Characteristics of a memory system

- Location
  - Internal (e.g. main memory, cache, processor registers)
    - Accessible with a single machine instruction
    - Typically volatile memory
  - External (e.g. optical disks, magnetic disks, tapes)
    - A function is needed for access
    - Typically nonvolatile
- Capacity
  - For internal memory expressed in terms of bytes (1 byte = 8 bits) or words
  - External memory capacity is expressed in terms of bytes
  - For individual memory chips is expressed in bits (for example in memory chip data sheets)
- Performance

Memory performance

- Access time (latency)
  - The time it takes to complete memory access – time from the instant that the address is presented to the memory to the instant that data is made available or has been stored to the memory location
- Memory cycle time
  - Access time plus any additional time required before a second access can commence on the system bus
- Transfer rate
  - The rate at which data can be transferred into or out of the memory

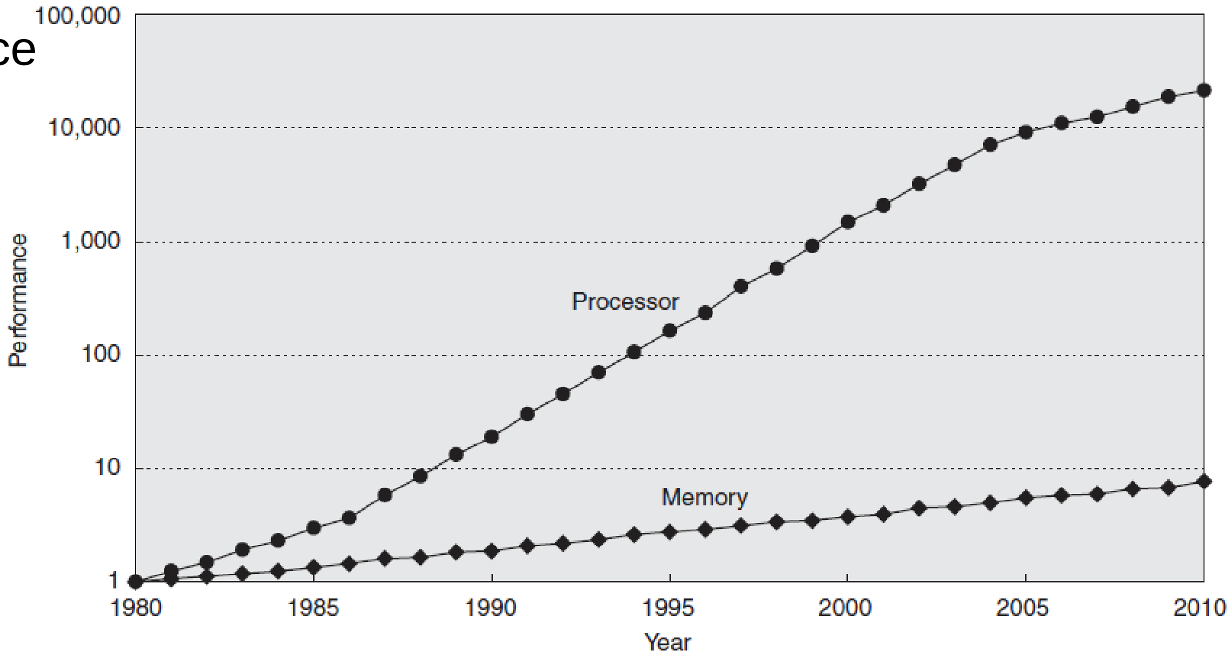
Concepts of internal memory

- Word
  - The “natural” unit of organization of memory
  - Typically equal to the number of bits to represent an integer (size of processor registers)
- Addressable unit
  - Many systems allow addressing at the byte level but there are systems where addressable unit is the word
  - Length of address in bits and number of addressable units are related. If address has A bits then number N of addressable units is $2^A = N$
- Unit of transfer
  - For main memory this is the number of bits read out of or written into memory at a time
  - Need not to be equal with word or addressable unit length
  - For external memory data is often transferred in much larger units than a word referred to as blocks

Memory system

- Data and instructions are stored in memory
- Computer pioneers predicted that programmers would want unlimited amounts of fast memory
  - Memory capacity of computers has increased rapidly (from 64 kilobytes of original IBM PC to gigabytes of modern desktop computers)

- Processor performance has grown faster than memory performance

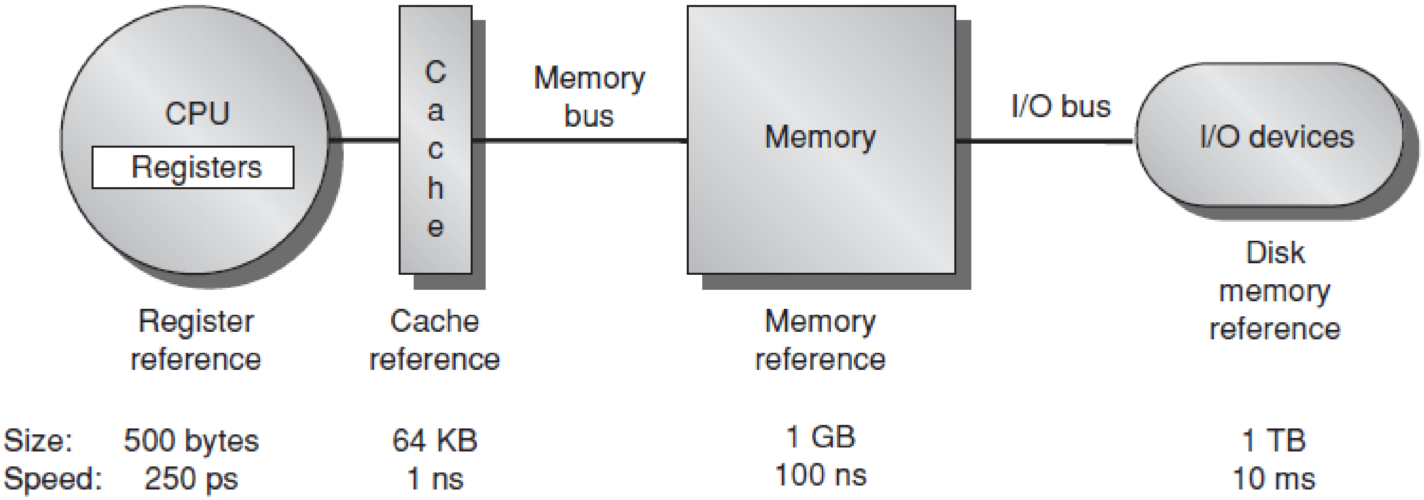


Memory hierarchy

- An economical solution to desire of having a large and fast memory is a memory hierarchy
  - Since fast memory is expensive, memory is organized into several levels where a higher level is smaller, faster and more expensive than the next lower level
- The goal is to provide a memory system with cost per byte almost as low as the cheapest level of memory and speed almost as fast as the fastest level
  - The importance of memory hierarchy has increased with advances in computer performance
- Each level maps addresses from a slower, larger memory to a smaller but faster memory higher in the hierarchy

Memory hierarchy

- Different levels in hierarchy are implemented using different technologies
  - Hard disks
    - Magnetic storage
  - Main memory
    - DRAM (dynamic RAM) – inexpensive, slow
  - Cache memory
    - SRAM (static RAM) – expensive, fast

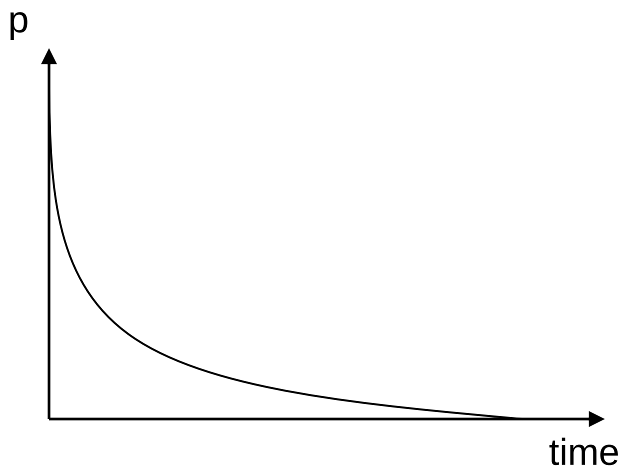


Principle of locality

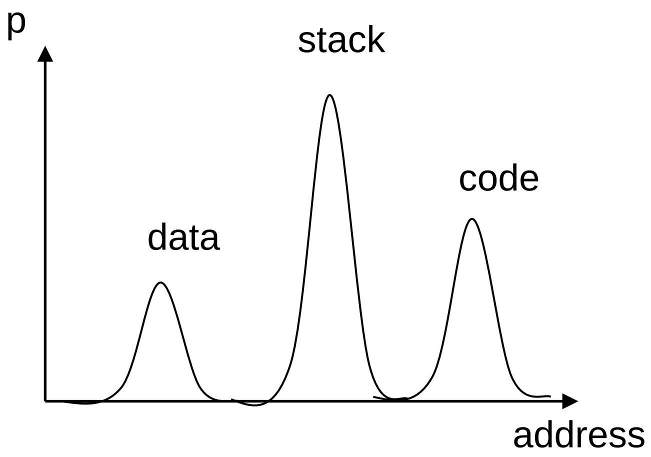
- Programs do not access memory uniformly
- *Principle of locality* states that programs access relatively small portion of their address space at any instant of time
- There are two types of locality:
  - *Temporal locality* (locality in time) – if an item is referenced it will tend to be referenced again soon
  - *Spatial locality* (locality in space) – if an item is referenced, items whose addresses are close by will tend to be referenced soon

Principle of locality

- *Temporal locality* (locality in time)
  - if an item is referenced it will tend to be referenced again soon

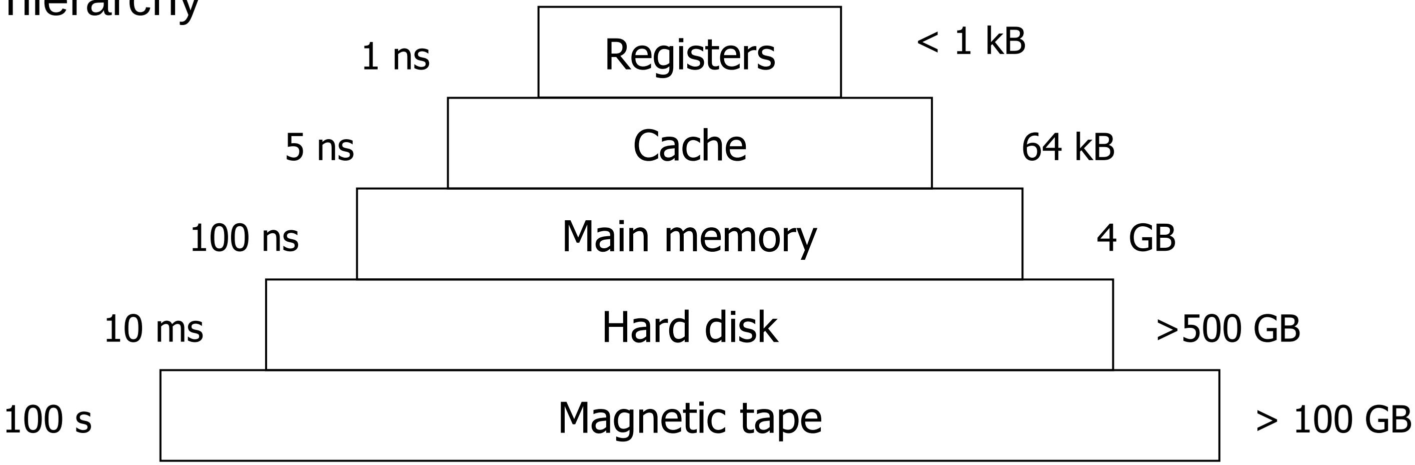


- *Spatial locality* (locality in space)
  - if an item is referenced, items whose addresses are close by will tend to be referenced soon



Memory hierarchy

- Memory hierarchies takes advantage of temporal locality
  - Keep recently accessed data closer to the processor
    - Programs and data that you run or need daily are kept on the hard disk
    - Program that is running now is in the main memory
    - The most important parts of the program are in cache memory
    - Data that instructions are processing is in the registers
- Memory hierarchies take advantage of spatial locality
  - Move blocks of multiple contiguous words of memory to upper levels of hierarchy



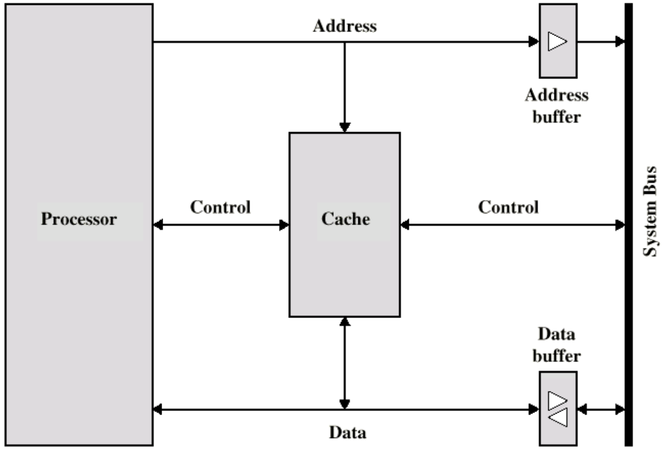
1 ns	Registers	< 1 kB
5 ns	Cache	64 kB
100 ns	Main memory	4 GB
10 ms	Hard disk	>500 GB
100 s	Magnetic tape	> 100 GB

Cache memory

- The level(s) of memory between CPU and main memory are called cache memories
    - Cache memories are not visible to programmer – they are temporary storage of recently used data
- Principle of operation
    - CPU makes a memory access
    - Cache controller checks if data from the requested address is already in the cache memory
        - If data is in the cache it is delivered to CPU (fast)
        - If data was not in the cache then access main memory and fetch a block of memory containing requested address in to cache and deliver data to CPU (slow)

Cache memory

- Cache memory is connected to processor's address, data and control buses
- Address and data buffers are connected to system bus which connects main memory to the processor
- **Cache hit** – data from requested address is in the cache, buffers are inactive and data is transferred only between cache and processor
  - System bus is not used on cache hits
- **Cache miss** – data from requested address is not in the cache, buffers are activated and memory access goes to the system bus
- To take advantage of *spatial locality* cache copies a **block** of data from main memory when a miss occurs
- System bus access on cache miss introduces additional delay to memory access time. This additional delay is called **miss penalty**
- Cache miss causes stall cycles to pipeline thus degrading performance



Cache performance

- Accurate performance prediction requires very detailed simulations of the processor and memory system
- We use a simplified model of the memory system for our performance estimates
  - CPU time can be divided to clock cycles spent executing the program and clock cycles spent waiting for the memory system (memory stall cycles)
  - Memory stall cycles come primarily from cache misses
  - CPU clock cycles include the time to handle a cache hit

$$
\begin{aligned}
Memory\ stall\ cycles &= Number\ of\ \ misses \times Miss\ \ penalty \\
&= IC \times \frac{Misses}{Instruction} \times Miss\ \ penalty \\
&= IC \times \frac{Memory\ accesses}{Instruction} \times Miss\ rate \times Miss\ \ penalty
\end{aligned}
$$

Cache performance

- Average memory access time can be calculated as

$$\begin{aligned} AMAT &= (T_{hit} \times P_{hit}) + (T_{miss} \times P_{miss}) \\ &= T_{hit} + P_{miss} \times T_{miss\ penalty} \end{aligned}$$

- In the above the miss penalty is the difference between T_{hit} and T_{miss}
- Average memory access time is improved by reducing miss rate, hit time or miss penalty
    - Miss penalty depends on the main memory technology and block size
    - Hit time increases when cache size increases

Miss categories

- *Compulsory* – the first access to a block cannot be in the cache (even if you had infinite cache)
- *Capacity* – if the cache is not large enough to contain all the blocks needed during the execution then some of the blocks need to be discarded and later retrieved
- *Conflict* – if the block placement strategy is not fully associative a block maybe discarded and later retrieved if conflicting blocks map to its set

Cache optimizations

- *Larger block size to reduce miss rate* – Take advantage of spatial locality and increase the block size. Larger block size increases miss penalty because it takes longer to fetch larger amount of data
- *Bigger caches to reduce miss rate* – Obviously capacity misses are reduced with a larger cache. Hit time may increase.
- *Higher associativity to reduce miss rate* – Increasing associativity reduces conflict misses
- *Multilevel caches to reduce miss penalty* – First level cache can be small and fast to match high clock rate and second level can be large to capture as many accesses as possible

Multilevel cache

- When a miss occurs access cache on lower level
  - The lower level cache operates on the same principle as upper level cache (hit comes from cache, miss comes from a lower level)
  - The lowest cache level accesses main memory on a miss
- Access time can be calculated by applying our AMAT formula twice
  - If L1 and L2 refer to first and second level caches respectively, we can write:

$$AMAT =$$

$$Hit\ time_{L1} + Miss\ rate_{L1} \times (Hit\ time_{L2} + Miss\ rate_{L2} \times Miss\ \ penalty_{L2})$$

Where can a block be placed in cache?

- Because main memory is larger than cache memory we need a mapping between main memory addresses and cache addresses
- Data is transferred between cache and main memory in blocks
  - Blocks are multiples of unit of transfer and typically powers of two (1, 2, 4, 8, 16 etc.)
  - Inside the cache the blocks are stored on **cache lines**
    - Cache lines contain additional information that is needed for management
- There are three categories of cache organization. The categories are called *mapping functions*

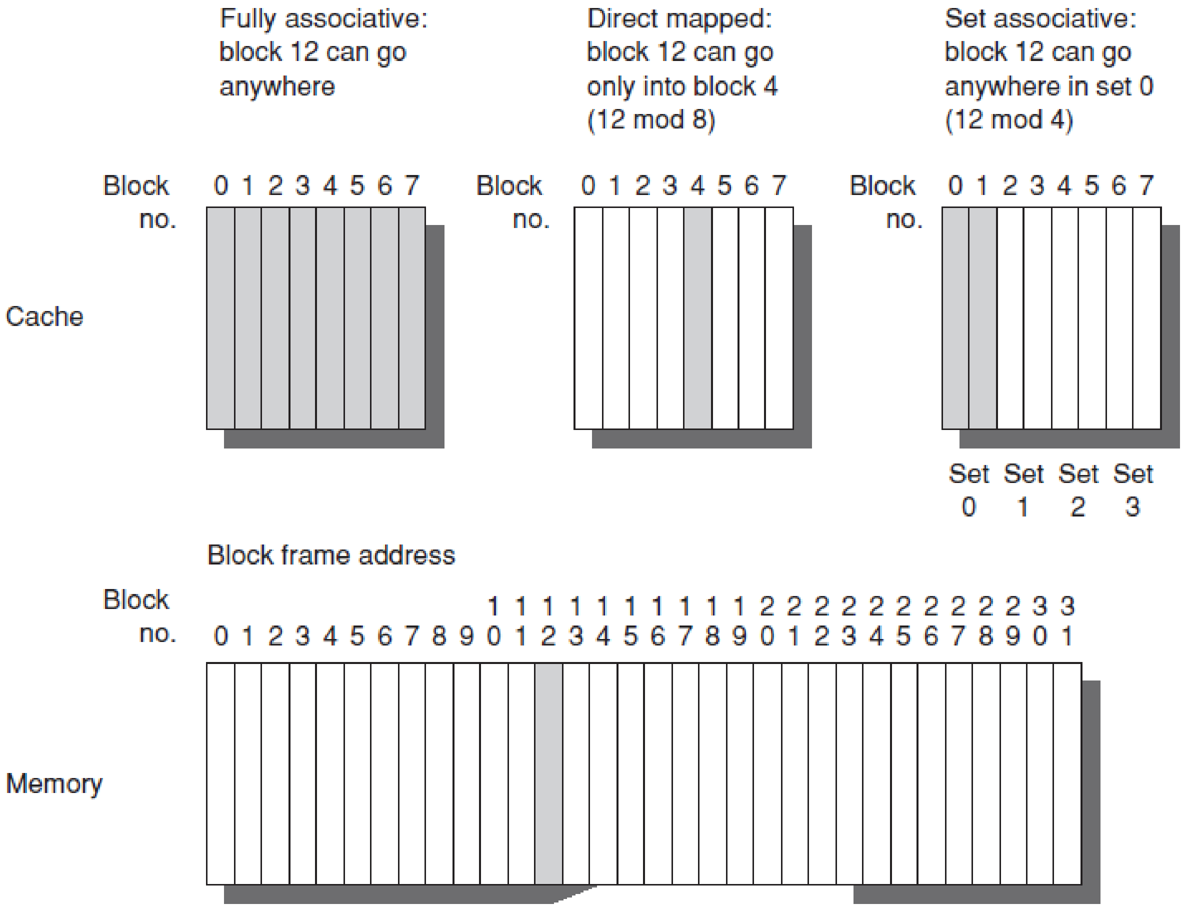
Mapping function

- If each block has only one place where it can appear in the cache, the cache is said to be **direct mapped**. The mapping is usually:

  (*Block address*) MOD (*Number of blocks in cache*)
- If a block can be placed anywhere in the cache, the cache is said to be **fully associative**
- If a block can be placed in a restricted set of places in the cache, the cache is **set associative**.
  - A **set** is a group of blocks in the cache
  - A block is first mapped onto a set and then block can be placed anywhere within that set
  - The set is usually chosen by bit selection

    (*Block address*) MOD (*Number of sets in cache*)
  - If there are *n* blocks in a set, the cache placement is called ***n*-way set associative**

Mapping functions



- This example cache has eight blocks and memory has 32 blocks
- The set associative organizations of cache is two-way set associative (two blocks per set, four sets)
- Real caches contain thousands of blocks and real memories contain millions of blocks

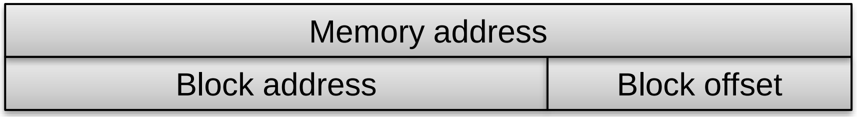
How is a block found if it is in the cache?

- Mapping function tells where a block is placed in the cache
  - Main memory is larger than cache so multiple blocks of main memory map onto same cache line
  - We need to store additional information which tells the address of the block in the main memory – we need to store a **tag** which contains this information on each line
  - There must be a way to tell if the data on the cache line is valid – we need to store **valid bit** on each line

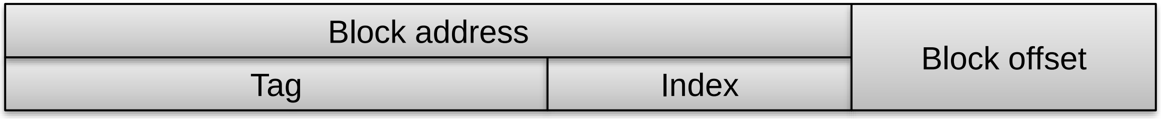
Address mapping

- Relationship of processor address and cache is simple
  - Address is divided into two fields: block address and block offset

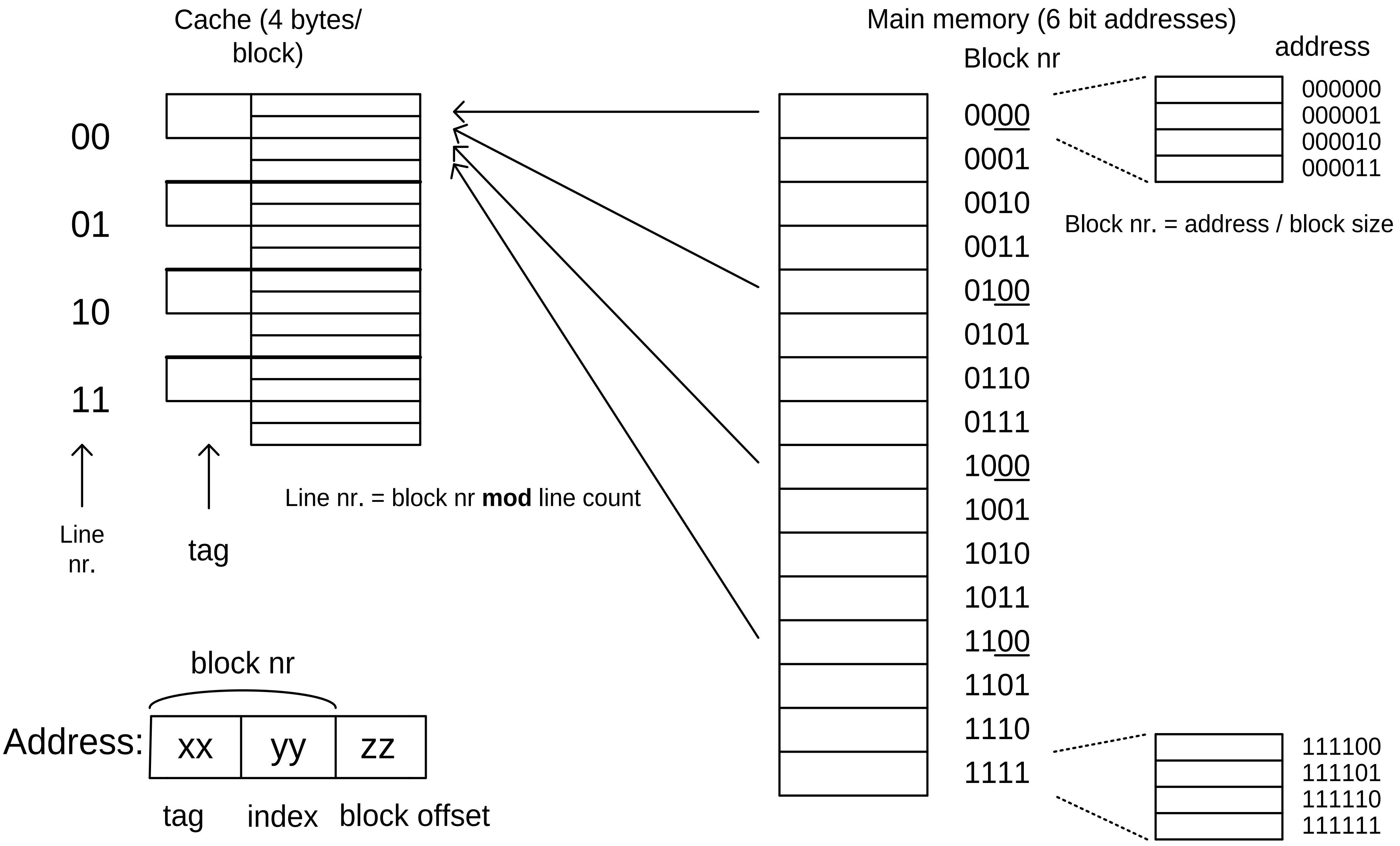
$$\mathit{Block\ address} = \mathit{Memory\ address} / \mathit{Block\ size}$$



- Direct mapping and set associative mapping divide block address to index and tag fields
  - Index field tells which line or set the block is mapped onto
  - Tag field is needed to determine whether we have a hit or miss



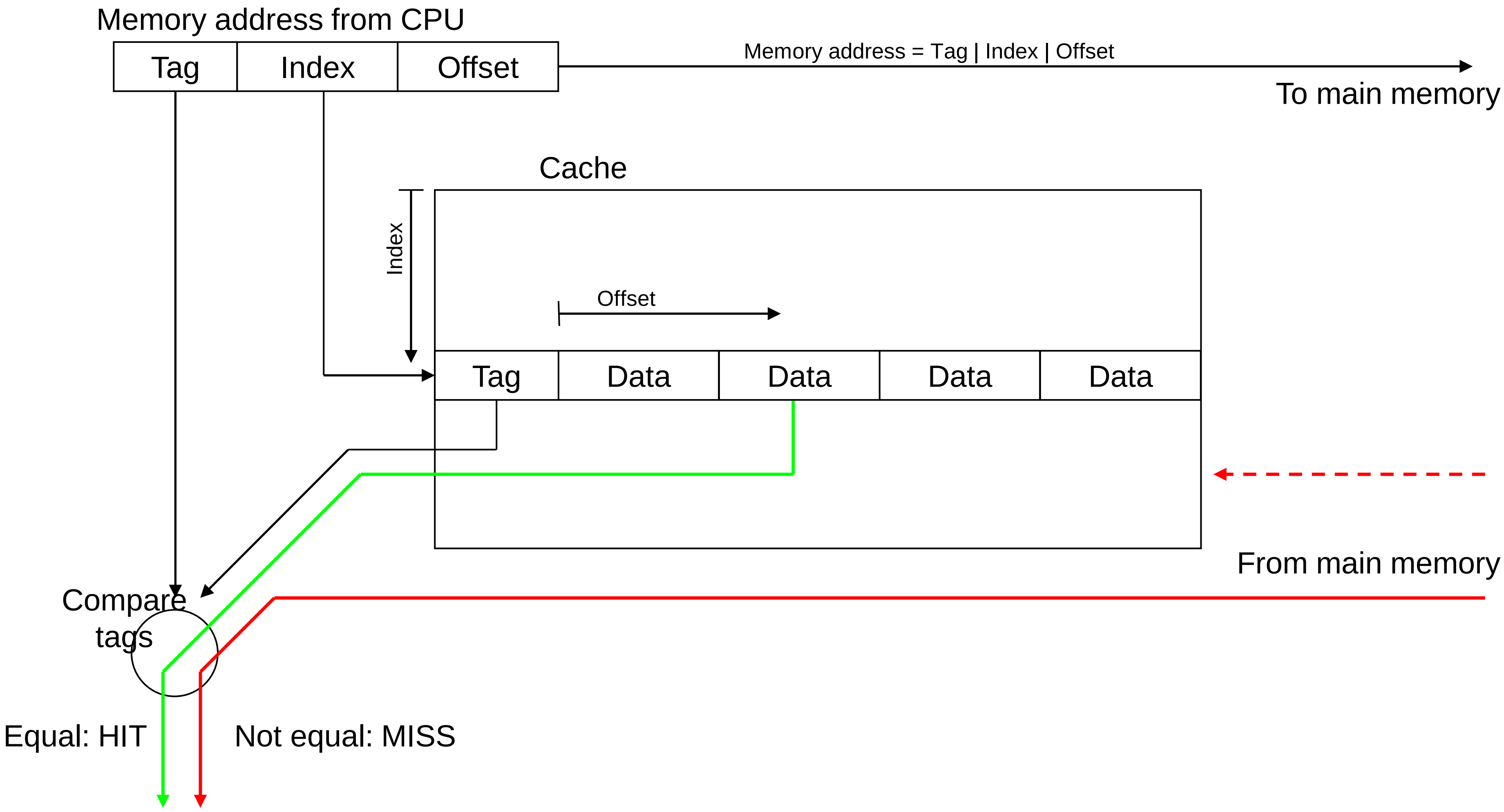
Direct mapping example



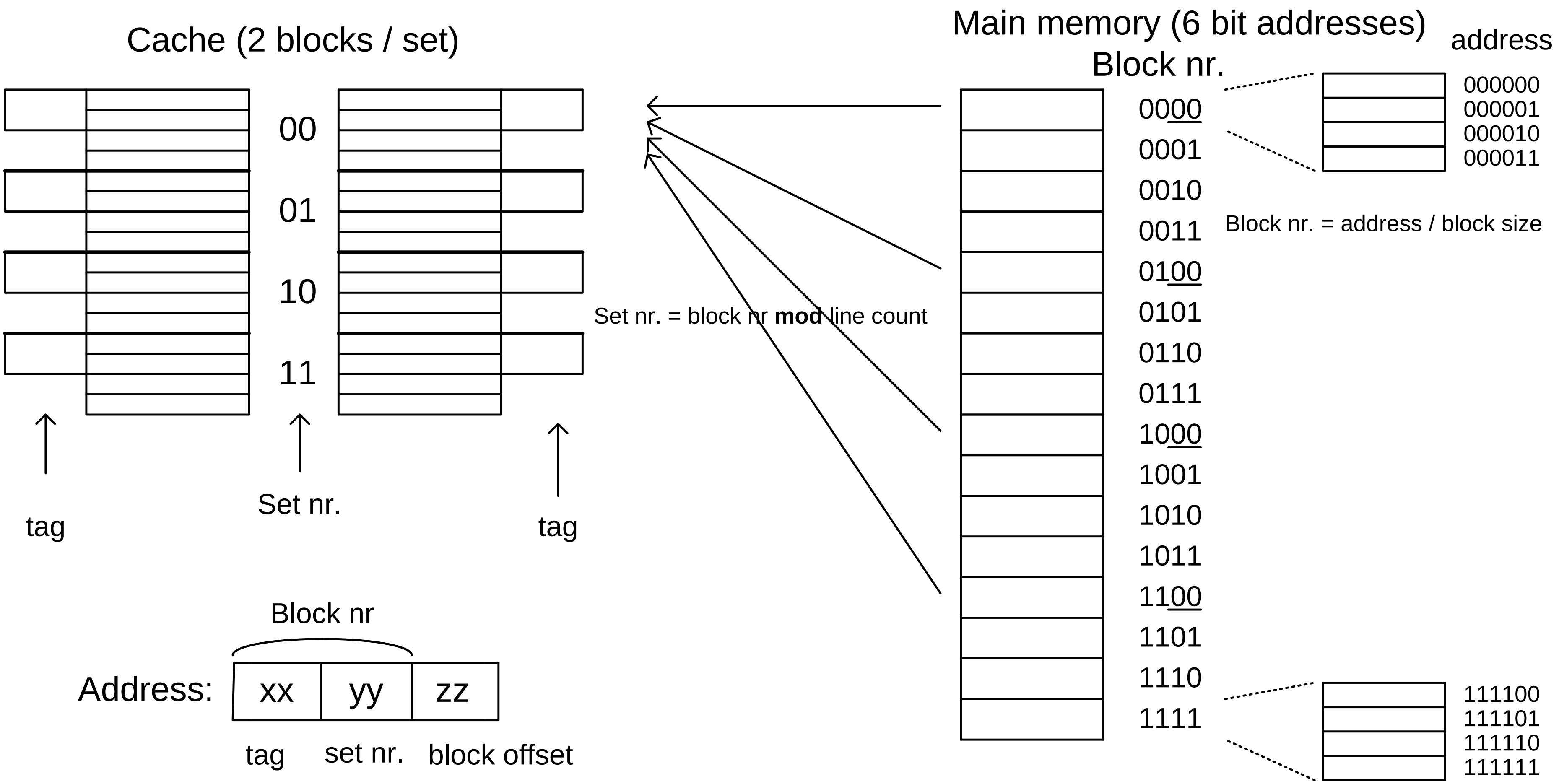
Block nr. = address / block size

Line nr. = block nr **mod** line count

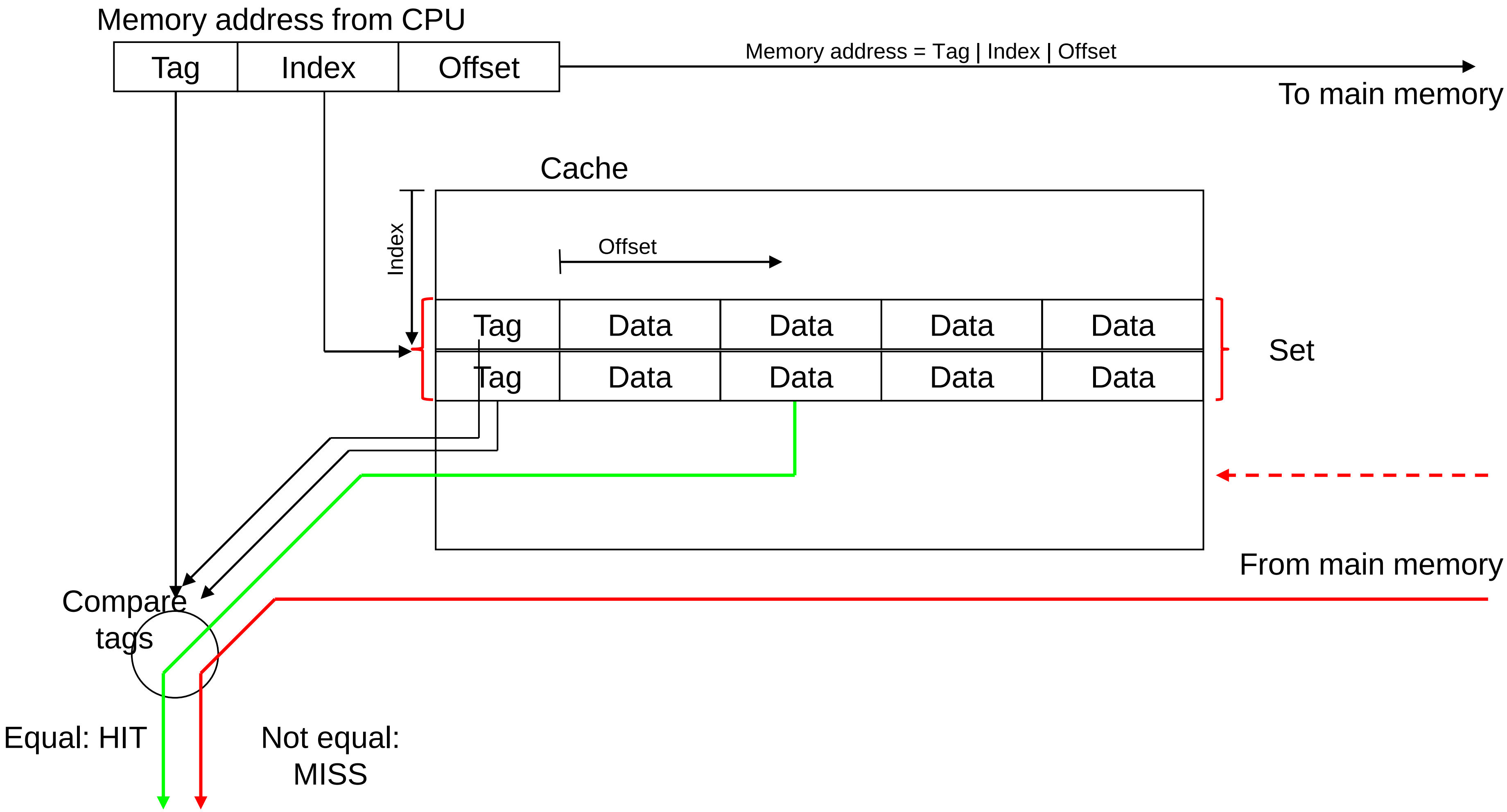
Direct mapping



Set associative mapping



Set associative mapping



Address mapping

- With all mapping functions we need the following information:
  - Number of address bits
  - Block size (always a power of 2)
  - Number of cache lines (always a power of 2)
- Then the memory address bits are divided into three fields:
  - Offset
    - Location of the data within the block
  - Index
    - To which line/set the data goes it the cache
  - Tag
    - When you combine tag and index you get the block number

Formulas 1

- Start with the address of data

$$Block\ nr = \frac{Address}{Block\ size}$$

- When you divide by a power of two you shift the dividend to the right
- Since the block size is always a power of two the calculation does not require any gates
  - Most significant bits are wired to block nr
  - Least significant bits are wired to offset
  - For example if block size is 8 (2^3) then three lowest bits go to offset and all the rest go to block nr

Formulas 2

- We need block number and the number of sets to calculate which set the block is mapped into
- Number of sets is calculated as follows:

$$Nr\ of\ \mathrm{sets} = \frac{Cache\ size}{Block\ size\ \times Rate\ of\ Associativity}$$

- If you have direct mapping the rate of associativity is 1 (there is space for just one block on each set)

Formulas 3

- Set number that block is mapped into is calculated as follows

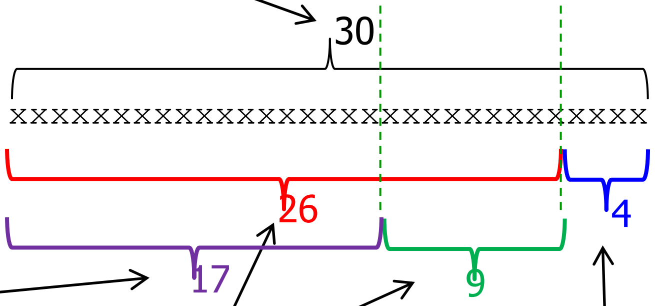
$$Set\ nr = Block\ nr\ \mathbf{mod}\ Nr\ of\ sets$$

- When you divide by a power of two to the remainder is the bits the drop out when you shift right so you take the as many bits from the right as the divisor exponent indicates

Example

- Maximum memory size 1 GB (= 2^{30} → 30 address bits)
- Block size 16 bytes (2^{4})
- Cache size 16 kB, 2-way set associative

Sets = 16 kB / (2 × 16B) = 512 = 2^{9}



Tag = Rest of the bits from block nr

= Block nr / Sets (alternative way of calculating tag)

Index = Block nr **mod** Sets

= Block nr **mod** 2^{9}

Block nr = Address / block size (2^{4})

Hit or miss?

- Only the tag needs to be compared
  - The offset should not be used in comparison since the entire block is present or not, hence all block offsets result in a match
  - Checking the index is redundant since it was used to select set (or line) to be checked
- Fully associative cache needs to compare all tags in the cache against the tag from memory address since a block can go anywhere in the set

The real world

- Direct mapping causes conflict misses
  - If two blocks that are needed at same time are mapped to same cache line there will be a miss each time different block is accessed
- Fully associative mapping is too expensive to implement
  - We would need to compare all tags in cache memory to determine hit or miss
- Real world caches are set associative
  - 8-way set associative cache is (almost) as good as fully associative cache in terms of conflict misses

Which block should be replaced on a cache miss?

- When a miss occurs the cache controller must select a block to be replaced with the requested data
  - Direct mapping is simple. You have no choice – only one line is checked and only that block can be replaced
  - Fully or set associative caches have multiple blocks to choose from
- Three primary strategies for selecting the block to replace
  - Random
  - Least Recently Used (LRU)
  - First in, first out (FIFO)

Replacement strategies

- Random
  - To spread allocation uniformly the blocks to remove are randomly selected within the set
  - Simple to build in hardware
- Least Recently Used
  - Relies on temporal locality: if recently used blocks are likely to be used again, then good candidate for disposal is the least recently used block.
  - Needs book keeping of references
  - As the number of blocks (per set) grows LRU becomes expensive to implement and is frequently only approximated
- First in, first out
  - Because LRU can be complicated to calculate, this determines the oldest block rather than LRU
  - Does account for locality as well as LRU → not commonly used

LRU approximation

- LRU is simple to implement if set size is two
  - Only one bit is required for book keeping
  - Bit indicates which of the two is oldest
- With 8 block set three bits per block are required to keep track of the references
  - This no longer trivial to calculate
  - Often approximated by half interval search (binary search)
    - Divide set in two halves and select the half that was LRU
    - Then divide that set in two halves etc. until you have one block to replace

What happens on a write?

- Reads dominate processor cache accesses
  - All instruction accesses are reads and most instructions don't write memory
  - Making the common case fast means optimizing for read
  - Amdahl's law reminds us that we can not neglect the speed of writes for high performance system
- Reads are easy to optimize
  - We can start block read from main memory as soon as we have block address – if we have a hit we can ignore the read result from main memory

Write policies

- Writes need to update both cache and main memory (or lower level memory)
- Two write policies:
  - Write through – Information is written to both the block in the cache and the block in lower level memory
    - Slows down writes (lower level write time determines write time)
  - Write back – Information is written only to the block in cache. Modified block is written to lower level only when it is replaced.
    - Must ensure coherency for multiprocessors and IO
      - Some processors leave part of the coherency assurance for user/OS