1. Memory technology & Hierarchy

Caching and Virtual Memory

Parallel System Architectures

Andy D. Pimentel

Caches and their 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 many 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	≃10
off-chip SRAM	4-8	≥10
L3 cache - off-chip SRAM	≃10	≥100
Main memory - DRAM	≥10	≥1000
Distributed memory	≥100	≥10000

Cache operation at multiple levels

- Caches contain copies of blocks of data from main memory - **cache lines**
- Reads to memory go down 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 up the hierarchy
  - **Cache miss** - the required data is not in the cache: request goes down a level until found
  - A cache miss at any level may overwrite old data when the requested new data is propagated up the hierarchy - “thrashing” occurs when the old data is needed shortly
- 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 transparently 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

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 lower level
  - **Write around (no-write-allocate)** - data is sent to lower level but not written to cache
  - **Write/Copy back** - data is written to cache but not sent down the hierarchy: the lower 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?
- Higher levels of cache are normally write around/through

Mapping from memory to cache

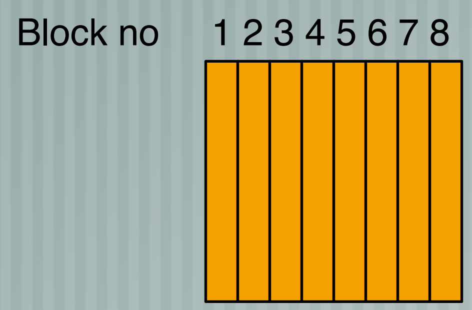
- The line or block size is the unit of data managed by the cache, typically 32-256 bytes
  - each line has a **tag** (from its address) stored in the cache and used to determine which memory block is mapped to the cache line
- A **cache mapping** determines which line(s) in a cache an address in memory 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
- **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

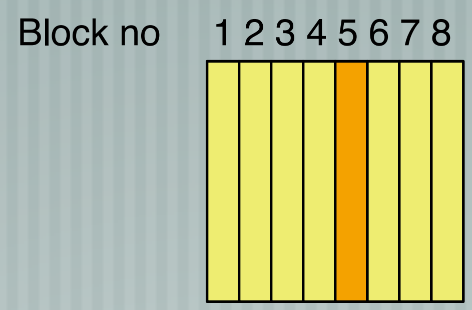
- The tag comprises enough address information to identify which block of memory the cache line holds
  - **The bits required depend on the mapping strategy**
- State used in algorithm to replace lines e.g. valid/invalid

Cache mapping - example

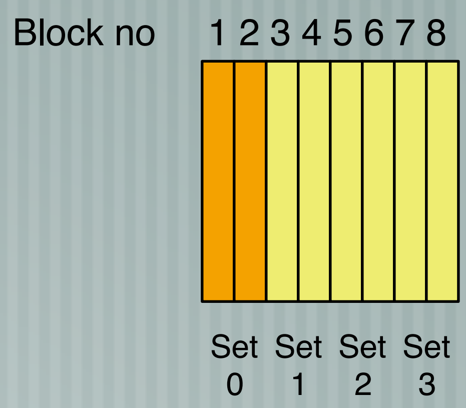
Fully associative



Direct mapped



2-way set associative

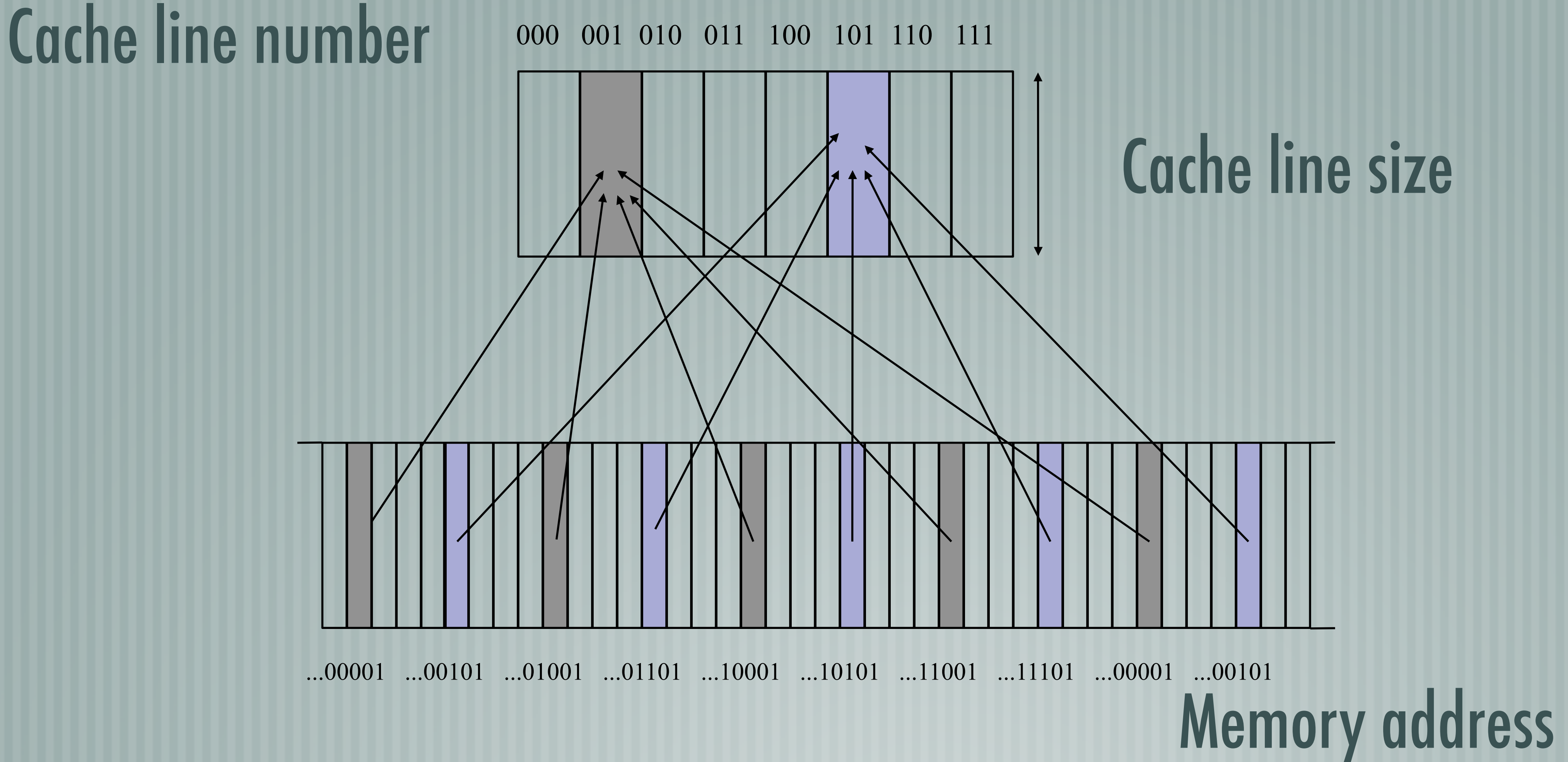


For the memory address 386, 32-byte cache lines and an 8 line cache:

<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

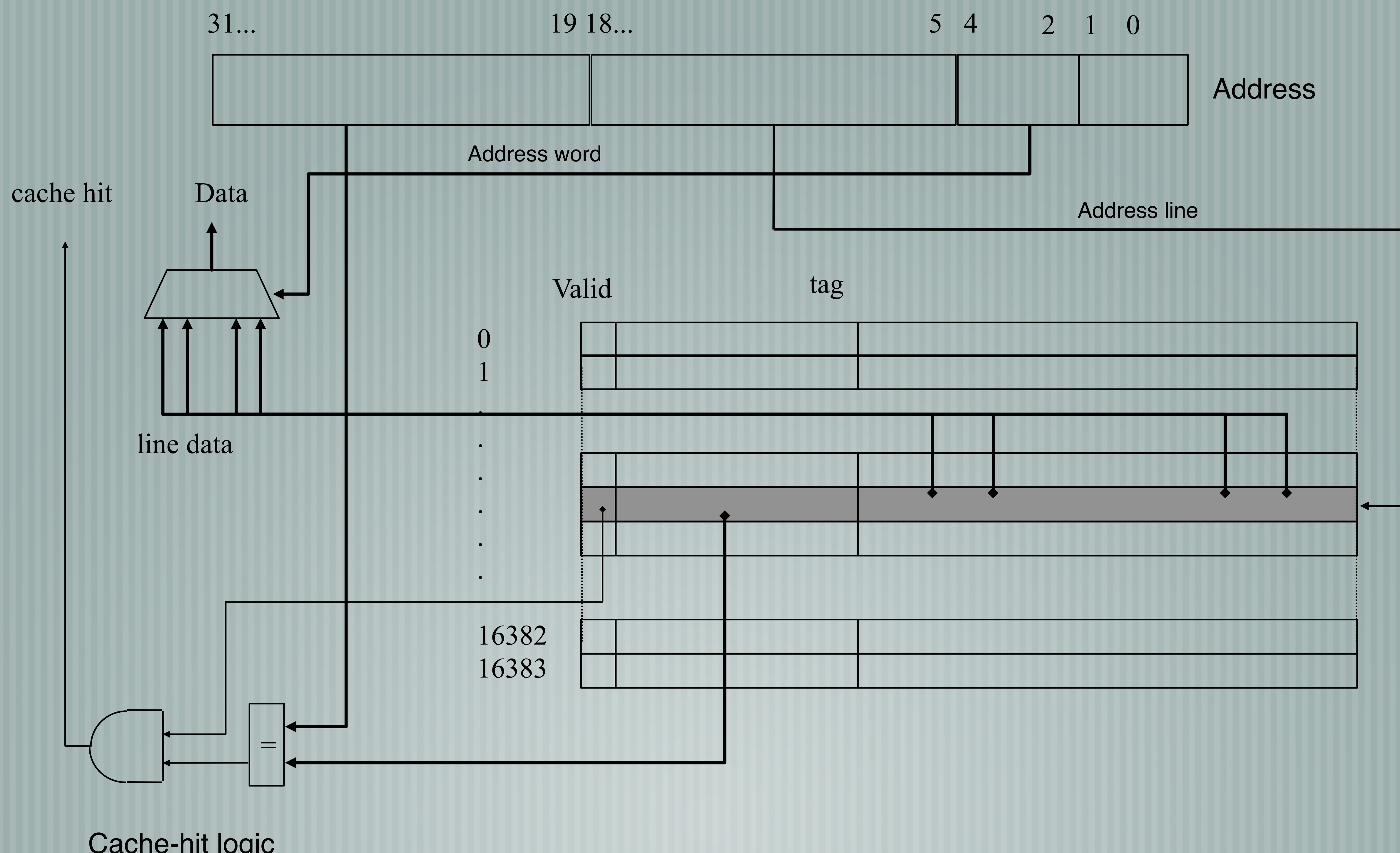
- A direct mapped cache is simple and fast
- ...but has problems from its inflexibility in mapping
  - Address strides (differences between consecutive addresses) of a multiple of the cache line size map subsequent accesses (to different memory blocks) all to the same cache line – **even though other 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

Direct-mapped cache addressing

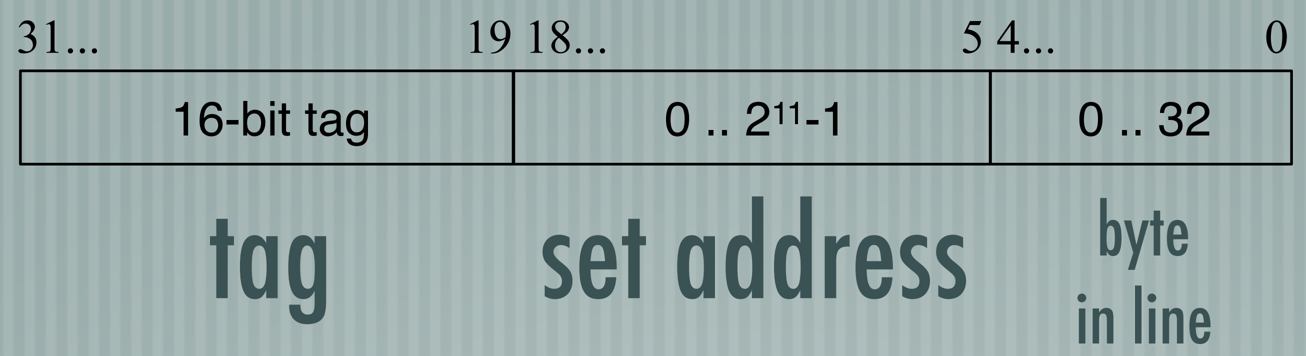
31... 19	18... 5	4... 0
13-bit tag	0 .. 2^{14}-1	0 .. 32
tag	line address	byte in line

- 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

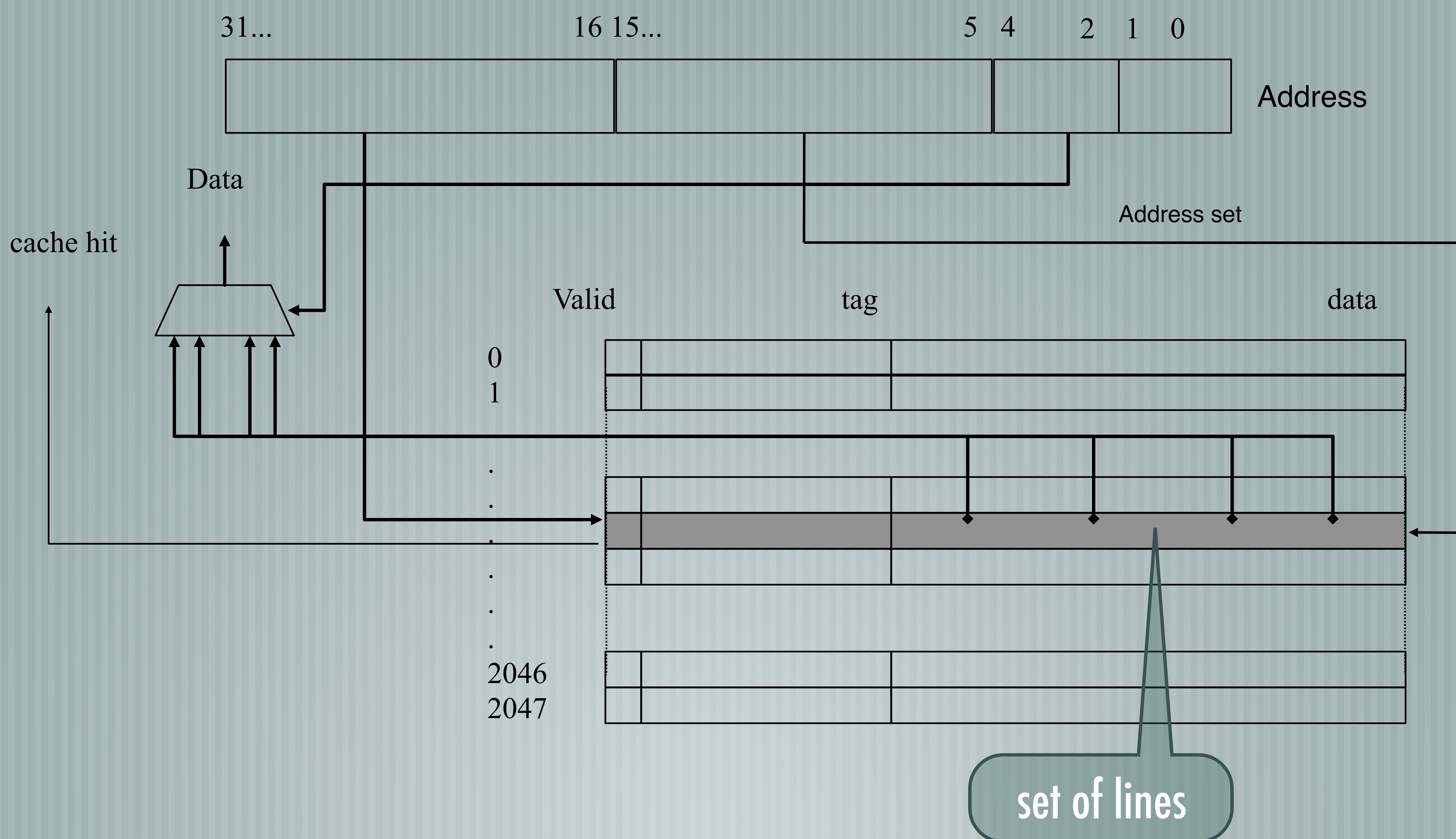


8-way set associative cache addressing

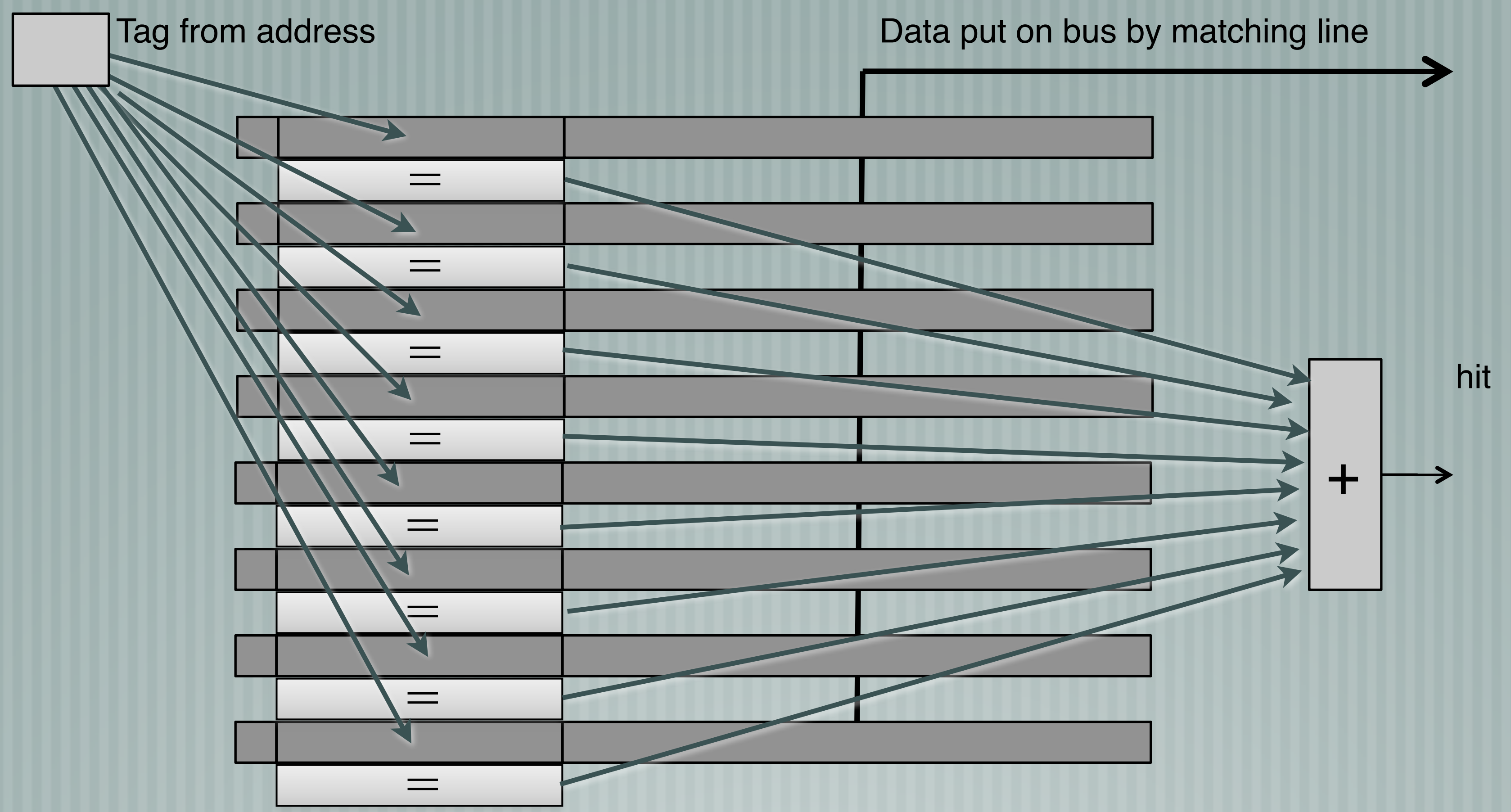


- E.g. a 32-bit byte address into an 8-way set associative cache of size of 512KBytes and a line size of 32 Bytes (i.e. 16K lines):
  - 5 bits of address (0..4) gives the byte offset in the cache line
  - 11 bits (5..15) address 2K sets of 8 cache lines (16K lines total)
  - 16 bit tag (16..31) determines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

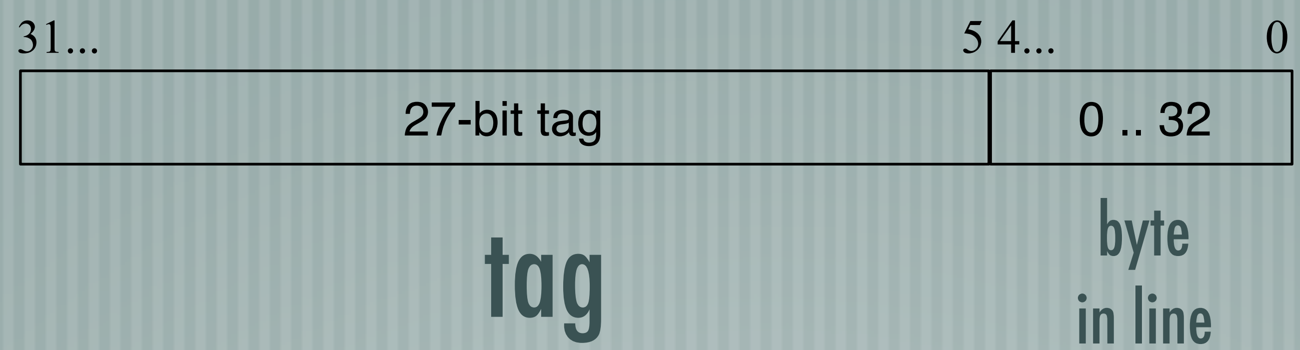


Line sets in associative caches



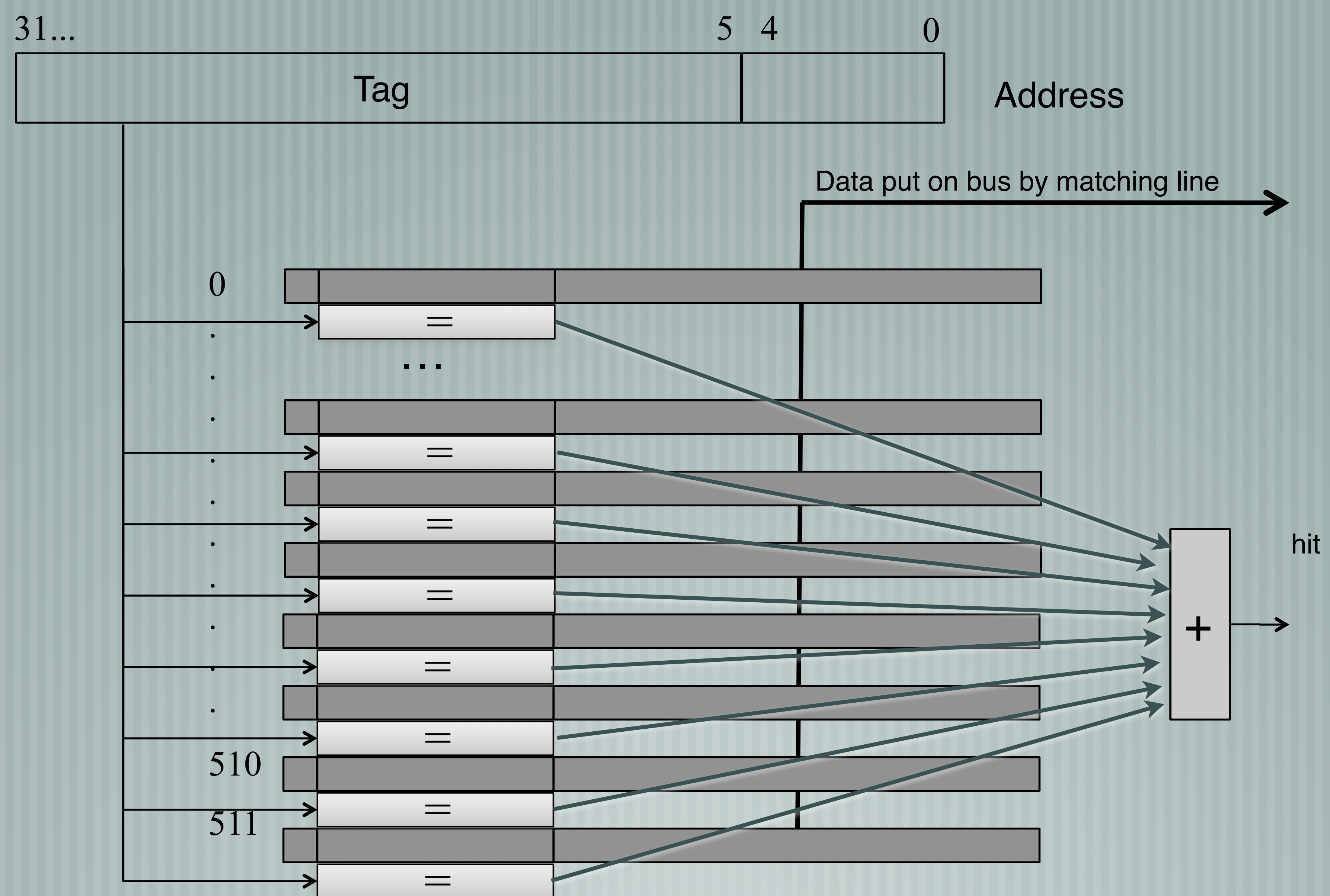
8 tags compared in parallel

Fully associative cache addressing



- E.g. a 32-bit byte address into an fully associative cache of size of 16KBytes and a line size of 32 Bytes (i.e. 512 lines - fully associative means each line requires a comparator):
  - 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



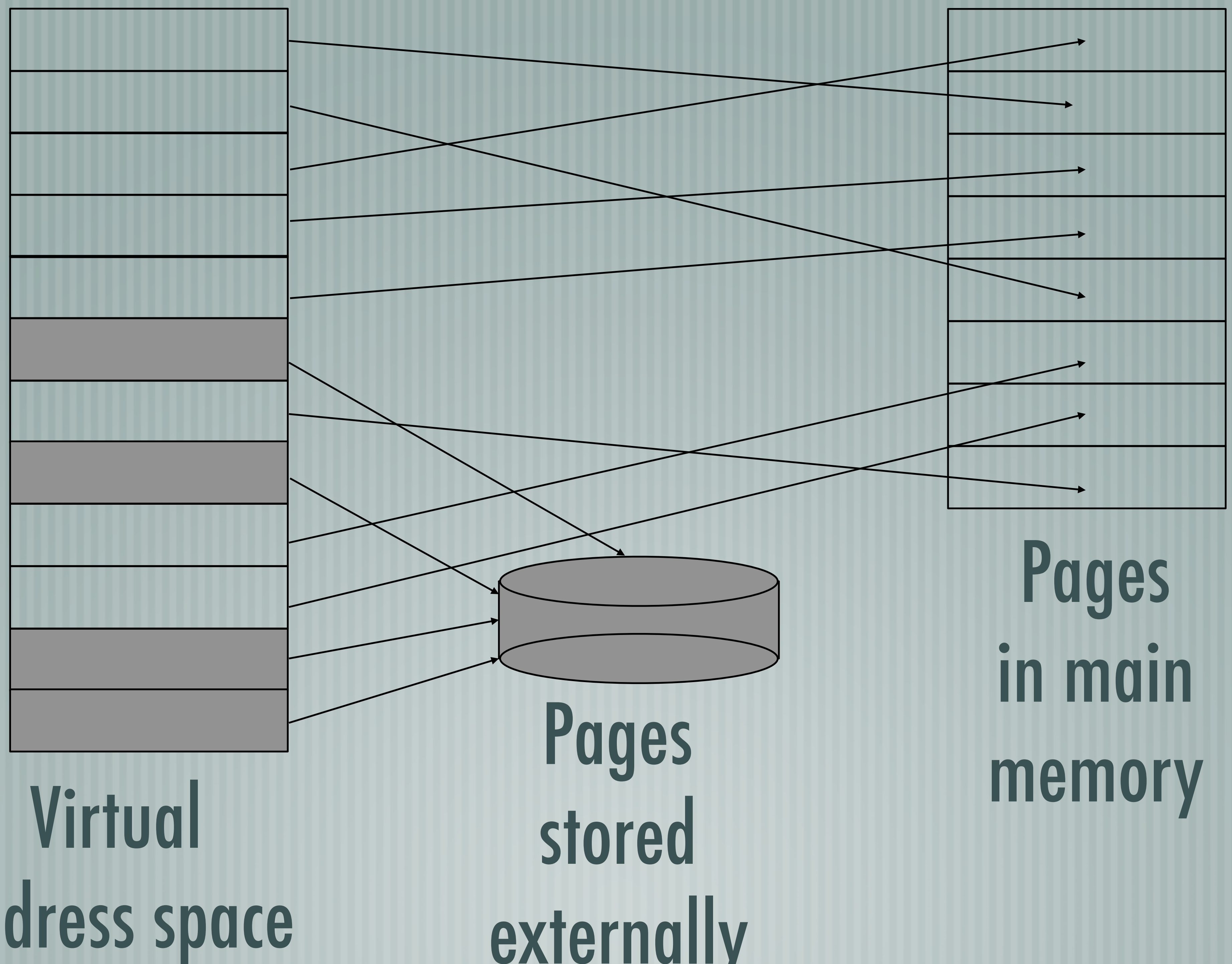
Virtual Memory

cf. Henessy & Patterson, App. C4

Virtual Memory

- It is easier for the programmer to have a large virtual memory than to program explicit I/O due to memory limitations
- Also in multi-programming memory is shared between many programs, some suspended or inactive for a while
  - **only a small fraction of virtual memory is used at any one time in a multi-programming environment**
- Virtual memory uses main memory to store only part of the larger virtual memory space and the remainder is held on external storage, e.g. discs
  - The unit exchanged between memory and disc is called a **page**

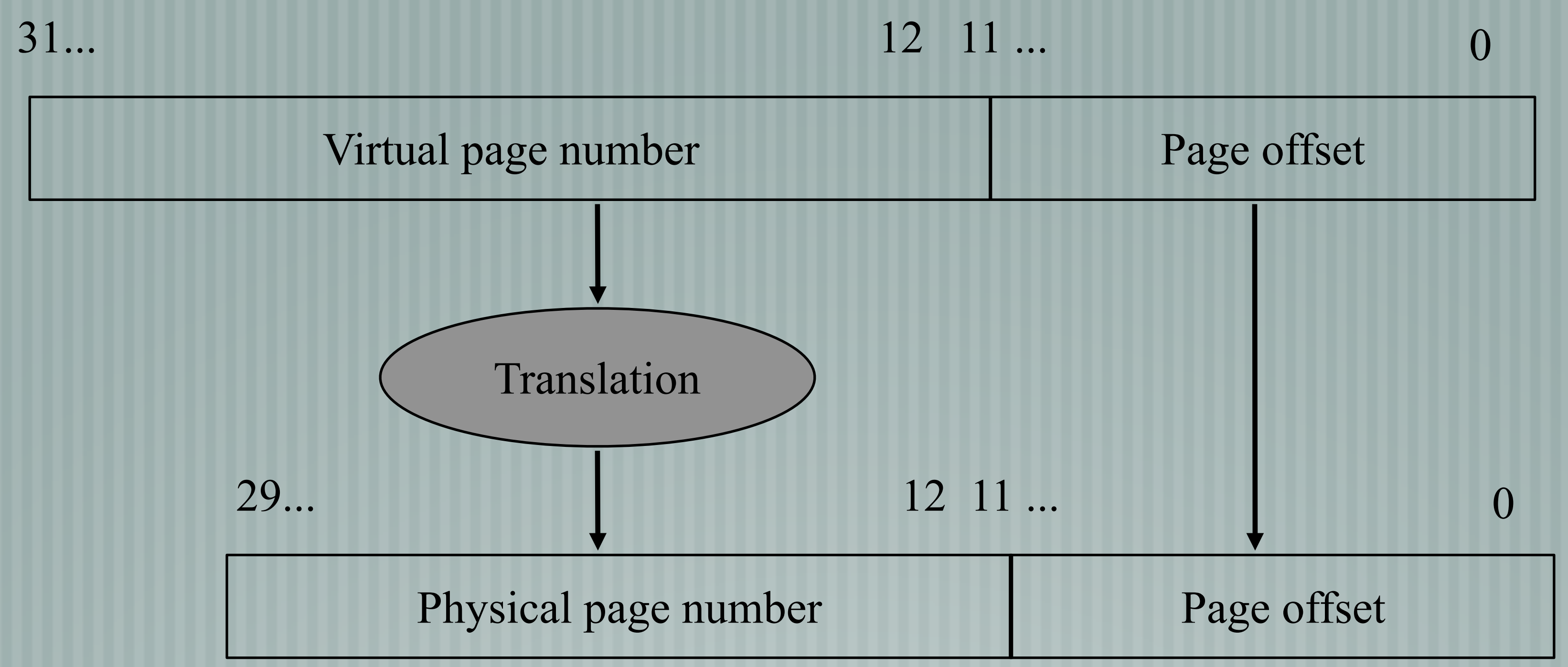
Virtual memory mapping



VM Terminology

- The address produced by the processor is called a **virtual address**
- This gets translated by a **MMU** via a **page table** into a **physical address** (PT hit) or **page fault** (PT miss)
- The page table is in main memory but has a special cache called a **TLB** (translation look-aside buffer)
  - Page faults usually managed by a software trap to an operating system
  - This mapping process is called **address translation**

VM Address translation



- This shows address mapping from a 4 GiB virtual address space onto in a 1 GiB physical address space using 4KiB memory pages
- The translation is performed using a 1M entries (3MiB) table in memory, addressed by the virtual page number

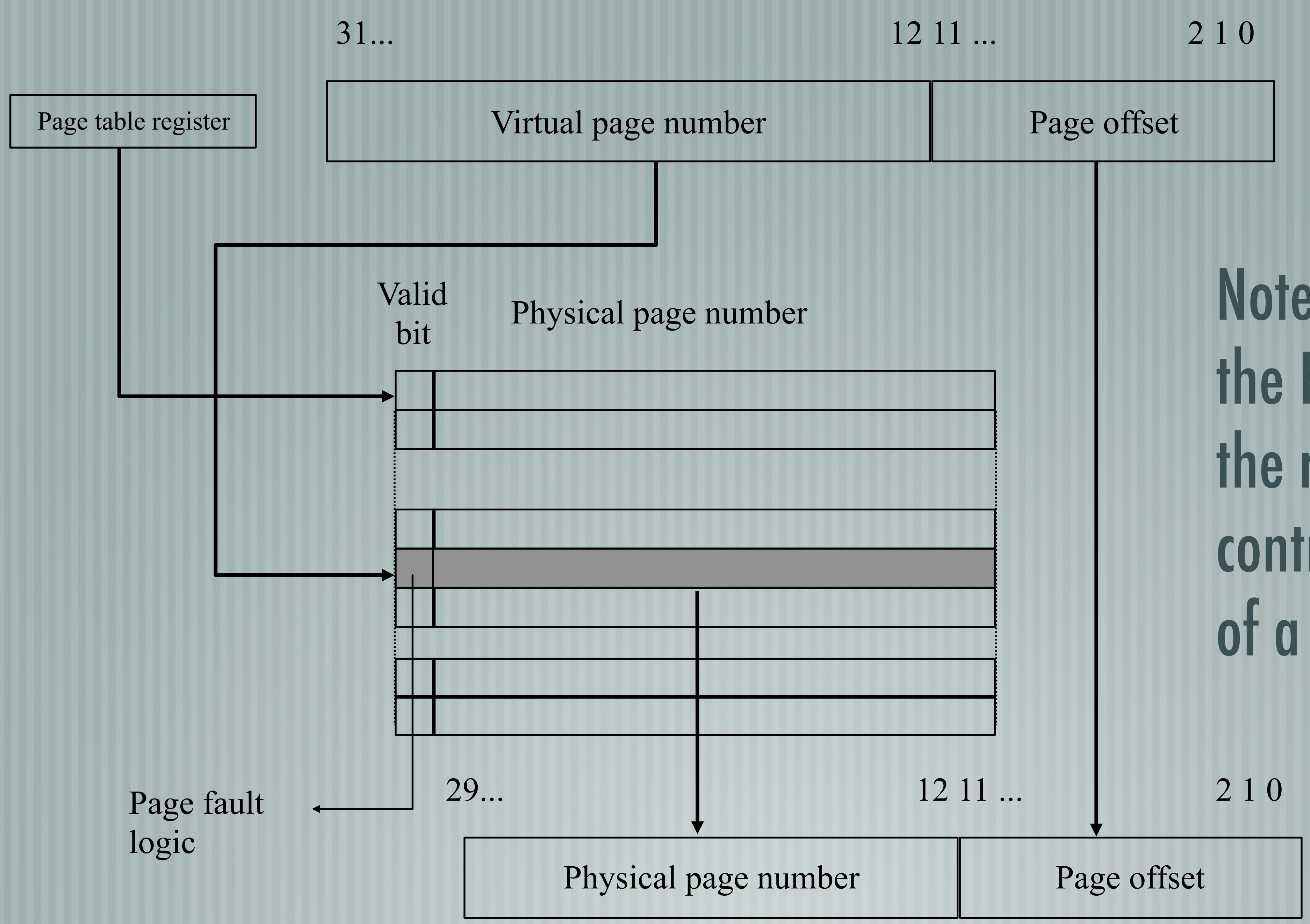
Virtual memory issues

- Need flexibility in page placement to avoid costly page misses
- Unlike cache mapping, VM mapping is implemented as a table in main memory - allows arbitrary mapping
  - indexed by virtual address
  - that yields the physical address
- Page misses are handled by software and incur a large penalty
- Pages must be **sufficiently large to amortize this large overhead and to minimize the mapping table size**
  - 4 to 64KByte is a typical page size
    with variable size pages can be as large as 1MByte

Replacement, processes and protection

- Sophisticated algorithms for placement can be coded in software
  - pages known to be often required can be locked down
- Each **process** has its own virtual address space and page translation
  - this means programs can not interfere (read/write) the memory of any other
- To achieve **protection**, user code must be prevented from altering the page tables
  - This is normally achieved by having different modes of operation (eg. user mode vs. kernel mode)
  - alternatively, using security capabilities on the page table data

Page table



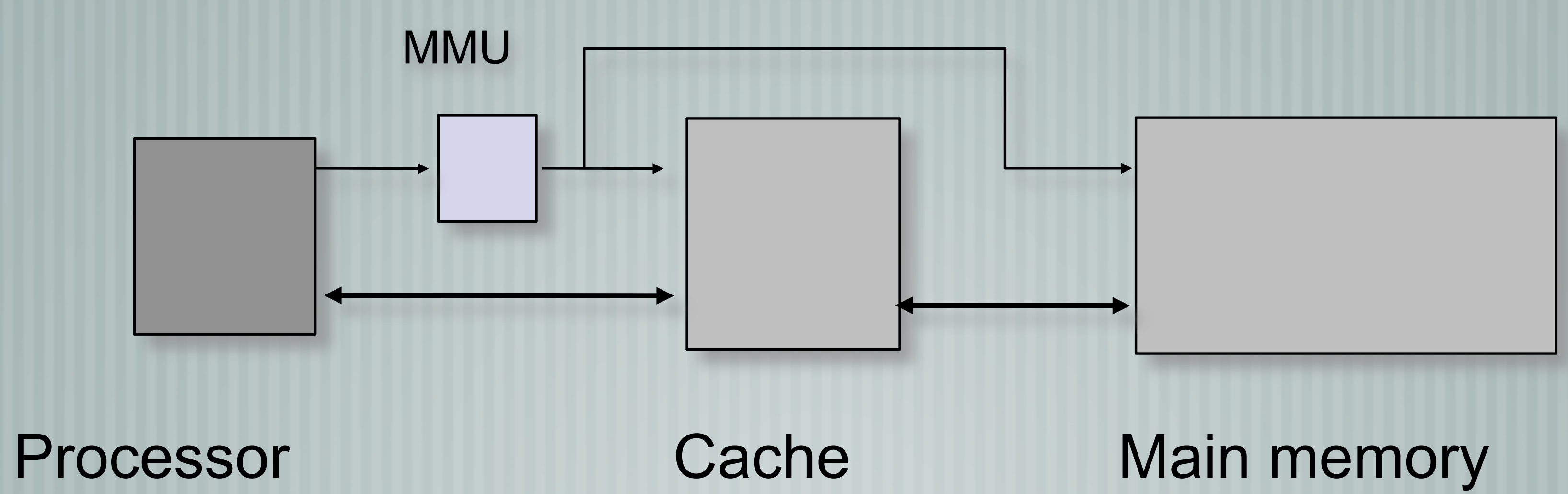
Note: the page table, the PC and the state of the registers all contribute to the state of a program

Translation Look-aside buffers

- Translation Look-aside buffers (TLB) cache the page table in small fast memory
- NB: The page table is too large to be held entirely in fast memory
- Without the TLB, access to memory would be twice as slow
  - One access to the page table for address translation
  - One to the data itself
- Address translation and L1 cache access can be performed in one or two processor cycles (as long as we get a cache hit)
- **Big question: which memory space do we cache: Virtual or Physical?**

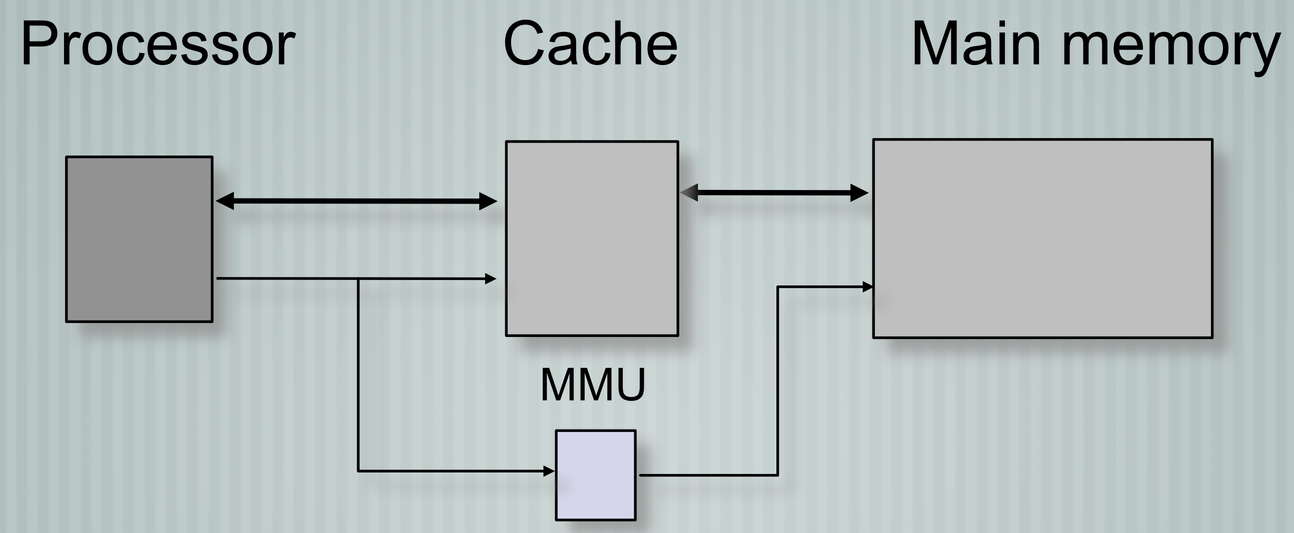
Physically addressed caches

- Addresses translated by memory management unit (MMU) before cache lookup
- Sequential - even with a TLB and cache hit,
  access can be slow as it requires sequential memory accesses



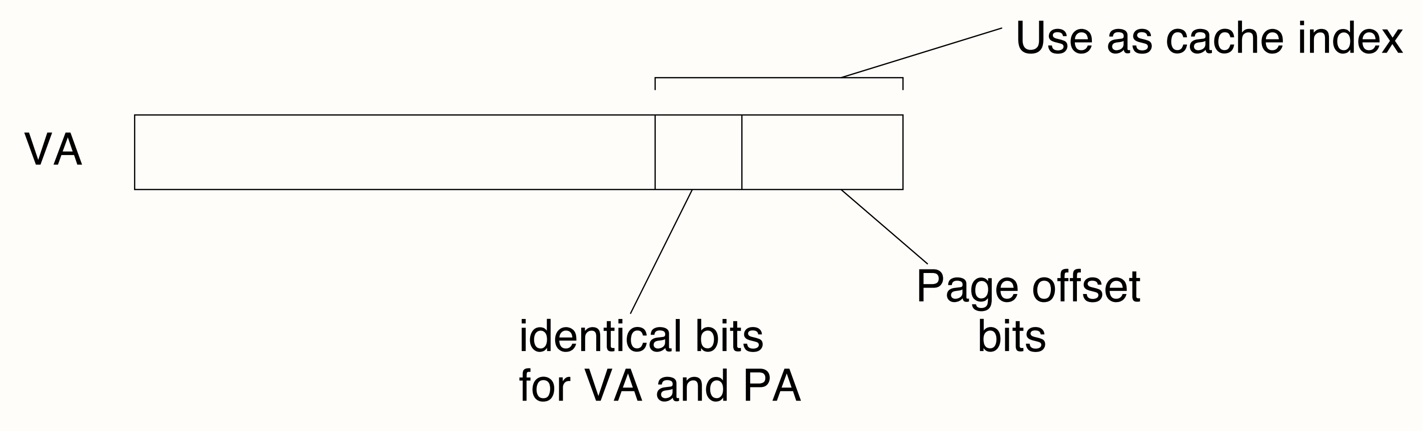
Virtually addressed caches

- Addresses translated by MMU in parallel with cache lookup
- **Aliasing** – is where the same virtual address in different processes maps to the same location in cache
- Context switching therefore requires a full cache invalidation (time expensive) or a process identifier in the tag (space expensive)
- Aliasing is averted if all processes share the same virtual address space



Virtually-indexed, physically tagged cache

- Cache indexing during address translation
- Page offset bits in virtual address used for cache index
  - Number of sets in cache limited (dependent on page size)
  - Solutions: larger sets or page coloring (OS support)



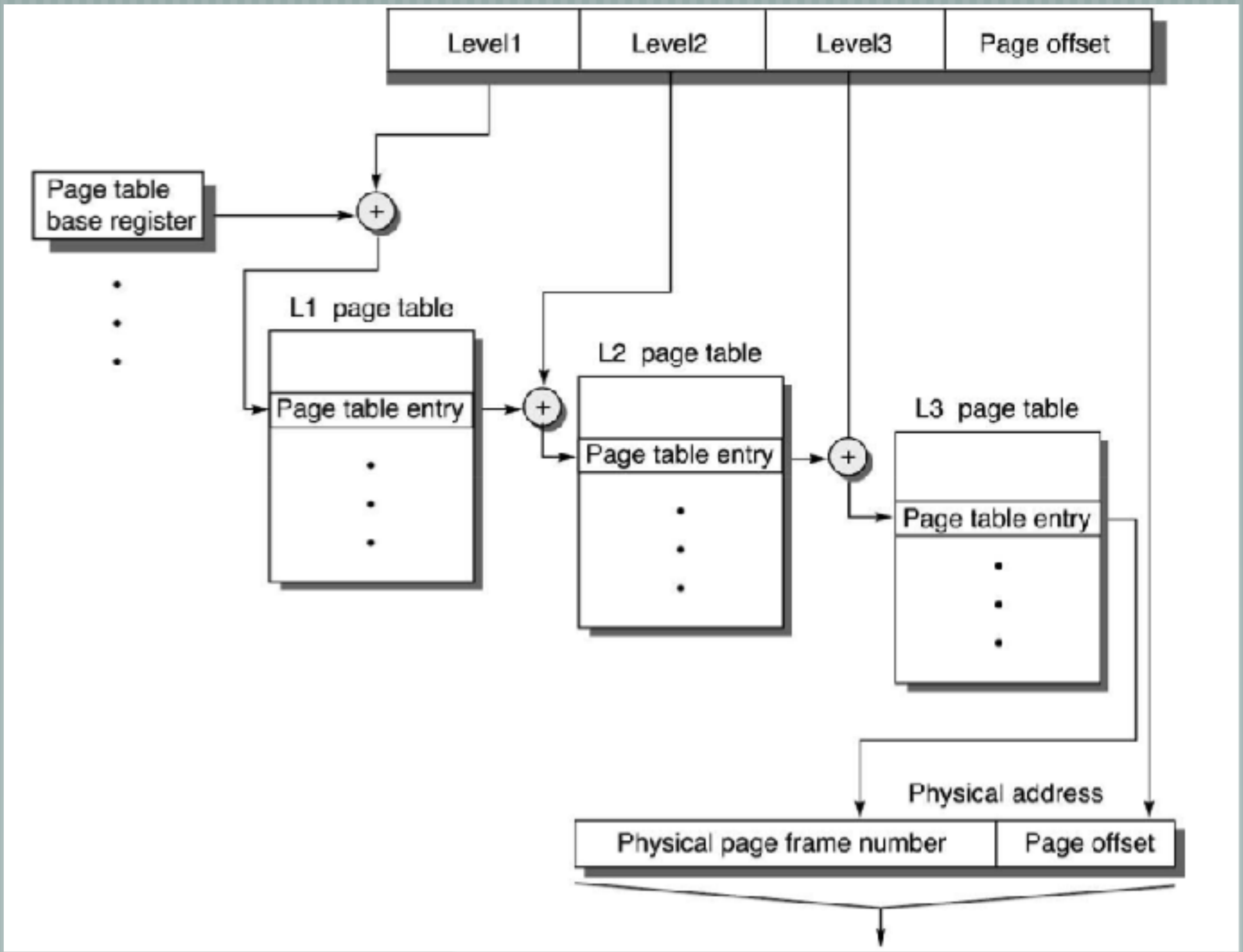
Page table size

- The example earlier was for 32-bit addresses and yielded a 3MiB table
- For a 64-bit architecture and say a 48-bit virtual address and 4KiB pages we get:
  - table size = $2^{48}/2^{12} = 2^{36}$ entries = 2^{39} bytes = 512GiB!!
  - and this is replicated for each process (!!)
- Solution is to **grow page table as required**
  - keep limit and check limit on each access
  - increase (e.g. double size) on each overflow

Page table size

- Address usage may be **sparse**
- Another solution is to use a **multi-level page table** as this takes advantage of sparseness
  - e.g. use very large pages and keep a table of these
  - within a large page keep a table of smaller pages (e.g. 4KiB)

Multi-level page table



Summary

Memory system summary

- The memory hierarchy is a critical component of both computer and algorithm design in determining performance
- A major problem is the rapidly increasing on-chip or processor clock rates and the relatively slow change in memory cycle times
  - **DRAMs are designed for density not speed**
- Caching works well with regular accesses to memory, but some applications do not possess this property - in this case we see the performance of the main memory system which may be 10-100 times slower than the processor performance!
- **New architectures will be needed, as the memory wall gets taller, that exploit latency tolerance to avoid memory-limited performance**