Memory, addresses, offsets

Mem

B0
B1
B2
B3
B4
B5
B6
⋮
BFFFF

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

Byte –
addressable
=
a sequence
of bytes

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
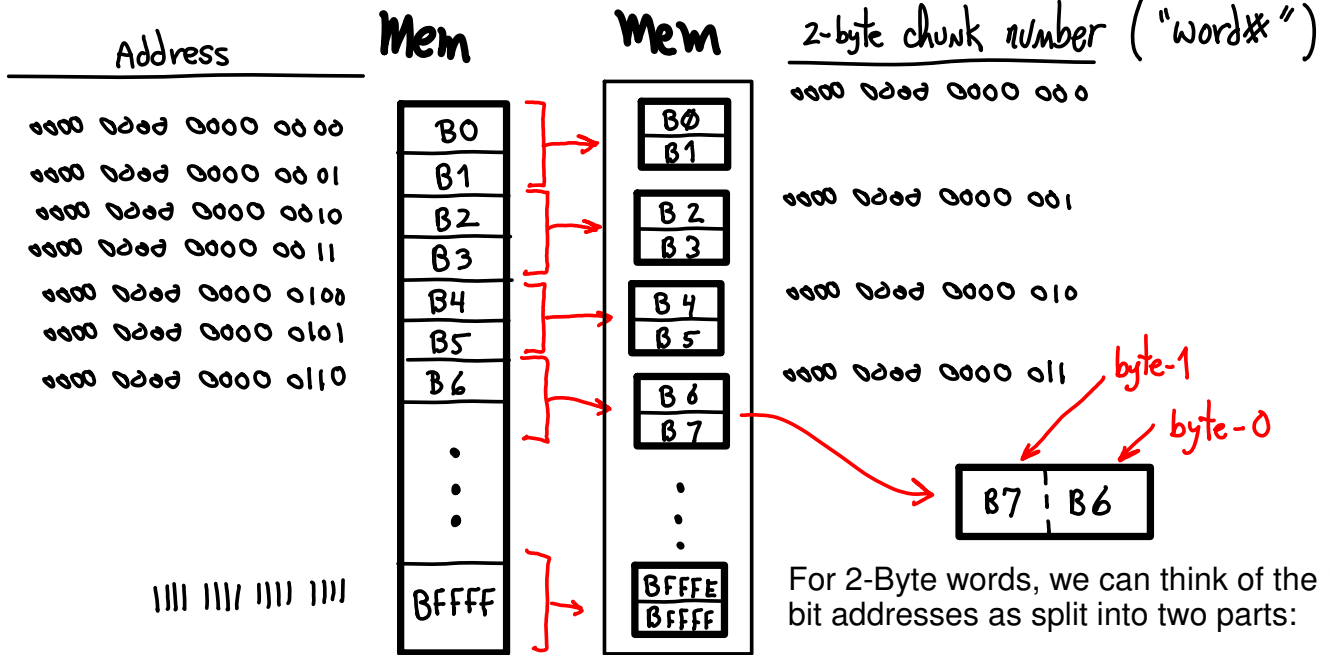   -- high address bits are chunk number

chunk = Byte

Address | Byte* |

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

Mem

B0
B1
B2
B3
B4
B5
B6
⋮
BFFFF

Address | Mem | Mem | 2-byte chunk number ("word#")

Address column (binary):
```
0000 0000 0000 0000
0000 0000 0000 0001
0000 0000 0000 0010
0000 0000 0000 0011
0000 0000 0000 0100
0000 0000 0000 0101
0000 0000 0000 0110
```

`1111 1111 1111 1111`

Mem (first): B0, B1, B2, B3, B4, B5, B6, ... , BFFFF

Mem (second): B0 / B1, B2 / B3, B4 / B5, B6 / B7, ... , BFFFE / BFFFF

2-byte chunk number ("word#"):
```
0000 0000 0000 000
0000 0000 0000 001
0000 0000 0000 010
0000 0000 0000 011
```

byte-1, byte-0

| B7 | B6 |

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

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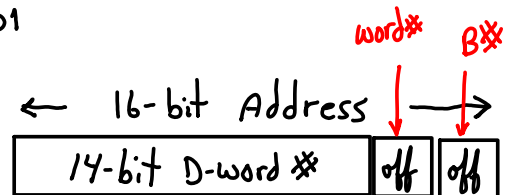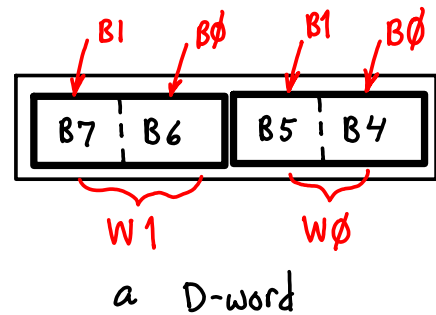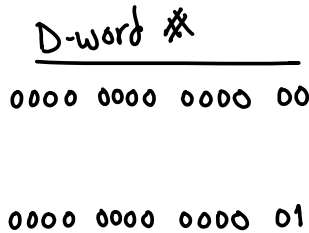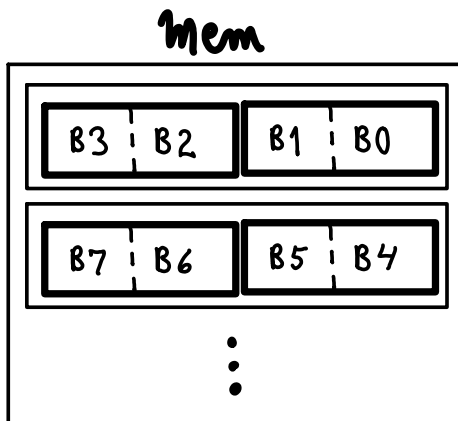-- Address[0] is byte offset into word

Each word has byte-0 and byte-1.

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

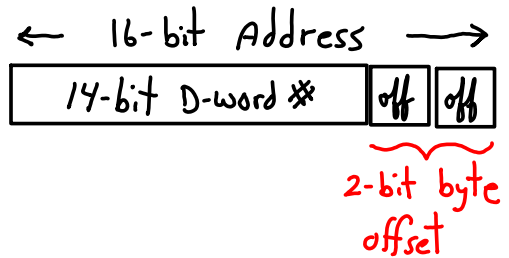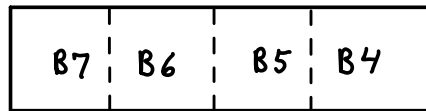← 16-bit Address →

| 15-bit object # | off |
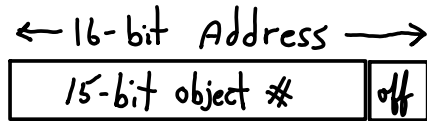
Compound objects

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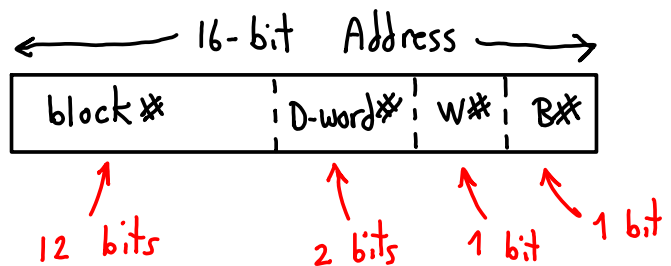-- different offsets at each level

B1   B0   B1   B0
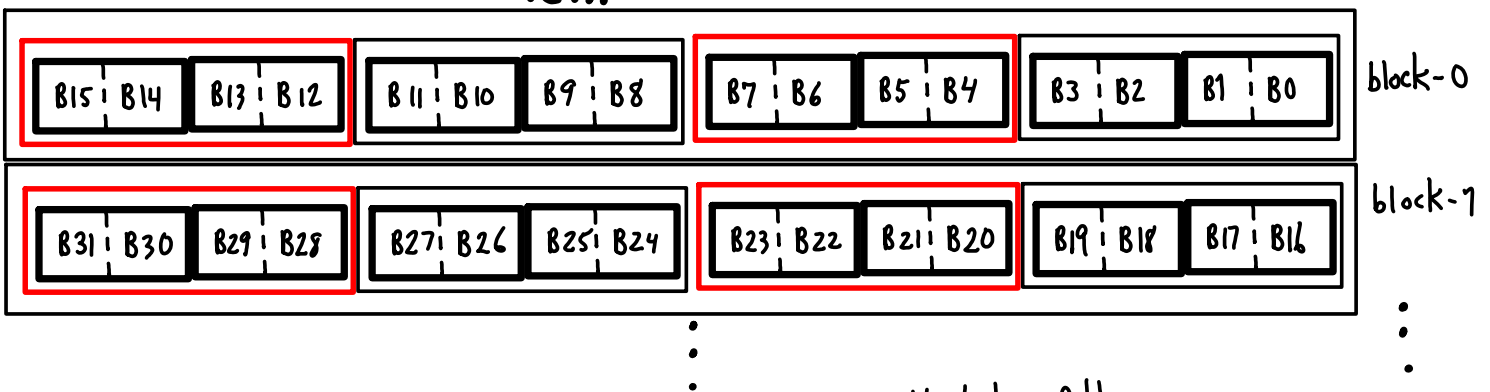
| B7 | B6 | | B5 | B4 |

W1   W0

a D-word

Mem

| B3 | B2 | | B1 | B0 |

| B7 | B6 | | B5 | B4 |

D-word #

`0000 0000 0000 00`

`0000 0000 0000 01`

word#   B#

← 16-bit Address →

| 14-bit D-word # | off | off |

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

B7	B6	B5	B4

← 16-bit Address →

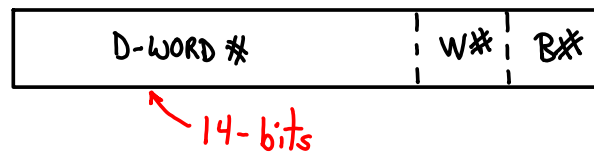| 14-bit D-word #	off	off

2-bit byte offset

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.
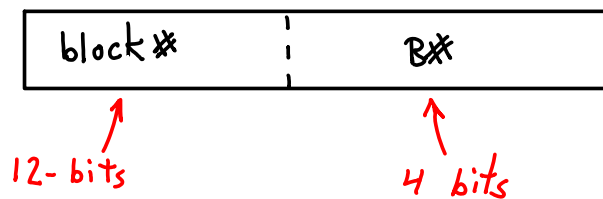
Mem

| B15 B14 | B13 B12 | B11 B10 | B9 B8 | B7 B6 | B5 B4 | B3 B2 | B1 B0 | block-0 |

| B31 B30 | B29 B28 | B27 B26 | B25 B24 | B23 B22 | B21 B20 | B19 B18 | B17 B16 | block-1 |

← 16-bit Address →

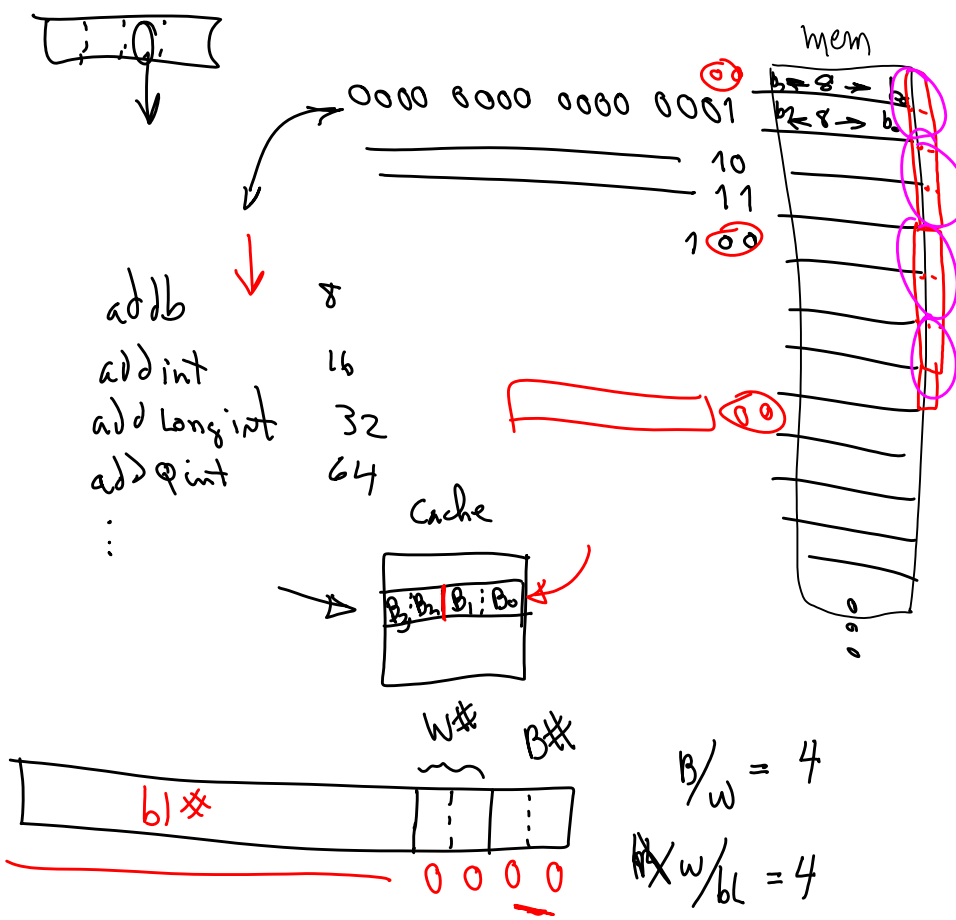| block #	D-word#	W#	B#

12 bits   2 bits   1 bit   1 bit

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.

D-WORD #	W#	B#

14-bits

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

block #	B#

12-bits   4 bits

And so on to pages, etc.

0

0000 0000 0000 0001

00

10

11

1 00

BYTES

B←8→

B←8→b

addb   8
add int   16
add long int   32
add Qint   64
⋮

Cache

$B_3 B_2 | B_1 : B_0$

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.

W#   B#

bl#

0 0 0 0

$B/w = 4$

$w/bl = 4$

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.

Page#   000000

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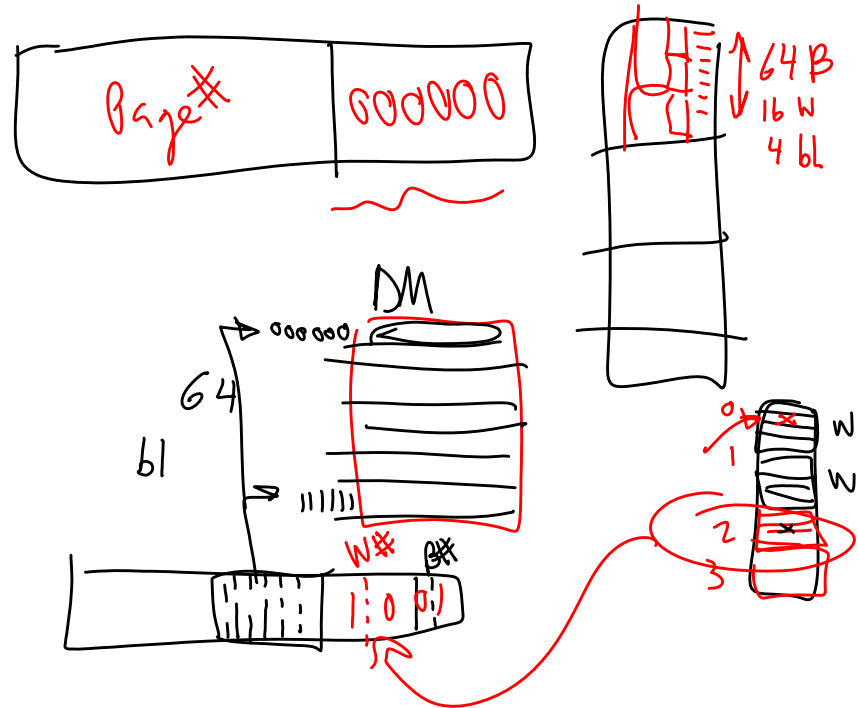
A large object, e.g., a page, can be thought of simply as containing some number of bytes. Here, pages align w/ 6 low bit equal 0 and page size is 64B. Of course, we can consider the page as having 4 4-word blocks or 16 4B words.

DM
000000

64
bl

W#   B#
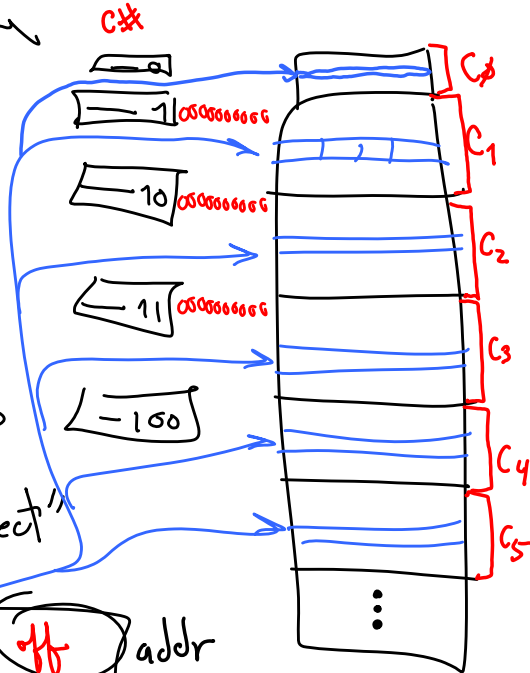1 : 0   bl

0
1
2
3

W
W

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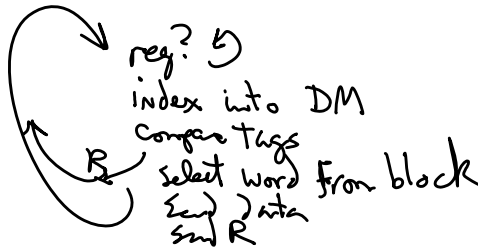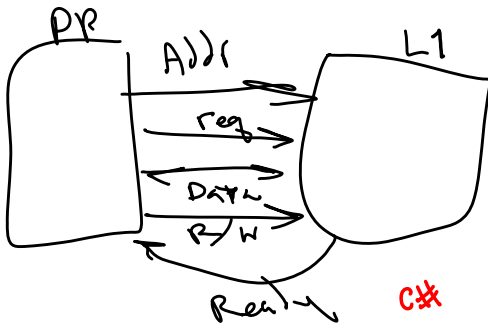
Communication between a CPU and L1 looks just like the CPU-Memory communication when no cache is present. From the CPU side it looks like a memory interface.

PP Addr L1
reg
DATA
R/W
Ready

reg? 5
index into DM
compare tags
select word from block
send data
send R

C#

64 entries
4 w/bl
4 B/w
2^{10} B = 1kB
"cache-sized-object"

C# | off | addr
i
bl#, block index
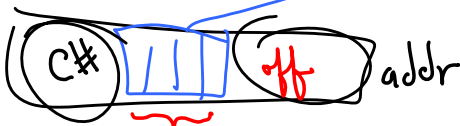W#, word index

C_0 C_1 C_2 C_3 C_4 C_5

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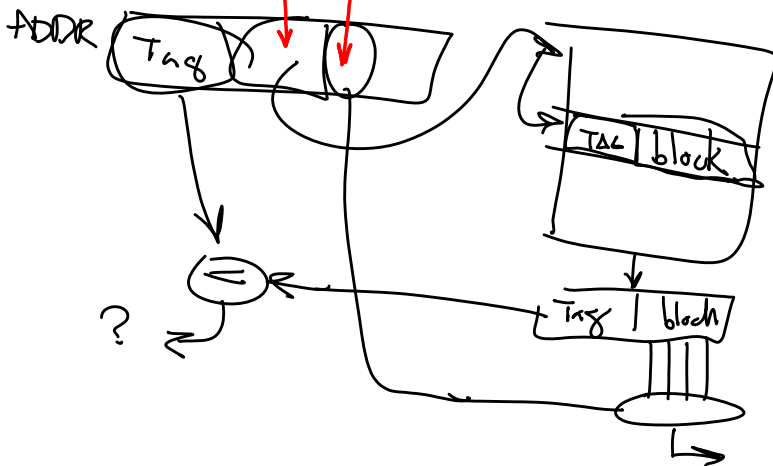
Blocks with different C#'s but the same index collide. The C# is the tag. Here there are 64 entries and 6 index bits, 4 word/block and 4B/word; 1 kB per C#. Alignment is w/ low 10 bits 0.

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.

ADDR Tag
TAG | block
Tag | block
?

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)