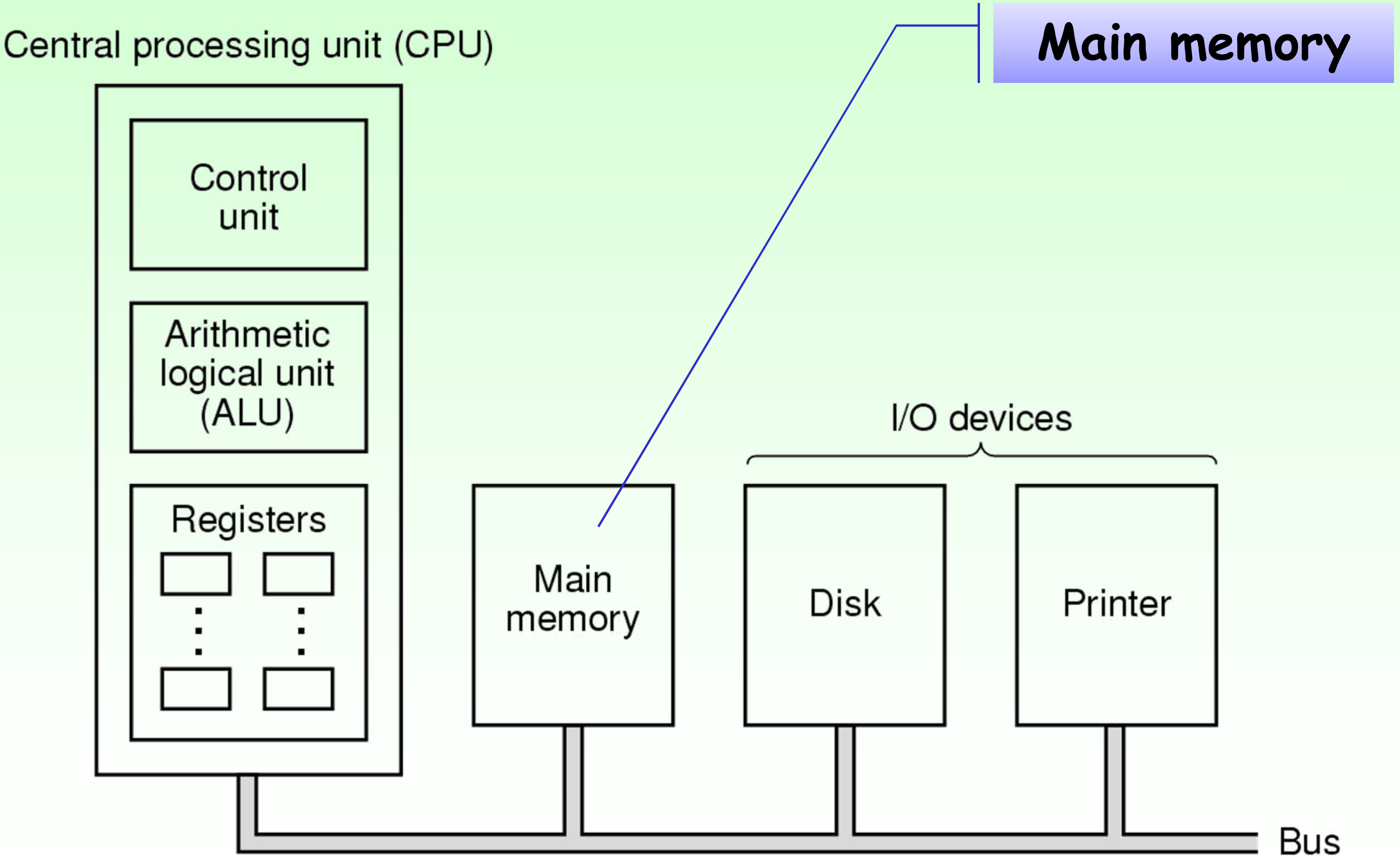
CSCU9V4
Systems

Systems Lecture 7

Computer Organisation 2

Main Memory & Cache

Structure of a Simple Computer

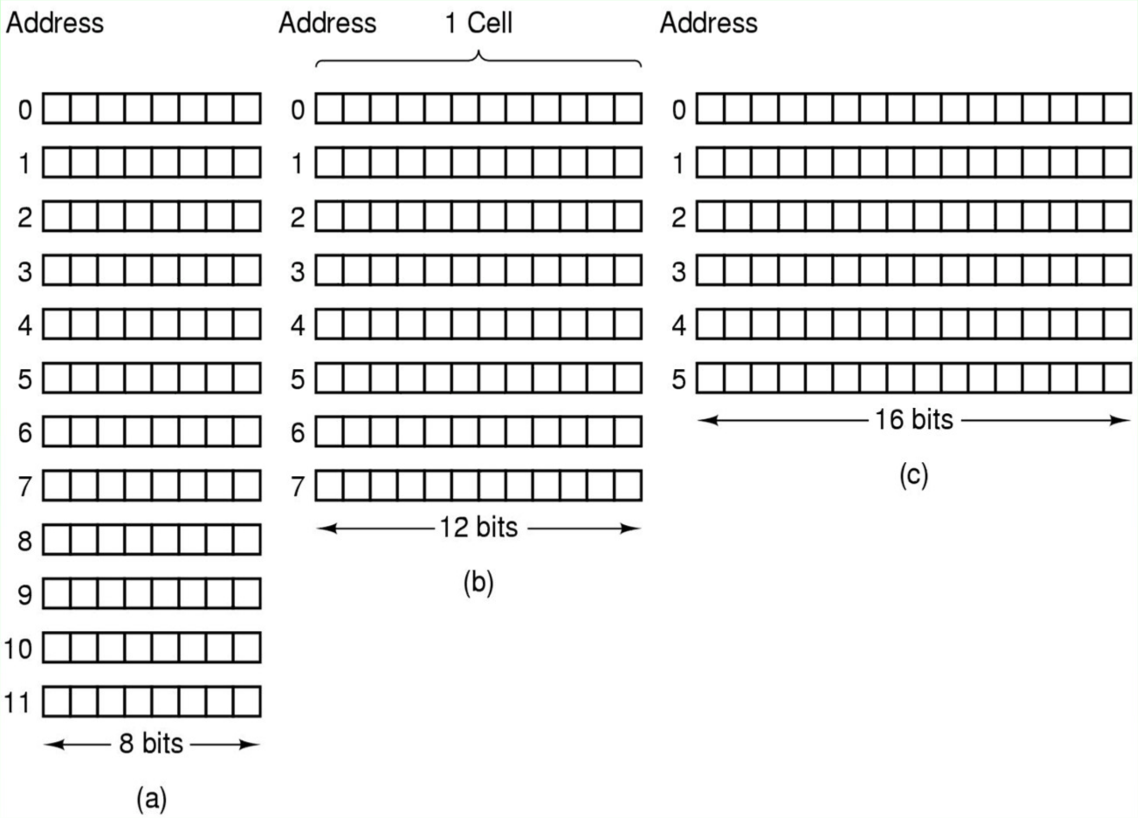


Main memory

Memory Cells

To the programmer, the main memory appears as a 1-dimensional array of *cells*.

- A *cell* is the smallest element of memory that can be accessed.
- Every cell has a unique address, starting (generally) at 0.
- Each cell is individually addressable.
- If a memory has n cells, their addresses will usually be 0 – (n-1)
- All cells in a computer are the same size (hold the same number of bits)
- Example shows 3 ways of organising cells
- Adjacent cells have consecutive addresses



Addressing Memory

- Computers use the binary system therefore express memory addresses as binary numbers
- An address with m bits, can be used to address 2^m memory cells.
- Previous example:
  - a) requires 4 bits ($2^4 = 16$);
  - b) requires 3 bits ($2^3 = 8$);
  - c) requires also 3 bits, because $2^2 = 4$ which is too small
- The number of bits in an address determines the number of directly addressable cells; it is independent on the number of bits in a cell
  - 2^{12} cells with 8 bits each and 2^{12} cells with 16 bit each, both need 12-bit addresses
- Cell is the smallest addressable unit!
  - Generally a byte: also explains why memory is sold by the byte (or Mbyte, Gbyte, ot Tbyte)
  - Also explains why memory capacity is a power of 2.

Example Word and Cell Sizes

Processor (microprocessor)	*Cell Size*	*Word Size*
6800, 6502, 6809, 6802, 8088	8	8
68000,68010,68020, 68030, 68040	8	32
8086, 80286	8	16
80386,80486, Pentium-Pentium 4	8	32
PowerPC G5, DEC Alpha, recent Intel processors (IA64, Core Ix, Nehalem, Sandy Bridge, etc)	8	64

Addressable memory:

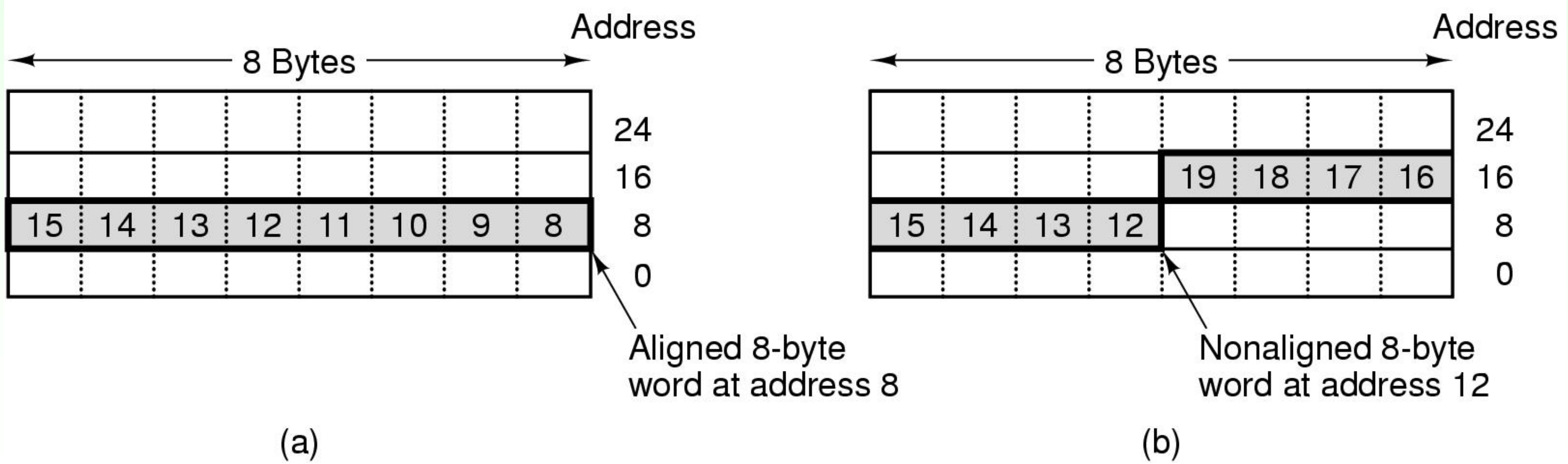
- Size of MAR defines number of addressable cells.
- 16 bits => 2^{16} cells = 64K cells (usually 64K bytes) (early microprocessors)
- 32 bits => 2^{32} cells = 4G cells (usually 4G bytes) (Pentium series)
- 42 bits => 2^{42} cells = 4T cells (usually 4T bytes) (Power PC G5, Intel Xeon E7 V2)

- Word size and address size now often the same eg 32 or 64-bit

Aligning Words

- Many architectures require words to be aligned to their natural boundaries (8-byte word may begin at address 0, 8, 16, …; but not at 4, 12, or 19)

- Memories operate more efficiently if they are aligned
- Pentium fetches 8 byte at a time, uses 36-bit physical address space, but only has 33 address bits
  - 3 least significant bits are not specified
  - No non-aligned memory reference possible!



Problems with Aligning Words

- Pentium II needed to be able to access memory at any location
  - Two memory accesses needed
  - Example: 4-byte word at address 7 (bytes 7 – 11)
  - One reference to address 0 (get bytes 0 – 7)
  - A further reference to address 8 (get bytes 8 – 12)
  - Hardware extracts the relevant bytes and puts a 4-byte word together

- Required extra logic on the chip -> slow and expensive!

Intel's advice

- **Intel® Pentium® 4**
  and Intel® XeonTM **Processor Optimization**
  Reference Manual
- For best performance, align data as follows:
- Align 8-bit data at any address.
- Align 16-bit data to be contained within an aligned four byte word.
- Align 32-bit data so that its base address is a multiple of four.
- Align 64-bit data so that its base address is a multiple of eight.
- Align 80-bit data so that its base address is a multiple of sixteen.
- Align 128-bit data so that its base address is a multiple of sixteen.

Memory Performance

- CPUs are much faster than memory
  - techniques such as pipelining, superscalar architectures make the CPU's memory demand very high indeed
  - A 2 GHz CPU (with a pipelined superscalar architecture) may need up to 8,000,000,000 instructions every second
  - A 2 or 3GHz processor is limited by operating over a 1GHz data bus

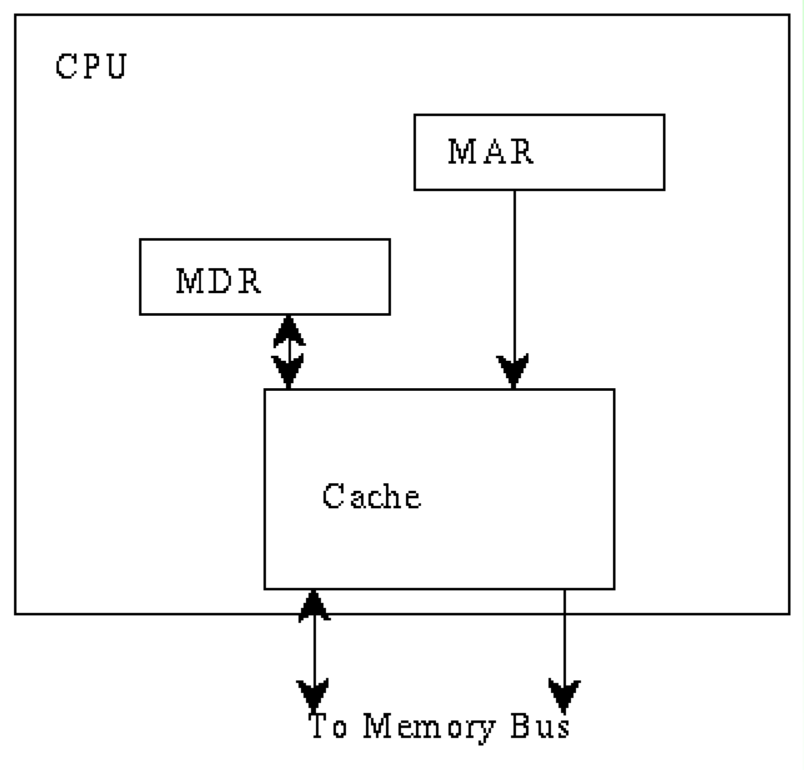
- Problem: Supplying sufficient data and instructions to the CPU
  - The "Von Neumann Bottleneck"

Improving Performance

- Faster memory?
- Use even faster buses (connection between main memory and CPU)?
- Place ultra-fast memory on the CPU chip
  - On-chip memory can be accessed much faster (why? nearer, + arguments from analogue signal processing, not appropriate for this course)
- Use cache memory
  - High speed memory on (or very close to) CPU chip
- Combine fast and slow memory
  - Aim to get the speed of the fast memory and the size of the slow memory
  - Keep the most heavily used words in the cache memory
  - If the CPU needs a word it first checks the cache
  - Which data should be stored in the cache?

Cache Memory

- Cache memory
  - Usually placed on the CPU chip
  - access is very fast
    - 10-100 times faster than main memory
  - memory bus is used for
    - cache misses
    - when data is written (need to maintain consistency).



Cache Misses

- A cache miss occurs when the CPU needs an item (instruction or data) which is not currently in the cache.
- When this occurs action must be taken:
  - item must be copied from main memory into the cache,
  - replacing something which is already in the cache
    - which may have to be saved in main memory before it is overwritten in the cache
- There are complications here:
  - which item in the cache will be selected to be overwritten?
- But it's worth it ...

Why does Cache Memory Help?

- Cache memory helps in practice because programs exhibit *locality of reference.*
- That is, recently used instructions and data are likely to be used again
  - true both for programs and data
  - particularly helpful for programs, because programs are read-only
    - no problems with writing and consistency.
- In particular, programs contain loops
  - most programs spend a lot of their time executing loops, going over the same instructions that are likely to be in the same area of memory
  - the larger the cache, the more likely the required data/instructions will be within the area copied to the cache

Example

```
for (index = 1; index<MAXXPOINTS; index++) {
     endpoint = floor(index *
          ((float)displayframelength/MAXXPOINTS));
     /* find mean of points */
     pval = 0;
     for (j = startpoint; j<endpoint; j++)
            pval += displayframe[j];
     pval = pval/(endpoint - startpoint);
     startpoint = endpoint;
     Xlocn += Xinc;
        PSlineto(Xlocn, pval * Ymag + Yaxis);
}
```

Say MAXXPOINTS is 250, and endpoint might be 50 to 500 each time: how much time is spent in the innermost loop?)

Example (Continued)

- This fragment of code (from a real program) is generally executed thousands of times. The code will all fit into a cache of any reasonable size.
- It will therefore only be fetched once - so relieving the CPU/Memory connection of a considerable amount of work.
- Many data items change each iteration - but not all of them. Here

```
displayframelength, Ymag, Yaxis, Xinc
```

are all the same in each iteration.

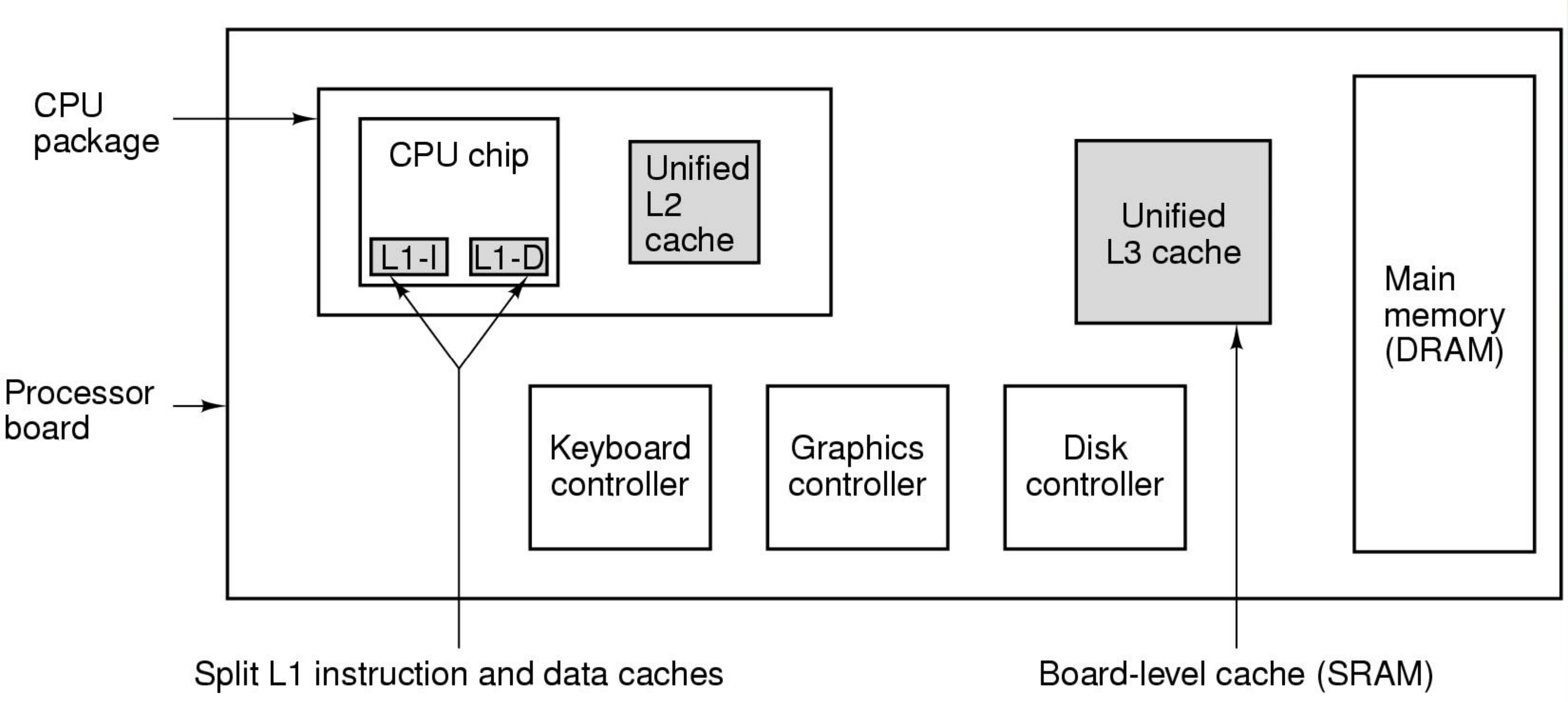
- The use of a cache memory can make a huge difference to the amount of data that needs to be transferred from the main memory to the CPU.
- The overhead of the cache is to be found in
  - greater circuit complexity,
  - dealing with cache misses.
- Cache hit rates are usually 90-99% for a reasonably large cache

Design of Caches

- Locality principle is used as a guide
- Main memories and caches are divided into cache *lines*
  - E.g. 64 byte line
- A full cache line is loaded into the cache
- More efficient than loading individual words
- Cache design is important:
  - Cache size
  - Size of cache lines
  - Cache organisation
  - Data and instructions kept in the same cache?
  - Number of caches?

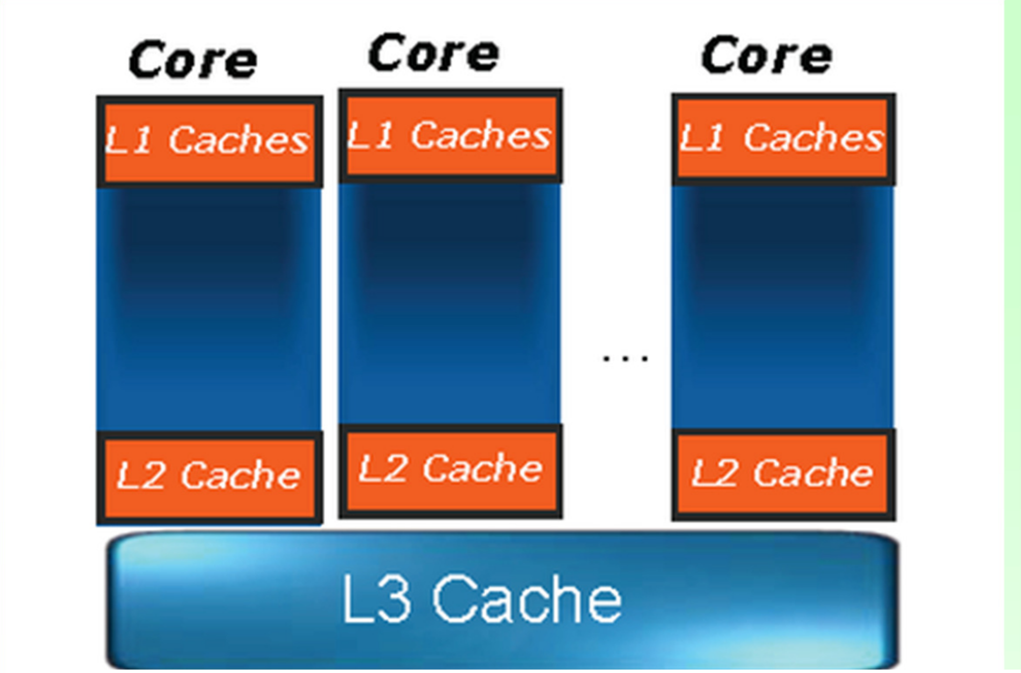
Number of Caches

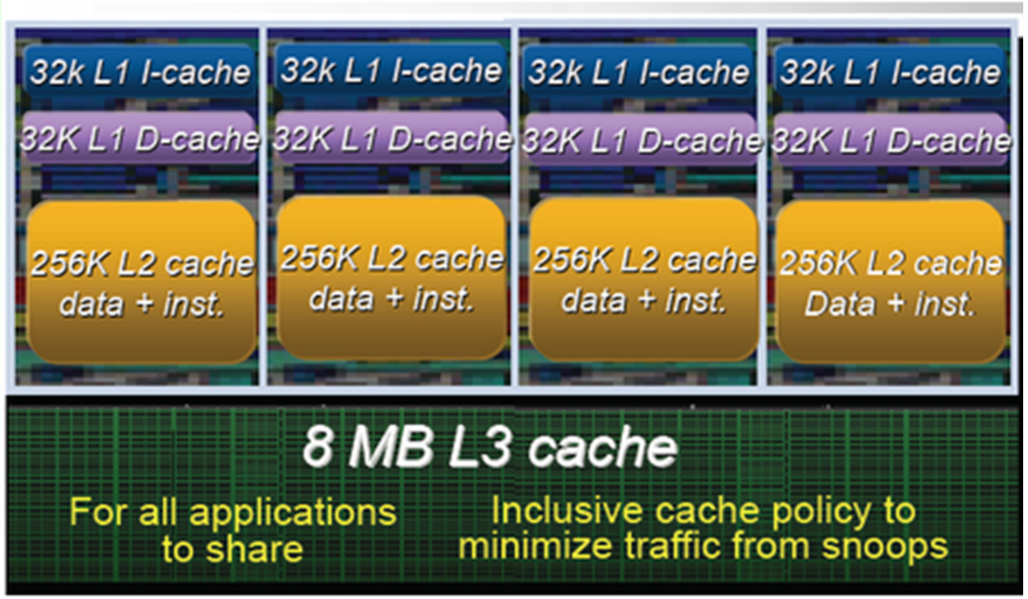
- Latency (time to load a memory item)
  - can be improved by a reasonable sized cache
- Bandwidth can be improved by having multiple caches



Multi-level caches

- Modern CPUs use multi-level caches, on-chip.
- Example is from Intel i7.
- L1 cache is in 2 parts, an instruction cache and a data cache: one per core
- L2 cache is unified: one per core
- L3 cache is unified across all (4) cores.




32k L1 I-cache
32K L1 D-cache
256K L2 cache data + inst.
32k L1 I-cache
32K L1 D-cache
256K L2 cache data + inst.
32k L1 I-cache
32K L1 D-cache
256K L2 cache data + inst.
32k L1 I-cache
32K L1 D-cache
256K L2 cache Data + inst.
8 MB L3 cache
For all applications to share
Inclusive cache policy to minimize traffic from snoops

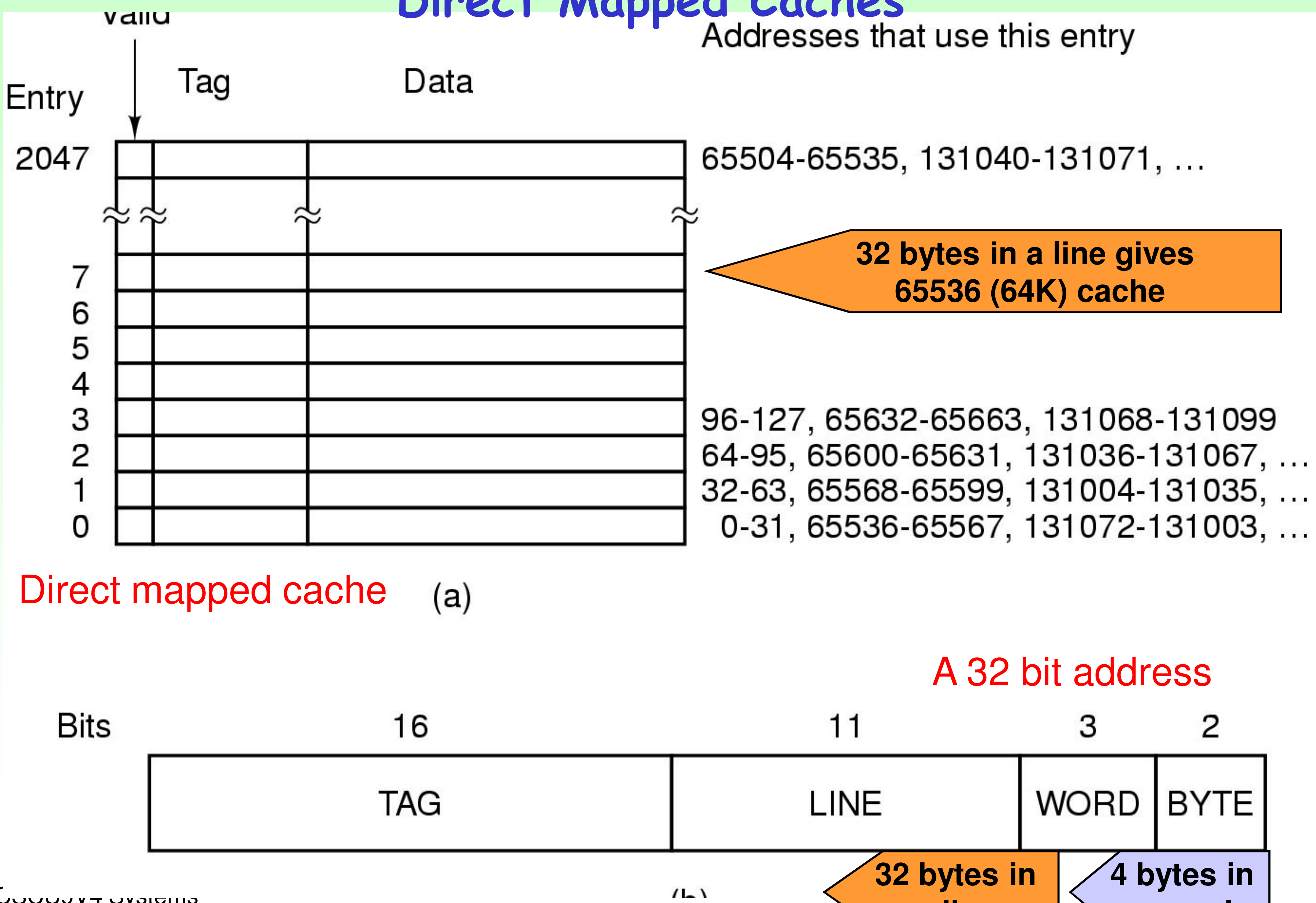
What to keep in the cache?

- Spatial locality
  - Words located close to the currently required one
    - Expected to be used soon

- Temporal locality
  - Words most recently used
    - Likely to be used again soon

- There are many forms of cache memory:
  Next few slides show some of these: included for completeness, not examinable!

Direct Mapped Caches



32 bytes in a line gives 65536 (64K) cache

Direct mapped cache

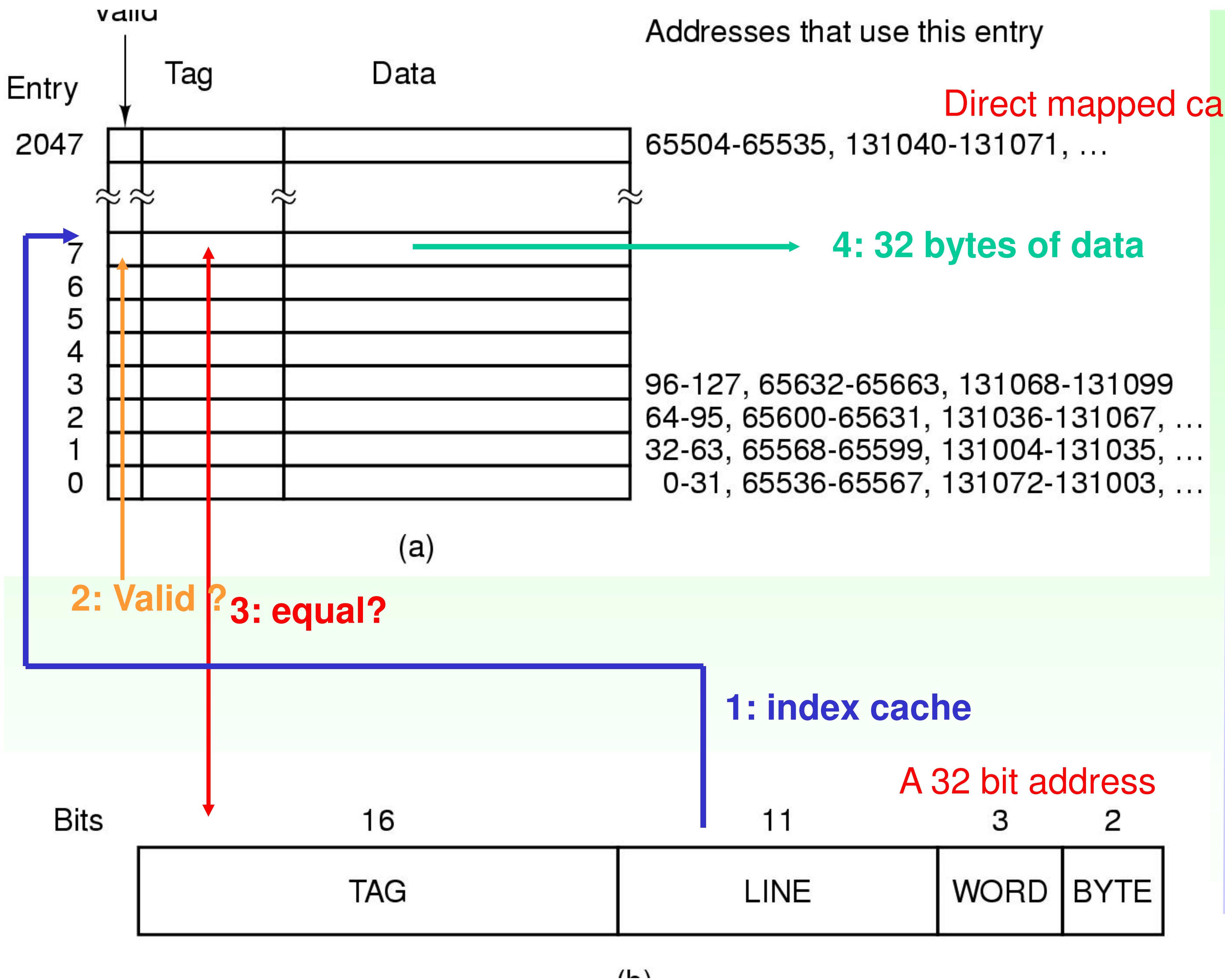
A 32 bit address

32 bytes in a line

4 bytes in a word

Direct Mapped Caches

Direct mapped cache



65504-65535, 131040-131071, …

96-127, 65632-65663, 131068-131099

64-95, 65600-65631, 131036-131067, …

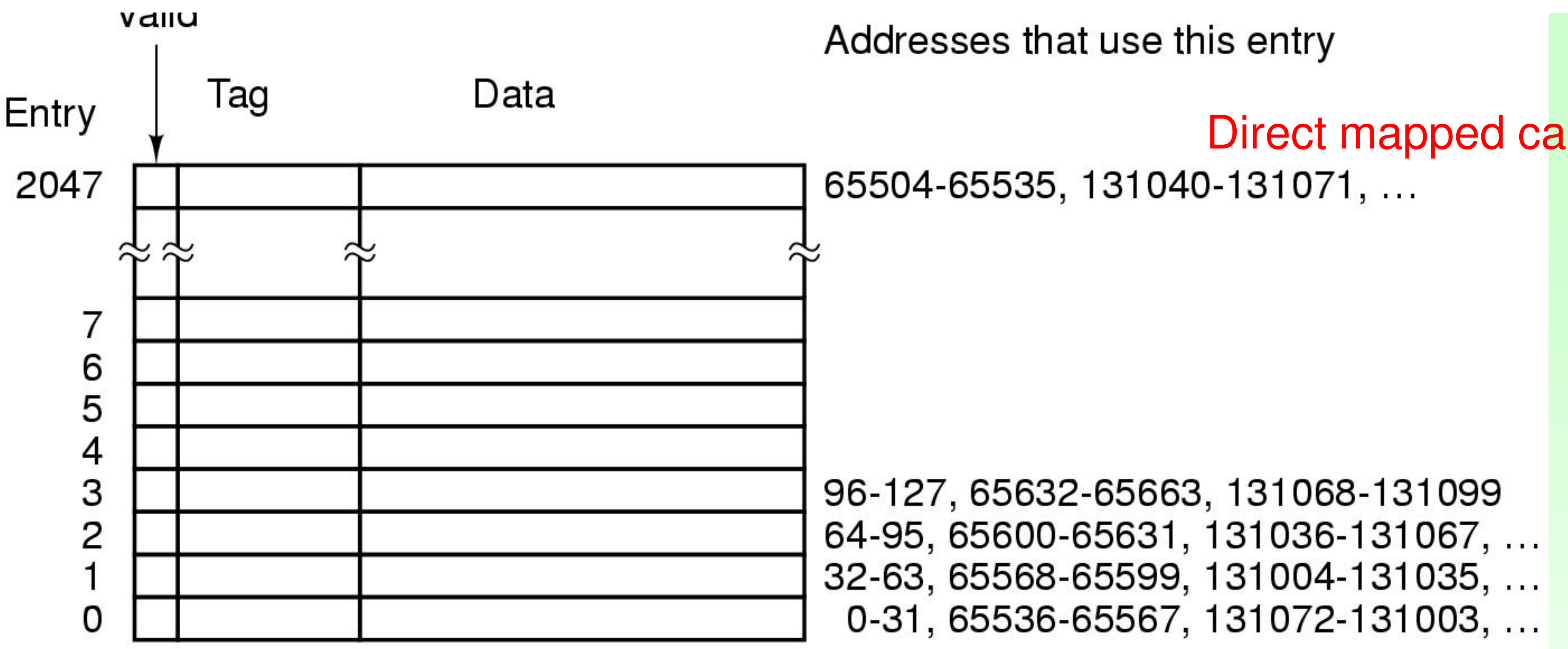
32-63, 65568-65599, 131004-131035, …

0-31, 65536-65567, 131072-131003, …

A 32 bit address

Direct Mapped Caches

Direct mapped cache



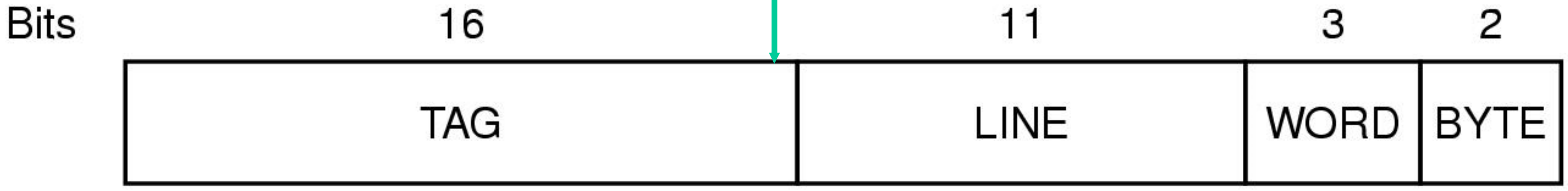
(a)

tag

Cache **cannot** store address X **and** x + n*2^{16}

2^{16} = 65536

A 32 bit address

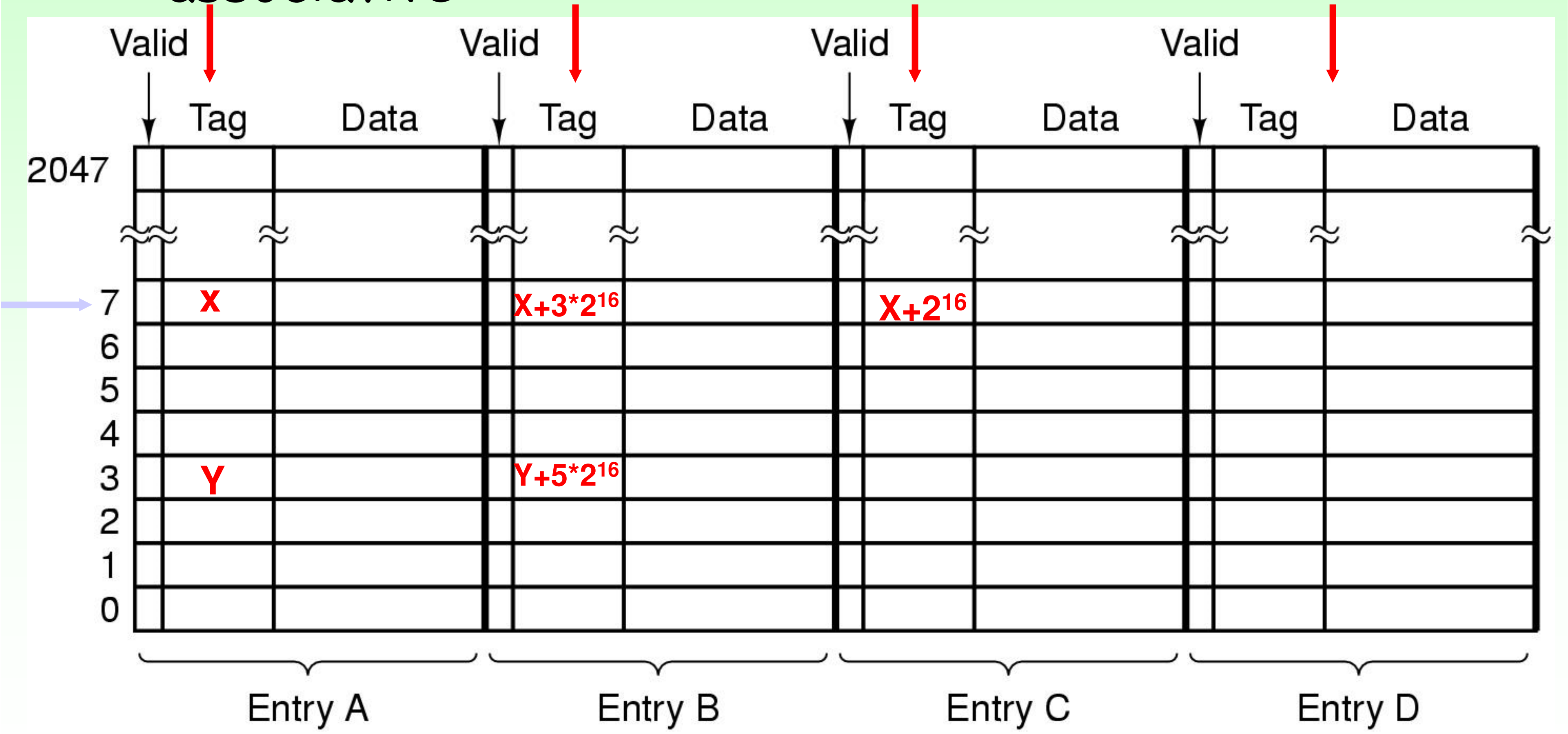


(b)

Set Associative Caches

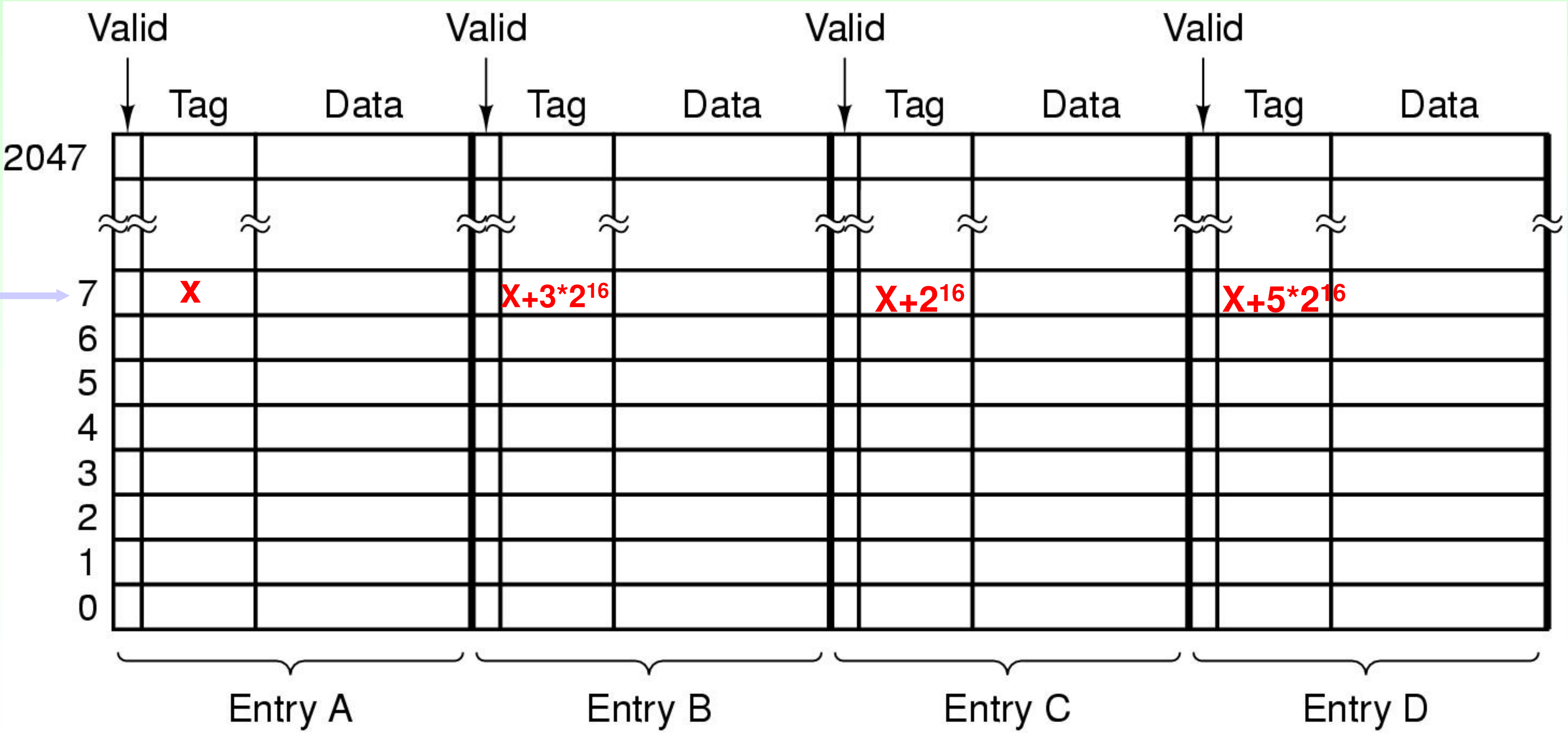
n is normally 2 or 4

- Allows two or more cache lines in each
- A cache with n lines per entry is called n-way associative



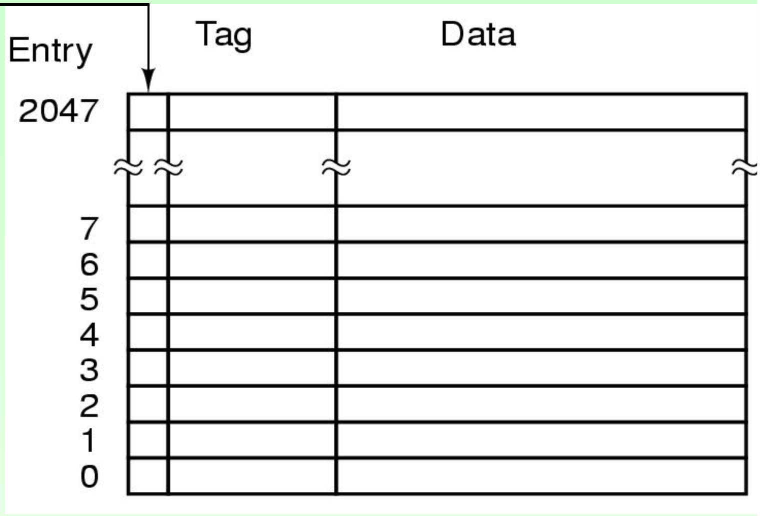
Set Associative Caches

- What if a line is full (all n-entries used) ?
- Overwrite Least Recently Used - LRU



Write operations : a special problem for caches

- Write a word
  - Either update the cache or make entry invalid
  - Usually entry is updated
  - What about main memory?



- Write immediately = **write through**
  - Simpler to implement
  - More reliable -> memory is always up-to-date
  - Consider errors -> recover the state of memory
- Write later (on re-use of Cache line) = **write deferred**
  - Used by more sophisticated implementations

write miss

- Writing data which is **not** in cache - do we load it into cache?
  - Yes : most "write deferred" implementations use **write allocations**
  - No: "write through" implementations would complicate a simple design

End of Lecture