# Memory Hierarchy and Cache Design

# Memory Hierarchy

- Physical memory (or computer storage) is organized in levels based on ***response time***.
- There are four major storage levels (increasing response time):
  - Internal: Processor registers and cache
  - Main – the system RAM and controller cards
  - On-line mass storage – Secondary storage (e.g. hard disk).
  - Off-line storage – Removable storage. E.g. USB flash drive (aka USB key) .
- The goal of a memory hierarchy is to keep all of the code & data needed in present and near future, close to the processor.
- Ideally, this information should be reachable in a single cycle.
- This may requires putting the cache on the processor chip.

# The Need for Caches

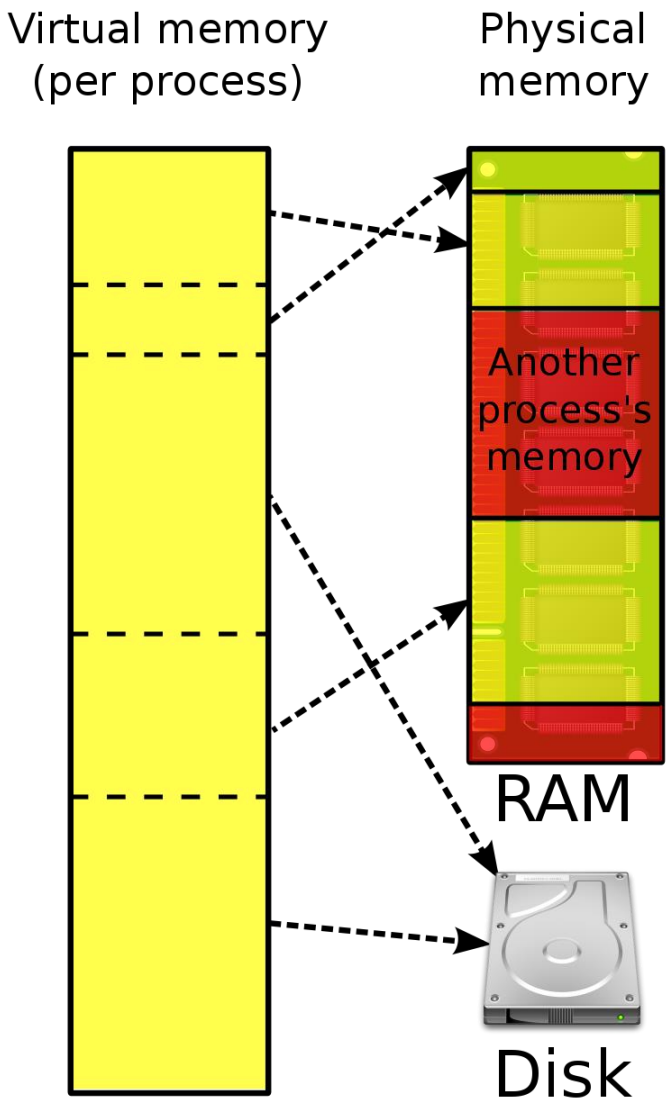
- There has been a steady increase in the mismatch between processor speed and memory latency.
- There is a need for a ***memory buffer*** between the high-performance CPUs and main memory.
- Caches are memory buffers introduced for this purpose.
- A cache is SRAM (Static Random Access Memory).
- An on-chip cache cannot be too large, because the time to access the cache is proportional to its size.
- Modern computers have multilevel caches where the cache size increases as we go further away from the processor chip.
- As the gap between processor and memory performance grew, multilevel caches became the norm.

# Caches & Memory Access

- Caches are most appropriate when the memory-accessing pattern of a running program is not random (i.e. regular).
- The most important kind of regularity, commonly referred to as ***temporal locality***, is present in a program when the code and data used in the recent past are highly likely to be reused in the near future.
- A second-order regularity, commonly referred to as ***spatial locality***, is present in a program when the code and data currently in use are highly likely to be followed by the use, in the near future, of code and data at nearby memory locations (i.e., at nearby virtual memory addresses).

# What is Virtual Memory?

- Virtual Memory or Virtual storage ***is not physical memory***. It ***is a memory management technique.***
- Provides an ***idealized abstraction*** of the storage resources that are actually available on a given machine.
- Creates the illusion to users of a very large (main) memory.
- ***We will not be covering Virtual memory in this course.***



# General Cache Properties

- Caches are much smaller than main memory so, at any given moment, they contain a small fraction of the executing program's code and data.
- In reality, most programs have large working sets, and there is no cache large enough to contain the entire working set.
- Thus, data items are brought in the cache but get flushed to make room for new data items. This repeats over the course of program execution.

# Cache Mechanisms

- When the processor issues a memory reference request for data, or the fetch-execute cycle initiates a memory access for a program instruction, a ***cache lookup*** is performed to get the requested data item or code segment.
- If the requested item (data or code) is present in the cache, it is a ***cache hit***. If not, it is a ***cache miss***.
- A cache hit leads to the rapid transfer of information from/to the cache to/from the pipeline (data/code load or data store)
- Information flows in both directions between cache and pipeline just as information flows in both directions between memory and processor.
- A cache miss initiates a recursive query through the remaining levels of the cache hierarchy and possibly the memory.

# Single-Level D-Cache

- For simplicity, we will consider the D-Cache or Data-cache. It hold program data only (no program instructions).
- Five basic cache-design questions will be considered:
  - **The Lookup**: How do we determine if a copy of a given data item is in the cache?
  - **The copy-decision**: When a memory reference (load or store) leads to a cache miss, do we automatically make a copy?
  - **The cache-line-size:** How many bytes do we copy?
  - **The placement:** Where do we put the copy?
  - **The Replacement:** What if the cache container where the new data item is supposed to be placed is ***full.***
  - **The Write**: How do we write updated data back to main memory?
- A look at cache architecture will answer these questions.

# Cache Architecture

- The smallest building block of the cache is called a frame or a block. The size of a frame is given in bytes.
- A cache is an array of containers. Each container is addressed by its index. The number of frames inside each container is determined by the ***cache mapping scheme***.
  - In a **Direct Mapped Cache**, a cache container is a frame. In other words, the cache is an array of frames.
  - In an **m-way set associative cache**, a cache container is a set that has m frames. In other words, the cache is an ***array of S sets*** and each set contains m frames. (m is $2^n$ and n>=1).
  - In a ***fully associative cache***, the whole cache is one large frame.
- The ***cache capacity (in bytes)*** is the product of the total number of cache frames and the size of each frame in bytes (aka cache line size).

# Data Access Mechanism
# Cache Mapping Schemes

- Recall that the purpose of cache is to speed up memory accesses by storing recently used data closer to the CPU.
- When accessing information (data or instructions), the CPU first generates a main memory address.
- If the information has already been ***copied to the cache***, its cache address will be different from the main memory address.
- But how then does the CPU use a main memory address to locate data in the cache?
- The CPU uses a specific mapping scheme that interprets (or maps) the ***main memory address into a cache location***.
- The cache mapping scheme also determines where data is placed in the cache when it is copied from main memory.

# How is Data Copied from Main Memory to Cache?

- **Recall:** The main memory address used to access data in main memory is ***also used*** to access the same data in cache.
- Main memory and cache are both divided into blocks (frames) of the same size (the block size varies on different machines).
- When the CPU generates a ***memory address*** to access data, the cache is searched first to see if the required data (e.g. byte or word) exists there.
- When the requested data is not found in cache, the entire main memory block in which the data resides is loaded into cache.
- **What**? Does this sound like complete wasting of cache space.
- **This Scheme works!** Recall ***principles of locality***, a data reference is highly likely to be followed by other nearby data references.

# Main Memory Address Mapping

**How can the main memory address be used to access the cache**?

**Address Mapping**:

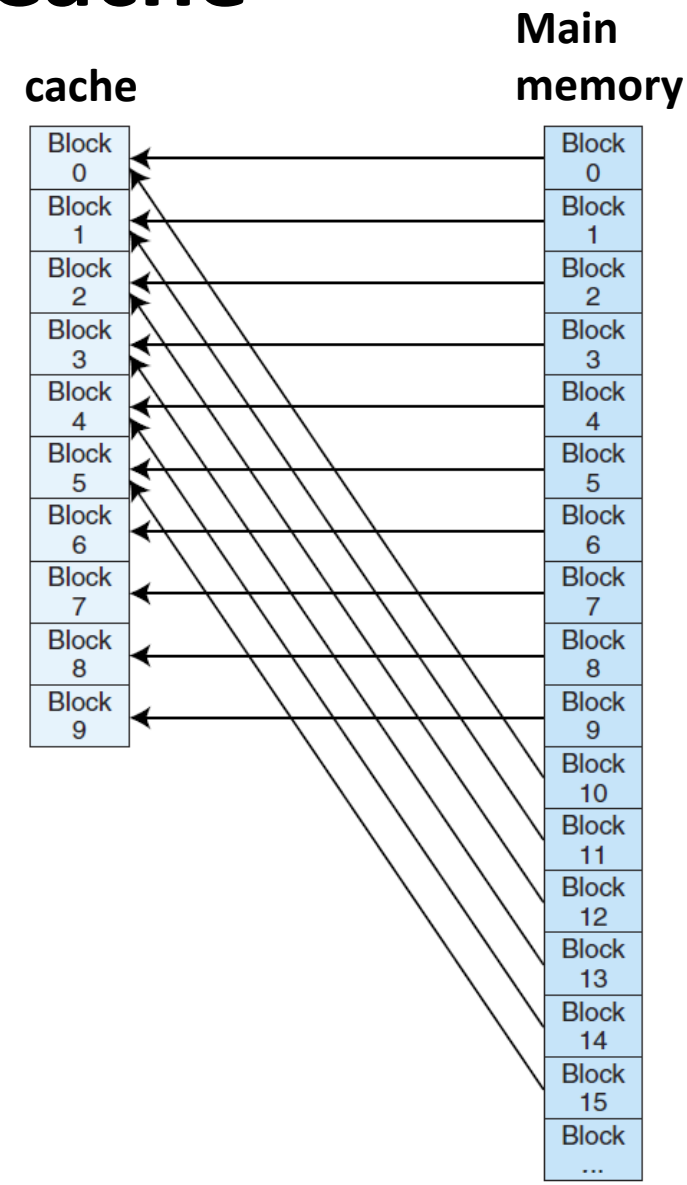
- The bits in the main memory address are separated into distinct groups, called ***fields***. We will have 2 or 3 fields depending on the cache-mapping scheme. The interpretation of the 3 fields also depends on the mapping scheme.
  - **Offset:** indicates the start position (byte offset) of the data item within the cache frame.
  - **Index:** set or frame where the data is located in the cache.
  - **Tag:** serves as a content identifier since many memory locations are mapped to the same cache frame.
  - Index & Tag fields together aka cache line number or memory block address.

# Direct-Mapped Cache

- A cache structure in which each memory location is mapped to exactly one location in the cache.
- The typical mapping between addresses and cache locations for a direct-mapped cache is simple.
- In this example, the ***cache has 10 frames*** (indexed 0 to 9). Main memory blocks 0 to 9, 10 to 19, .. etc. get mapped to cache frames 0 to 9.
- # frames = number of frames in cache

Frame index in cache =

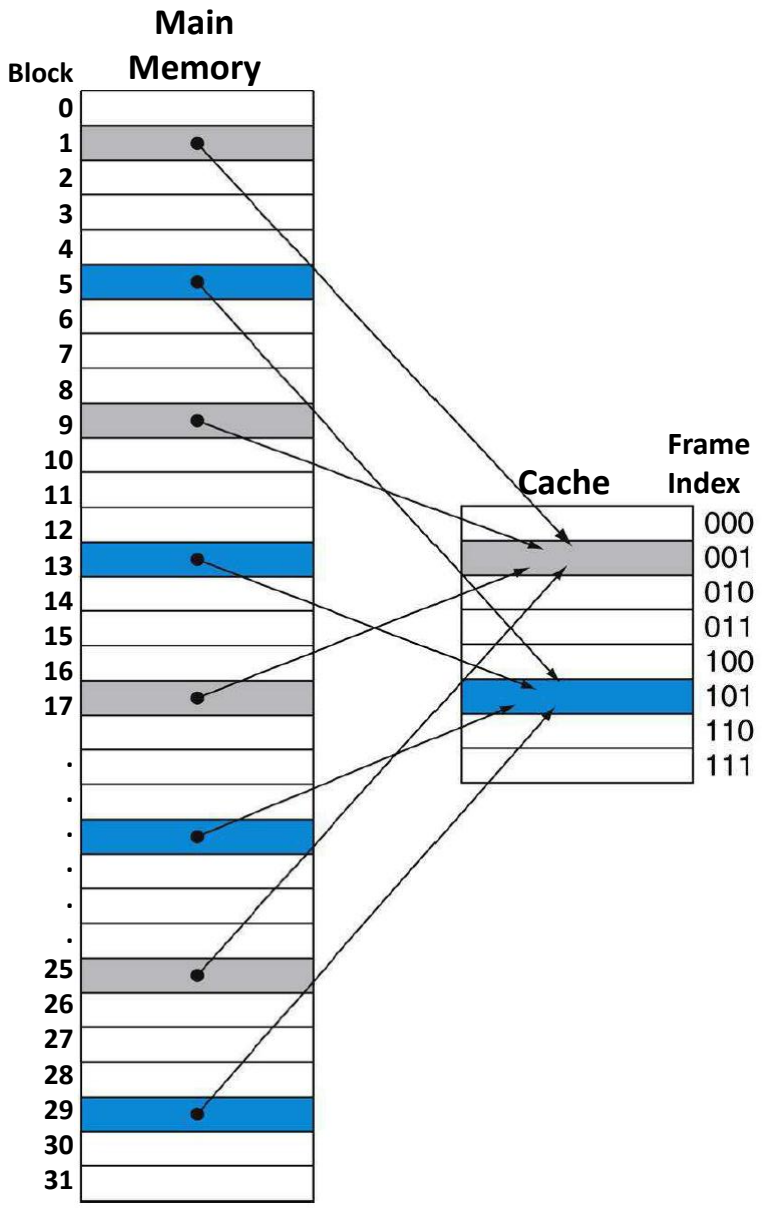
memory block number mod # frames




**Direct Mapping of Main Memory Blocks To cache frames**

# Direct-Mapped Cache - Example

- This example shows the mapping of main memory address to cache address.
- Byte-addressable memory: $512=2^9$ bytes. So memory address is 9 bits.
- Block size (frame size) = 16 = $2^4$bytes
- memory has 512/16 = 32 blocks
- Cache has 8 frames
- Cache capacity = 8 X 16 = 128 bytes
- Blocks 1,9,17,25 in memory map to frame 001 in cache and blocks 5,13,21,29 in memory map to frame 101 in cache (Block ID mod 8).



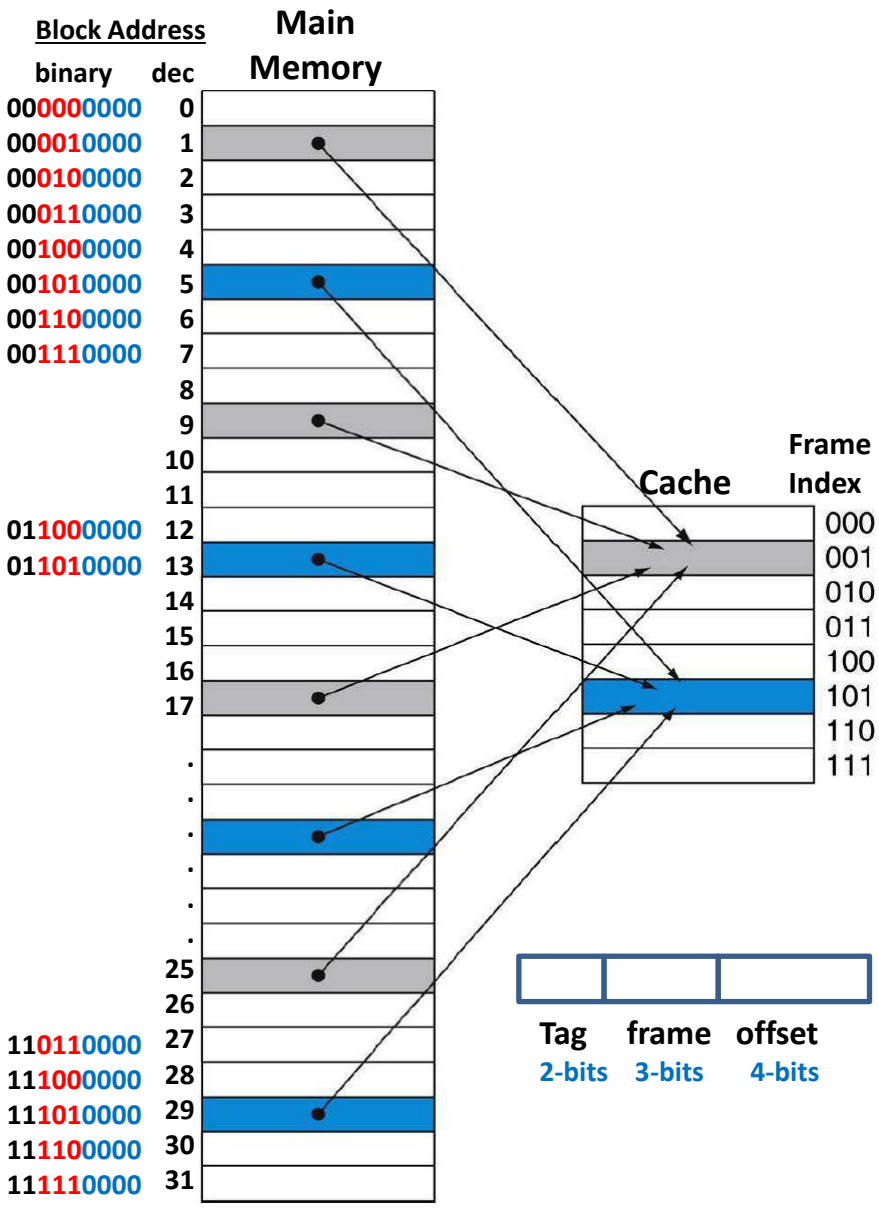
# Direct-Mapped Cache - Example

So how is the 9-bit memory address mapped to access the cache?

- The address is divided in 3 fields:
  - Offset: determined based on block size.

    Block size = $2^4$ bytes so we need 4-bit offset (in blue)
  - Index: Frame index is determined based on number of frames in the cache. There are $2^3$ frames so we need 3 bits (in red)
  - Tag: Takes the remaining 2bits. Since 4 different memory blocks map to the same cache frame, the tag acts as an identifier to indicate which one of the 4 blocks is actually in the cache.

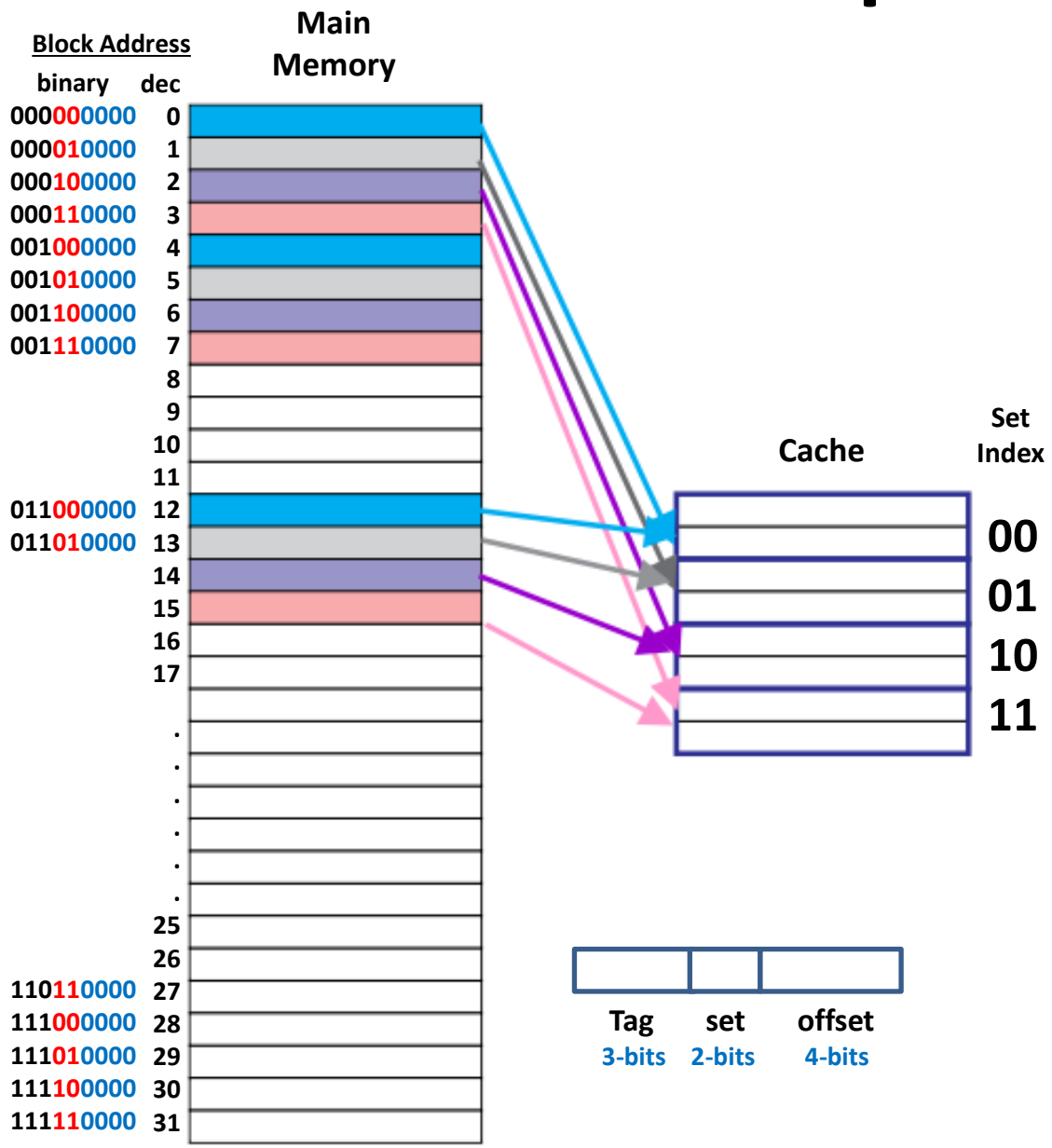
  **Example:** Cache frame 101, blocks 5,13, 21 and 29 map to the same frame. The 2-bit tag would be set to 00, 01, 10, or 11.



# 2-Way Set-Associative Cache- Example

Consider the same memory in previous example ($2^9$ bytes, 9-bit address). Remember, memory does not change but cache design changes. The cache still has the same number of frames organized as 2-way associative.

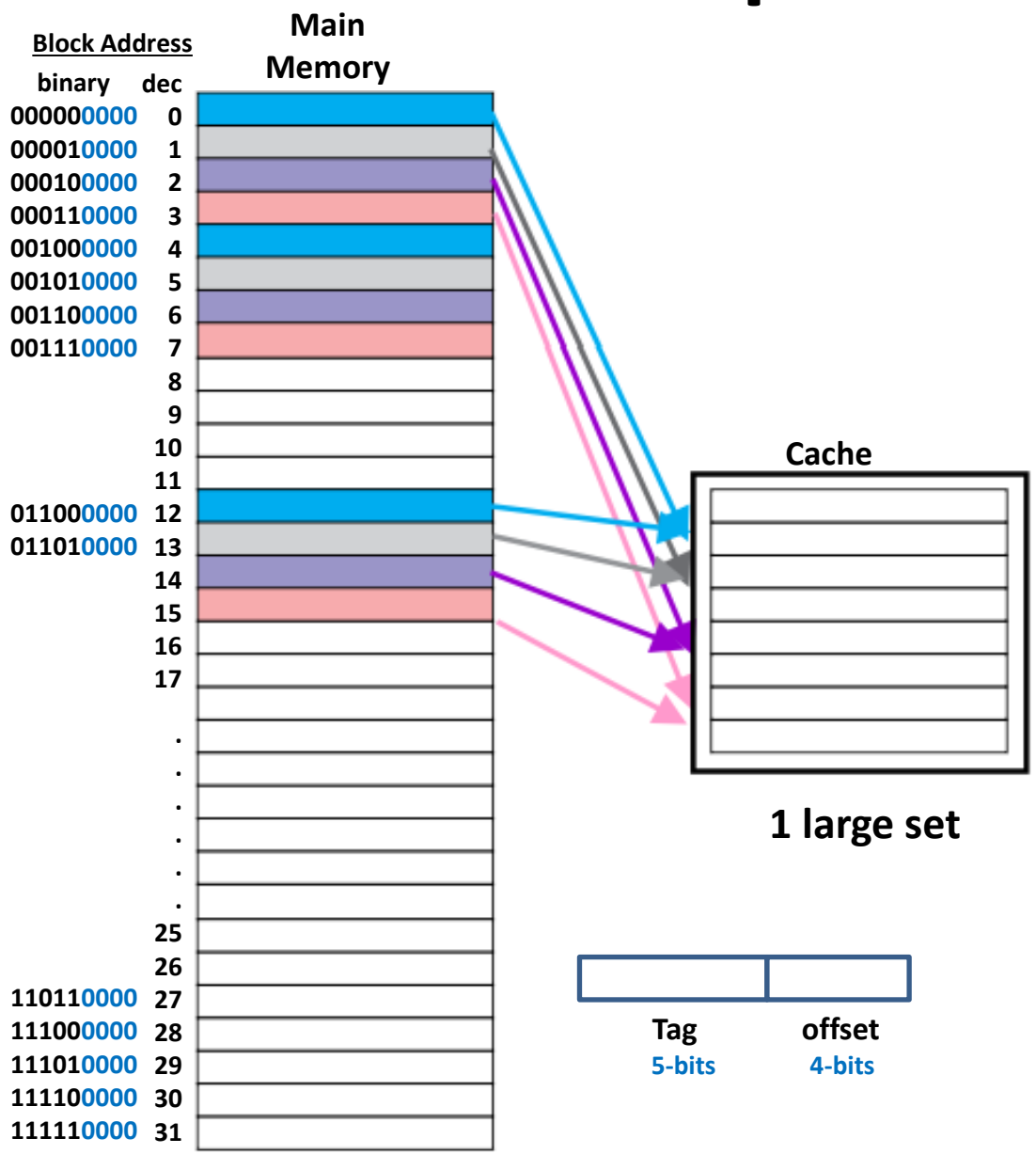
- Memory address fields:
  - **Offset**: 4-bits (in blue) as before.
  - **Set Index**: There are $2^2$ sets so we need 2 bits (in red)
  - **Tag**: Takes the remaining 3 bits. We have 32 memory blocks mapping in 4 sets. ***Every 8 blocks map to the same set.*** So we need a ***3-bit identifier***.
  - Take cache set 01 as an example, blocks 1,5,9,13,.. (***identify all 8 of them***) map to same cache set. The 3-bit tag would be set to 000, 001, 010, 011, 100,101, 110, 111 respectively.



# Fully-Associative Cache- Example

Consider the same memory in previous 2 examples ($2^9$ bytes, 9-bit address). The cache still has the same number of frames organized as a fully associative. Cache. The cache is considered as 1 large set with 8 frames so memory blocks can map to any frame in the set.

- Memory address fields:
  - **Offset**: 4-bits (in blue) as before.
  - **No more Index field required**.
  - **Tag**: Takes the remaining 5 bits. We have ***32 memory blocks mapping to 1 set.*** So we need a ***5-bit identifier***.



# Cache Mapping Schemes Summary

Lets summarize what we have seen so far for the same memory as it has been supported by the 3 different cache mapping schemes.

Memory size = $2^9$ bytes, 9-bit address.

Cache capacity = 8 frames x 16 bytes each = 128 bytes

**Important Note**: The tag size increases as the number of memory blocks mapping to the same cache container (set or frame) increases.

| Mapping Scheme | offset | Index | Tag |
|---|---|---|---|
| **Direct Mapping** | 4-bit | Frame Index: 3-bit | 2-bit |
| **2-way associative** | 4-bit | Set Index: 2-bit | 3-bit |
| **4-way associative** | 4-bit | Set Index: 1-bit | 4-bit |
| **Fully Associative** | 4-bit | No Index | 5-bit |

# Tag and Cache Lookup

**We have seen** that tag size (in bits) increases as the number of memory blocks mapped to the same cache container (set or frame) increases.

**How is the tag used to fully identify a cache block**?

- When a memory block is loaded into a cache frame, the tag (determined by the mapping scheme) is stored (***tagging***) with the actual data block.
- Caches use expensive associative memory that allows comparison of many tags in a single search. ***Cache Lookup will be covered next class***.