Computer Architecture

R. Poss
Computer Systems Architecture group (UvA)
e-mail: r.c.poss@uva.nl

What is memory?

- A device with a single address/data/command interface and a protocol:
  - "a load from address X returns the value Y written by the most *recent* store to the same address X"

What is a cache?

Hint: data for loads flows like rivers from memory (upside) to processor (downside)

- Two “sides”:
  - **downstream**: the processor side
  - **upstream**: the memory side
- On the downstream side the cache behave likes a memory from the processor’s perspective
- On the upstream side the cache behaves like a memory client to the upstream memory’s perspective

What is a cache? (cont)

- Caches follow the memory protocol downstream: “a load from address X returns the value Y written by the most *recent* store to the same address X”
- However they have 2 extra “features”:
  - they answer requests of the downstream client at a **faster or equal rate** than the memory upstream
  - they **reduce the number of requests**: fewer requests sent upstream than received from downstream

Caches and design

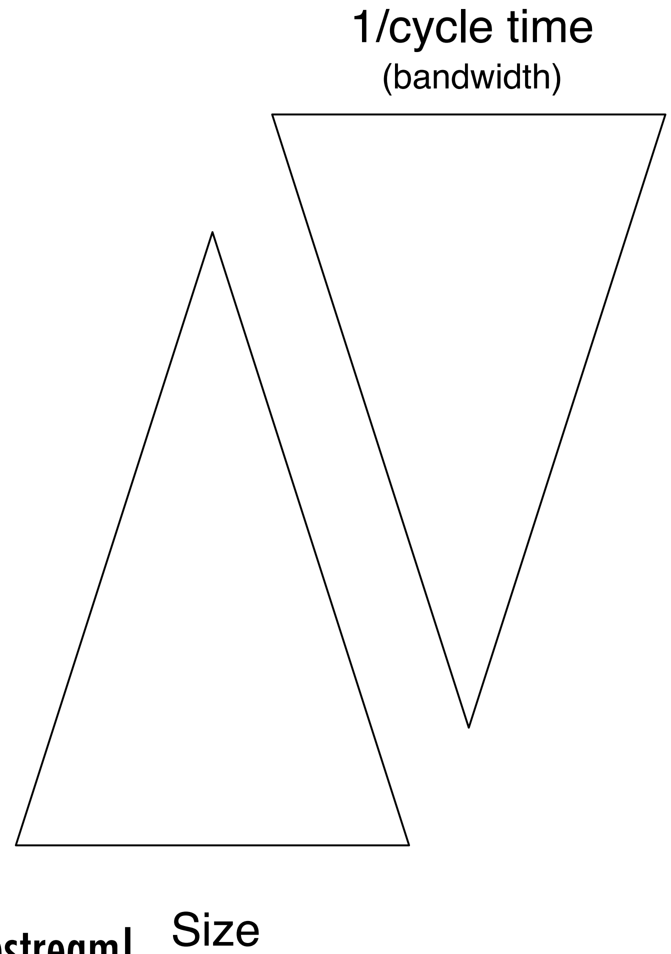
cf. Henessy & Patterson, Chap. 5

Caching - summary

- Caches are small fast memories that store recently used data close to the processor (usually on-chip)
- As the memory wall has grown, the number of *levels of cache* between main memory and the processor has increased
  - **from 0 to 1 to 2 and now some systems use 3 levels**
- Caches are largely transparent to the programmer
  - **but programmers must be aware of the cache while designing code to ensure regular access patterns**

The processor’s memory hierarchy

	100MHz clocks	GHz clocks
Registers - on-chip SRAM	< 1	< 1
L1 cache - on-chip SRAM	1-2	1-6
L2 cache - on-chip SRAM	2-6	$\simeq$10
off-chip SRAM	4-8	≥10
L3 cache - off-chip SRAM	$\simeq$10	≥100
Main memory - DRAM	≥10	≥1000
Distributed memory	≥100	≥10000



<u>NB: despite the orientation, this side is upstream!</u>

Cache operation at multiple levels

- Caches contain copies of *blocks of data* from main memory - **cache lines**
- Reads to memory go up the memory hierarchy
  at each level a check is made to determine if the data is present at that level
  - **Cache hit** - the required data is in the cache: the data is taken from that level and propagated down the hierarchy (in the direction of processor)
  - **Cache miss** - the required data is not in the cache: the request goes up a level until found
  - A cache miss at any level may overwrite old data when the requested new data is propagated down the hierarchy - "*thrashing*" occurs when the old data is needed shortly
- Similarly, when data is written to the cache, it is written back to main memory either immediately, when space is required in the cache, or, in a multi-processor system, when another processor requires it.

Caching principles

- Caches provide reuse of recently fetched data tramsparenly to the programmer or compiler
  - **Shorter delay of access to same data after the first access to a longer delay memory**
- Caches rely on the principle of locality:
  - **Temporal locality** - information that has just been used is likely to be used again in the future.
  - **Spatial locality** - because a cache line contains more than one word of data, words close to the original miss will now be resident in the cache and may be accessed without further penalty.
  - The former requires *frequent* access to the *same data*
    the latter requires *regular access patterns* to memory
    e.g. regular small strides through memory – e.g. consecutive words

From the programmer's perspective

- The major problem is that not all codes exhibit the locality property...
  - a non-indexed scan in a large database may contain some spatial locality but it has no temporal locality
  - if the DB record is as large as a cache line, then not even spatial locality will be observed, each record will require a RAM read and a key check
  - Because of the implicit nature of their use, **without locality it is as if the cache did not exist at all and all accesses to memory are as slow as the slowest component**
- There is a trend (e.g. IBM's Cell) to use explicit local and global RAM with explicit mapping of data by programmer/compiler, but this makes programming more difficult & non portable

Cache design issues

- Caches can be:
  - **Unified** or **separate** w.r.t. data and instructions
    - L1 cache normally separate and L2/L3 normally unified
  - **Write through** - data is written to cache and also sent to the upper level
  - **Write around** - data is sent to upper level but not written to cache
  - **Write/Copy back** - data is written to cache but sent up the hierarchy: the upper level memories may become *inconsistent* with respect to program state
    - Copy back is used in multi-processor systems: a write around/through strategy can consume a large amount of bus or network bandwidth
    - How to maintain coherence between multiple copies?
- Lower levels of cache are normally write around/through

Level 1 cache miss

- A processor's data-path will contain two level-1 caches for concurrent data and instruction access – *pipeline operation*
- Hitting this cache is very important for performance:
  - **The cache (I or D-cache) hits if the required data is present - then data is typically accessed in a single cycle**
  - **On a miss the pipeline will stall; in the worst case until a higher level of memory hierarchy hits and provides the required data**
  - **This may require a read to the main memory**
  - **Only then can the pipeline continue the stalled instruction**
- Some processors allow multiple concurrent accesses to memory by allowing instructions to issue and/or complete out of programmed order - we will come back to this later

Mapping from memory to cache

- The line or block size is the unit of data managed by the cache typically 32-256 bytes
  - each line has a **tag** (from its address) stored in the cache and used to determine which memory block is mapped to the cache line
- A **cache mapping** determines which line(s) in a cache an address in memory can mapped to:
  - **Direct mapped** (simplest) yields a unique line in cache for any given block in memory - based on its address
  - **Fully associative** (most complex) allows any memory block to be mapped to any cache line
- Associative addressing is expensive; **Set-associative** cache gives a compromise between these extremes
  for example a "4-way set associative" cache has sets of 4 lines where a line may be mapped to
- Associative mapping requires concurrent tag matching to find a line in a single memory cycle

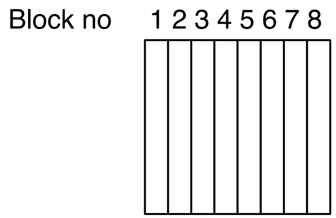
Cache lines

state	tag	Data

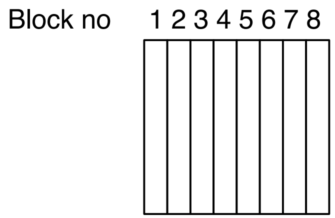
- The tag comprises enough address information to identify which block of memory the cache line holds
  - **The bits required depend on the mapping strategy**
- State used in algorithm to replace lines e.g. valid/invalid

Cache mapping - example

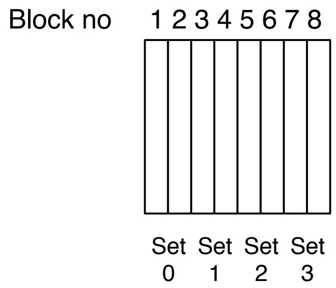
Fully associative



Direct mapped



2-way set associative

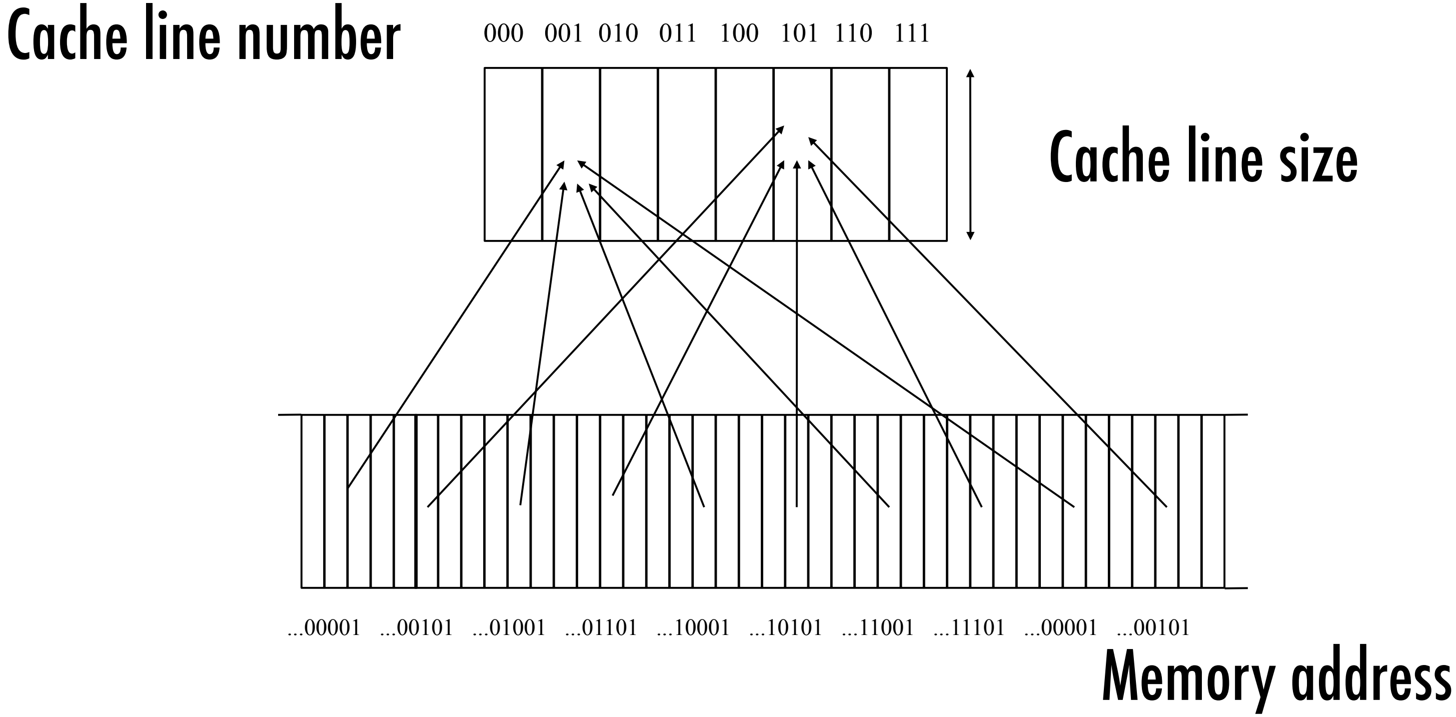


For the memory address 386, 32-byte cache lines and an 8 line cache:

<block addr> = floor(<mem addr> / <cache line size>) = floor(386 / 32) = 12

- Direct mapped: line = <block addr> mod <nr. of lines> = 12 mod 8 = 4
- 2-way set associative: <nr. of sets> = <nr. of lines> / <set associativity>
  set = <block addr> mod <nr. of sets> = 12 mod 8/2 = 0
- Fully associative: one set of 8 lines, so anywhere in cache

Direct mapped caches



Direct-mapped caches

- A direct mapped cache is simple and fast
- ...but has problems from its inflexibility in mapping
  - Address strides (differences between consecutive addresses) of a multiple of the cache line size map subsequent accesses (to different memory blocks) all to the same cache line – **even though other lines may be empty!**
  - This is called a **pathological access pattern**
- Direct mapped cache is often used as 2nd or 3rd level cache which is much larger and hence has less contention but the programmer must still be aware of this restriction

Evaluating cache performance

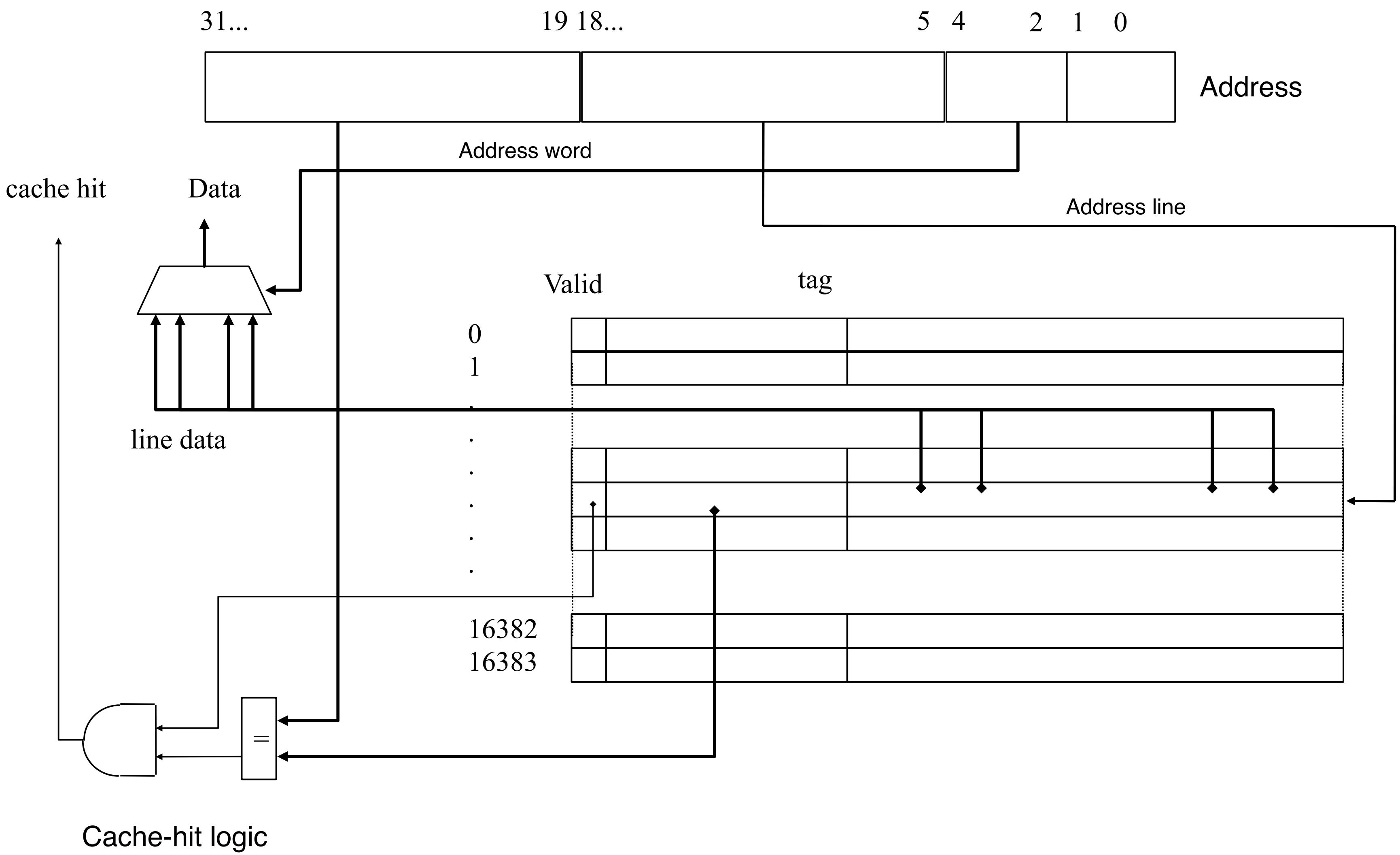
- As an exercise try the following...
  - Design a program to evaluate cache parameters of your workstation/laptop
  - Note that a significant difference in performance will be observed when data is being sourced from
    - L1
    - L2/3
    - Main memory

Direct-mapped cache addressing

31... 19	18... 5	4... 0
13-bit tag	0 .. 2^{14}-1	0 .. 32
tag	line address	byte in line

- E.g. a 32-bit byte address into a direct-mapped cache of size of 512KBytes and a line size of 32 Bytes (i.e. 16K lines) the address fields above comprise:
  - 5 bits of byte address (0..4) gives the byte offset in the cache line
  - 14 bits of cache line address (5..18) give cache line (16K direct mapped)
  - the remaining 13 bits (19..31) determine which block from the 8K possible memory blocks is mapped to the cache line
    tags stored in cache line, matched with the address from the processor to check hits

Example 4-byte access in DM cache



Write policies

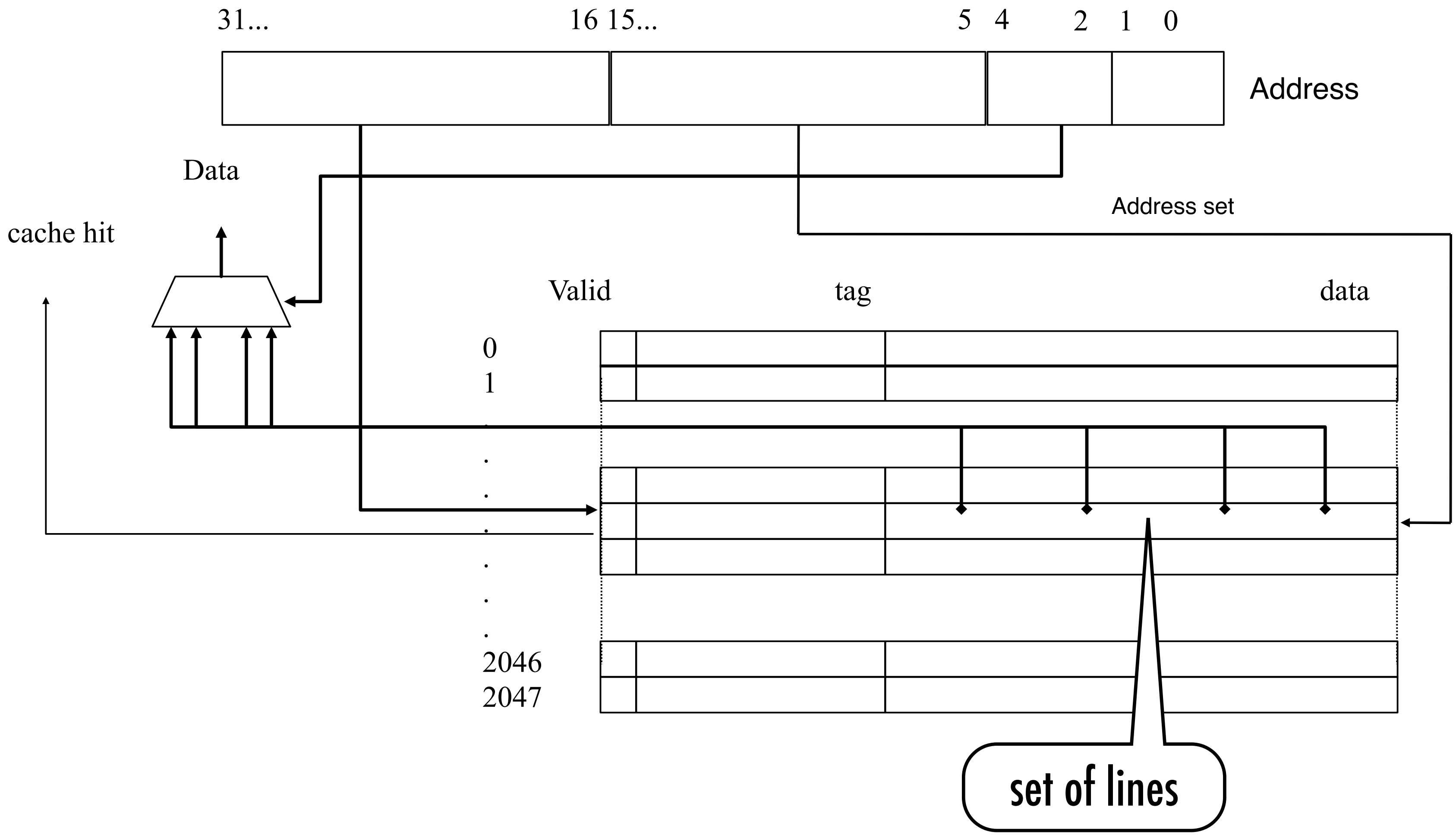
- **Write-through**: stores from CPU are copied to cache and simultaneously sent to memory
- **Write-back**: stores from CPU stay in the cache until the line needs to be replaced (causes **evictions**)
- **Write-around**: behaves write-back/write-through on hit (line already in cache); if conflict then store goes around the cache directly to memory, no copy kept locally

8-way set associative cache addressing

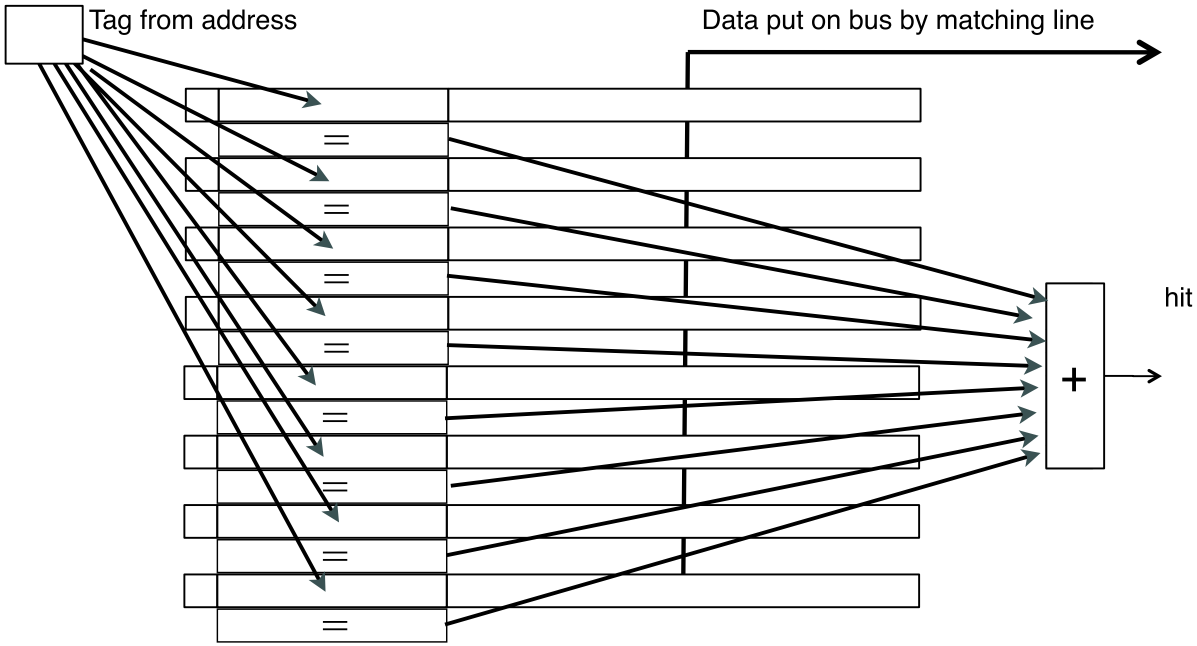
31... 19	18... 5	4... 0
16-bit tag	0 .. 2^{11}-1	0 .. 32
tag	set address	byte in line

- E.g. a 32-bit byte address into an 8-way set associative cache of size of 512KBytes and a line size of 32 Bytes (i.e. 16K lines):
  - 5 bits of address (0..4) gives the byte offset in the cache line
  - 11 bits (5..15) address 2K sets of 8 cache lines (16K lines total)
  - 16 bit tag (16..31) determines which block from the 64K possible memory blocks is mapped to one of the cache line in that set;
    stored as tag in the cache line and matched with the address from the processor

4-byte access in 8-way set associative cache



Line sets in associative caches



8 tags compared in parallel

Fully associative cache addressing

- E.g. a 32-bit byte address into an fully associative cache of size of 1KBytes and a line size of 32 Bytes (i.e. 32 lines - fully associative means each line requires a comparitor):
  - 5 bits of address (0..4) gives the byte offset in the cache line
  - 27 bits (5..31) determine which block from the 128M possible memory blocks is mapped to one of the cache line in that set
    stored as tag in the cache line and matched with the address from the processor

Access to fully associative cache