) while lower order bits tend to have higher variability. However, we already have our lowest order bits used to address within our block, so we'll use the next best thing: the lowest possible order bits excluding those already used for blocks.

How many bits do we need to tag each of our starting addresses with now?

If our block size allows us to not specify the lowest b bits, and our set allows us to not specify the next lowest s bits, then we only need 32-b-s bits for the tag.

Oops, we've forgotten one small detail throughout this problem: How do we know if what's in the cache contains actual data? How can we fix this?

We can set a bit to indicate whether the pairing between address and data is valid.

Congratulations, you've designed your very own cache! Now let's see what happens if we change some of the parameters.
Let C be the cache size, B the block size, S the set size, and E the associativity (the number of potential starting addresses per set).

How can we relate C, B, S, and E?

C = B*E*S

What happens if E=1? What happens if S=1? What is the value of E when S=1? What is the range of E?

If E=1, we get a direct-mapped cache. If S=1, we get a fully associative cache. When S=1, E=C/B. If E > C/B, then S < 1, which doesn't make sense. E < 1 also doesn't make sense. So we must have 1 <= E <= C/B.