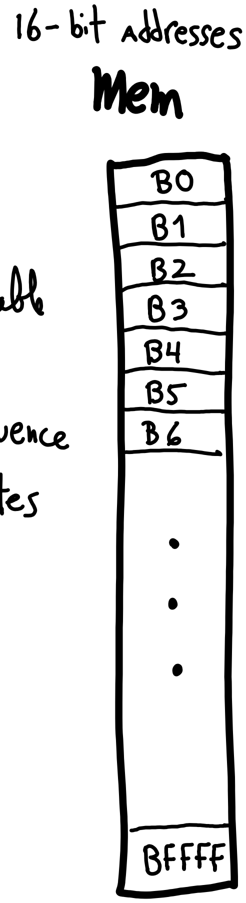
Memory, addresses, offsets



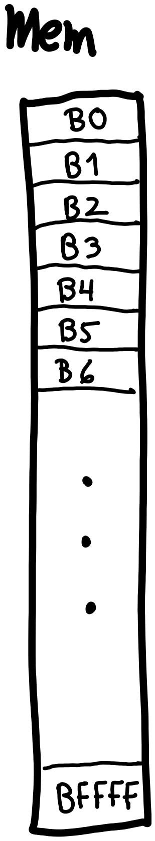
Byte-addressable = a sequence of bytes

Addresses refer to some number of bytes.

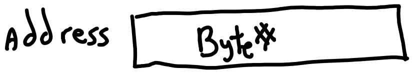
How many bytes is determined by the operation's data type.

Native data types
- data register size (32-bit, e.g.)
    - byte operation
    - half-word operation
    - word operation
- MAR size (40 bits, e.g.)
    - load word
    - load double word
    - load quad word
- virtual address (52 bits, e.g.)
    - page load

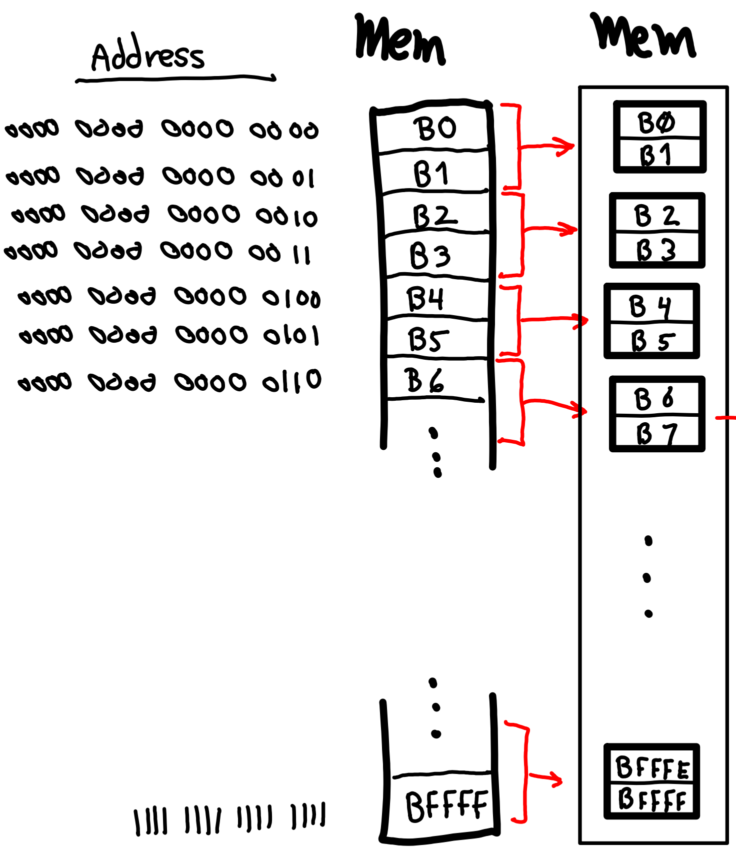
We can view memory as divided up
- aligned, non-overlapping chunks
    - aligned: first byte of first chunk is at x0000, e.g.
    - non-overlapped: memory is "tiled" by chunks
- chunk size depends on what we are interested in
- low address bits are offset into chunk
- high address bits are chunk number

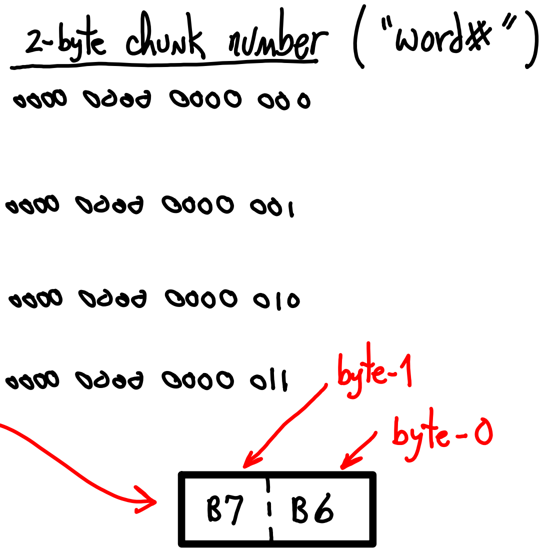


chunk = Byte



in byte addressable memory, all address bits are used to specify a Byte-sized chunk.



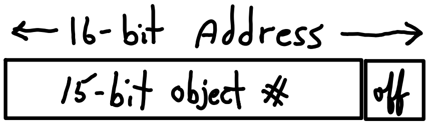


For 2-Byte words, we can think of the 16-bit addresses as split into two parts:

-- Address[15:1] is word number
-- Address[0] is byte offset into word

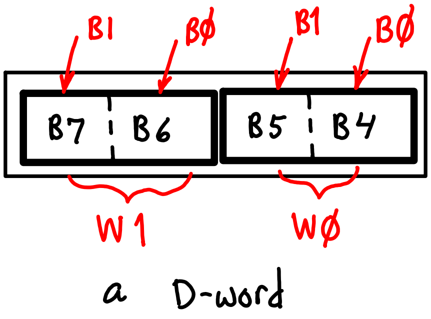
Each word has byte-0 and byte-1.

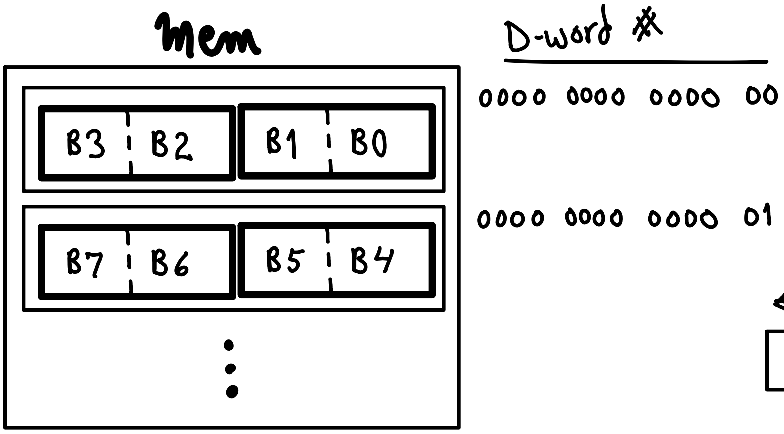
Objects are aligned, offset bits are zero at start of object.

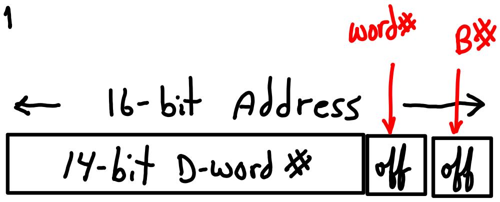


Compound objects

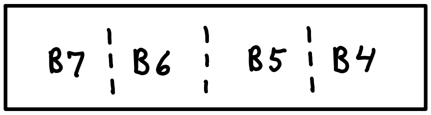
-- hierarchical inclusion
    -- higher-level composed of k lower-level objects
-- different offsets at each level

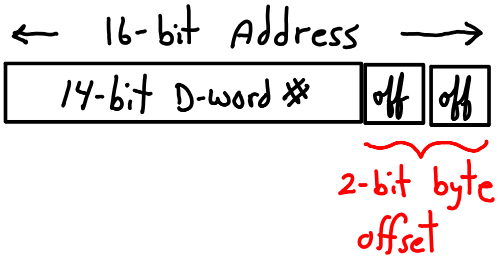




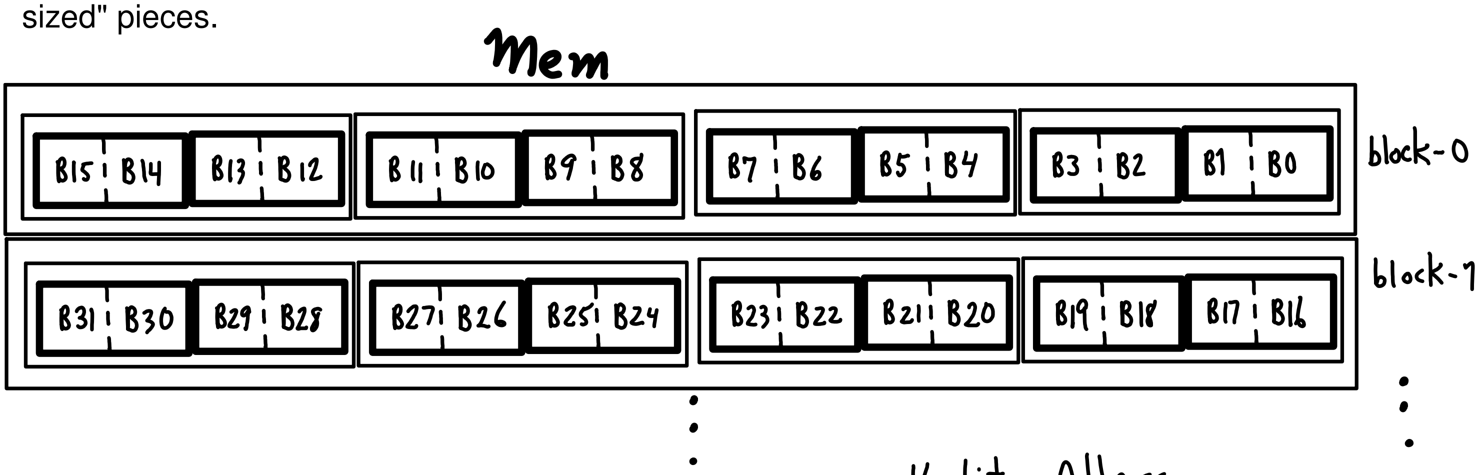


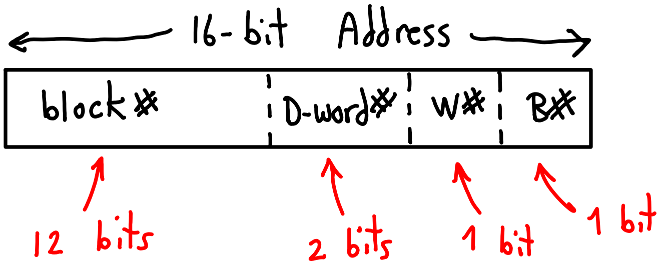
Of course, we could also see a D-word as composed of bytes.



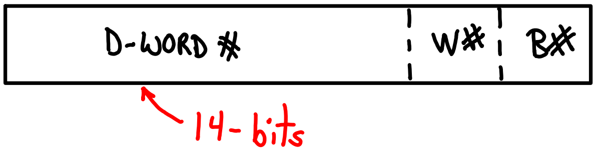


Suppose we have a cache. Say cache blocks are 16 B. We can say a cache block is 4 D-words, or 8 words, or 16 B. We can think of memory divided up into 16 B "cache block-sized" pieces.

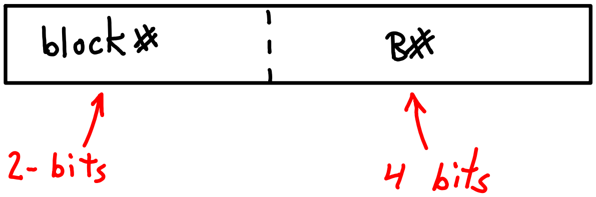


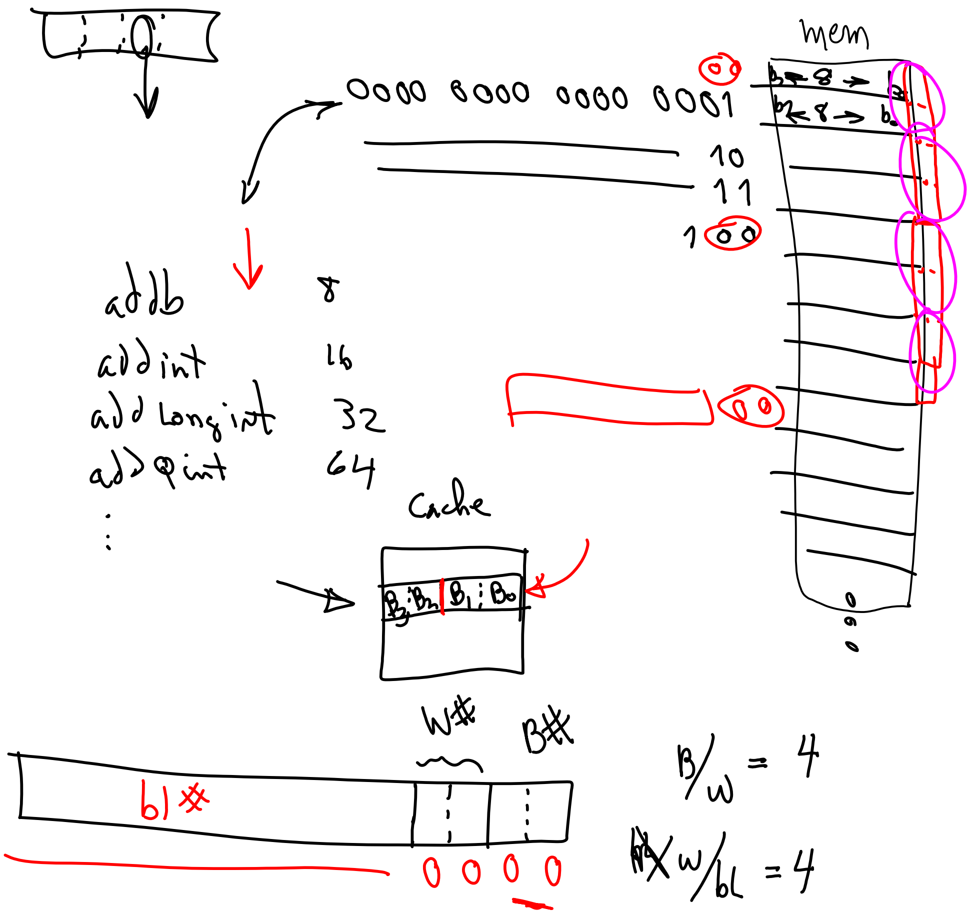


Of course, we can again flatten the hierarchy however we care to. Here the D-word# no longer refers to which D-word in a cache block, but which D-word of the entire memory.



Here, we consider the cache block to be composed only of bytes.

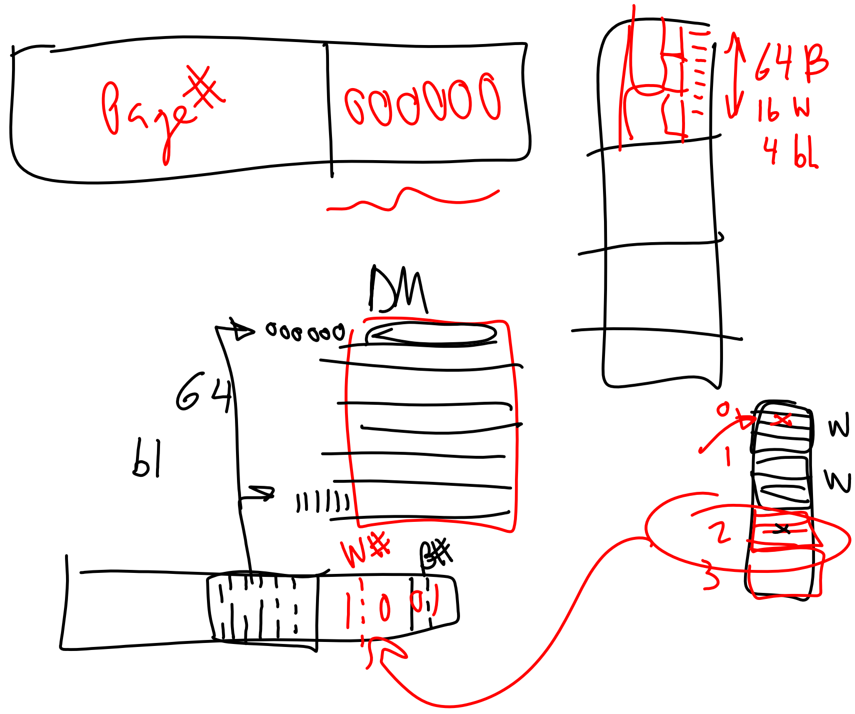




Different sized objects align by their low-order address bits. A 16-Byte object aligns at addresses w/ last 4 bits all 0; 8-Byte objects align w/ low 3 bits all 0.

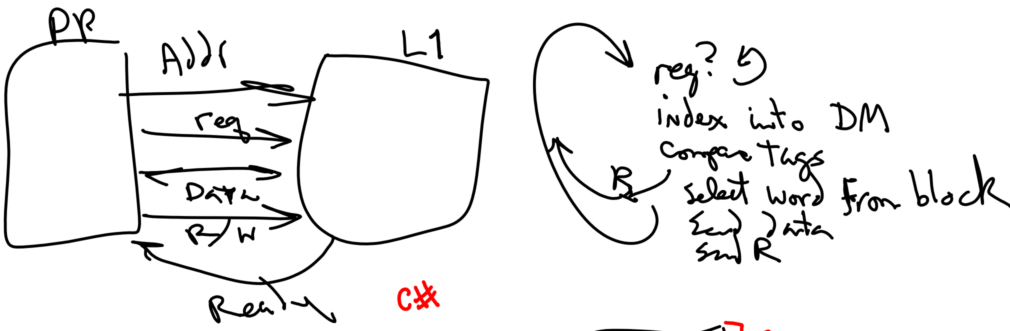
Which objects are relevant depends on context of discussion.

For aligned objects, we can think of the bit fields as indicating which object within a larger object. Here, 4B words within 4-word blocks.

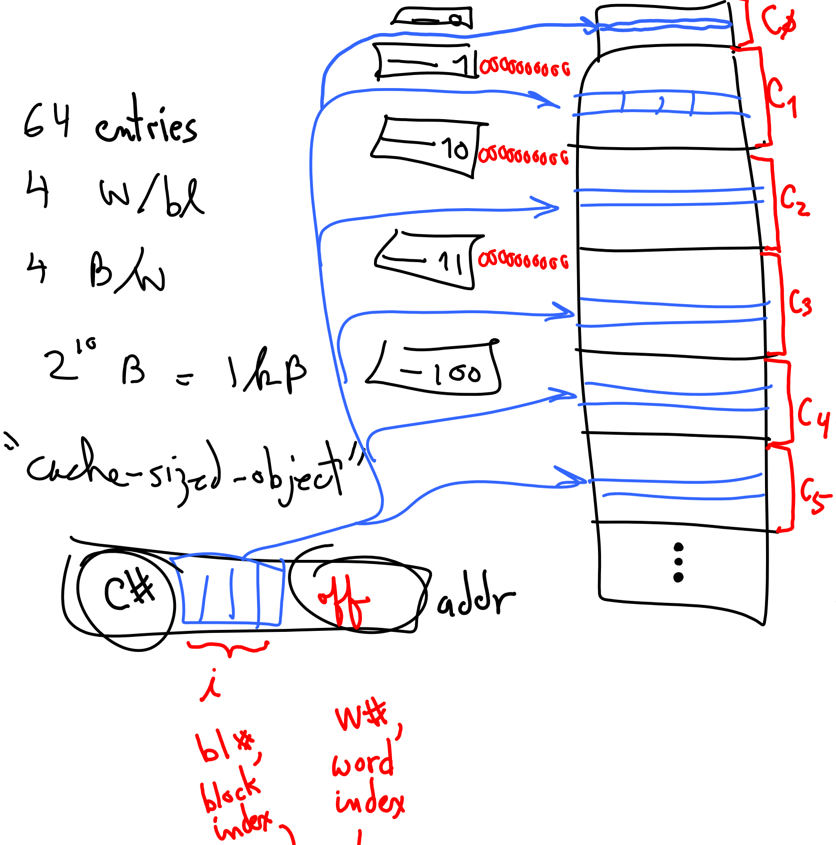


A large object, e.g., a page, can be thought of simply as containg some number of bytes. Here, pages align w/ 6 low bit equal 0 and page size is 64B. Of course, we can consider the page as having 4 4-word blocks or 16 4B words.

For a DM, some number of bits are index into the cache. Here there are 4 words per block. The number of entries determine how many bits are used for indexing. If the DM has lots of entries, the index bits can include some of the low-order page# bits.



Communication between a CPU and L1 looks just like the CPU-Memory communication when no cache is present. From the CPU side it looks like a memory interface.

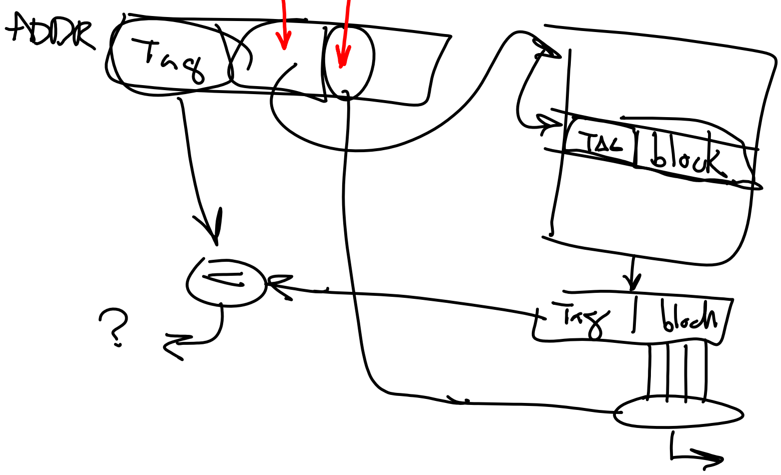


We can think of aligned, cache-sized objects. The upper bits would be thought of a the C#, or which cache-sized object in memory. The cache index is then which block within a cache-sized object. The low bits are the offset into the block.

Blocks with different C#'s but the same index collide. The C# is the tag. Here there are 64 entries and 6 index bits, 4 word/block and 4B/word; 1 kB per C#. Alignment is w/ low 10 bits 0.

If the discussion context was L2 instead, then the bits considered C# and index would shift according the L2's size and number of entries. Offset bit fields would be by block/word sizes.

Note that the degree of associativity has no effect on these numbers. n-way associatvie has n DMs: it allows collisions to be accomodated. The total size of the cache is independent; the DM size sets the assignment of bit fields.



Given an address, accessing the item involves the following (for read, write is a little different:

-- index into the DM (or mulitple DMs)
-- get the cache line (tag+block+otherBits)
-- compare the two tags
-- use W# to select which word in block
-- send word to CPU (or write data to word)