# Lecture – 10
# Memory Hierarchy and Cache Design

## Data Access Mechanism

**Instructor**: Samia Hilal

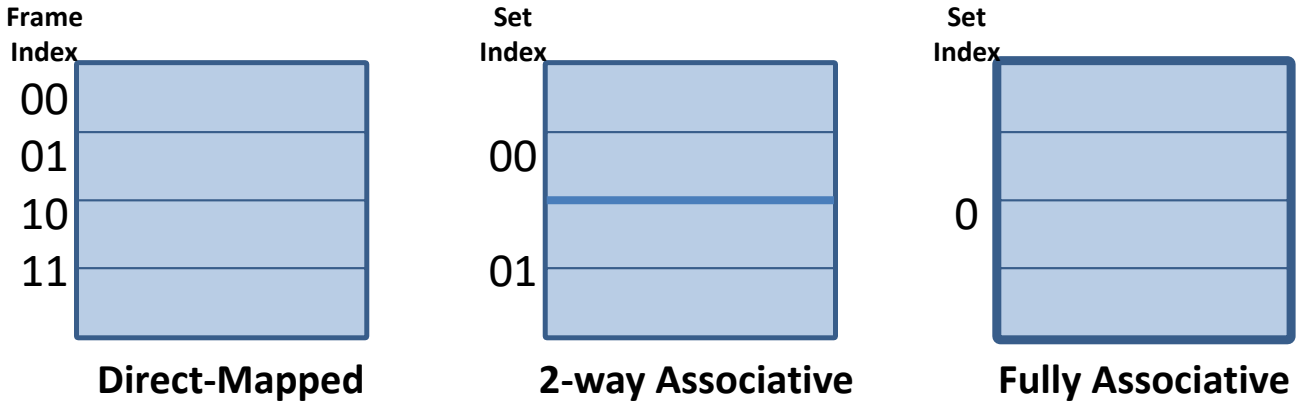
Winter 2020

# Recall - Memory Architecture

- We consider ***byte-addressable*** memory only. Each byte has a unique address. All modern computers use ***byte-addressing***.
- In other words, the smallest building block of memory is 1 byte.
- Main memory size is given in bytes and can always be represented as **$2^n$ bytes, n is the length of the memory address in bits.**
- A ***memory reference*** indicates a main memory address of a given byte (or the starting byte of a larger item). This address does not change by the cache or the cache mapping scheme.
- When a computer system ***has a cache***, the main memory is divided logically into blocks of bytes for cache accessing purposes. Memory addresses are not affected. The size of a main memory block is given in bytes and is equal to frame size.

# Recall - Cache Architecture

- A cache is an array of containers. The building block of the cache is a frame. The size of a frame is given in bytes.
- A cache is organized based on the ***cache mapping scheme***.
  - **Direct Mapped Cache**: A cache container is a frame. In other words, the cache is an array of frames.
  - **m-way set associative cache**: A cache container is a set of m frames. The cache itself is an ***array of S sets.***
  - ***fully associative cache***: The cache is ***one set of frames***.



# Recall - Cache Architecture

**Important Note**: The Cache mapping scheme ***does not*** change The total number of frames in the cache (# of frames) or frame size. It changes the layout (organization) only.

**Ca*che capacity (in bytes)* = # frames** x **size of frame in bytes**

# Data Access Mechanism

- **Recall**: Cache speeds up memory access by copying (storing) recently accessed data closer to the CPU. At any given time, The cache contains a small fraction of main memory data.
- We consider a unified cache where program instructions and data are copied to the same cache.
- When a program starts executing, none of its data or instructions are in the cache yet.
- Data access is initiated by a main memory address (or reference).
- The CPU looks for data ***in the cache first.***
- The CPU uses the ***cache mapping scheme*** to interpret the ***same main memory address into a cache location.***
- **Important Note**: Many memory addresses map to same cache location. The tag field is the unique identifier that tells us which main memory block is residing in a given cache frame.

# A Cache Hit or A cache Miss

- The CPU access to the cache has one of 2 possible outcomes:
  - **Cache Hit:** Requested data is already in the cache and can be accessed.
  - **Cache Miss:** Requested data is not in the cache and has to be copied.
- Following a ***cache miss***, the entire main memory block in which the requested data resides is loaded into cache.
- **This Scheme works!** Recall ***principles of locality***, a data access is highly likely to be followed by other nearby data references.
- **Note**: Memory does not change when data is copied to cache.
- **Cache Hit Rate** (or ratio) is given as the percentage of cache hits over the overall number of cache accesses.
- **Cache Miss Rate** = 1 - Cache Hit Rate (ratio of misses to overall)
- The remaining part of this lecture will be presented in a series of examples for each of the mapping schemes.

# Direct-Mapped Cache - Example

**Example 1**: A computer has a Byte-addressable memory, size is 512 bytes. A 128-bytes direct-mapped cache. Each frame is 16 bytes.

The system accesses the following 8 hex memory locations in this exact order: 1E8, 1EF, 0B9, 1B8, 0A6, 0BE, 1C2

Calculate the cache hit ratio for the given sequence.

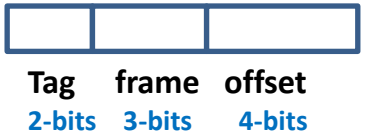
**Recall** The 3 fields in a direct-mapping scheme:

- **Offset:** indicates the start position (byte offset) of the data item within the cache frame.
- **Frame Index:** Frame ID where data is located in the cache.
- **Tag:** serves as a content identifier since many memory blocks are mapped to the same cache frame.
- Frame Index & Tag fields together (aka cache line number or memory block Id

# Direct-Mapped Cache

**Solution**: Start by *doing the following calculations* & address representation:

- Memory size is 512 (=$2^9$) bytes ➔ memory address is 9 bits.
- Block size (frame size) = 16 (=$2^4$) bytes ➔ offset is 4 bits
- Cache has 8 (=$2^3$) frames ➔ Frame Index is 3 bits
- ***Tag take the remaining 2 bits***
- and memory has 512/16 = 32 (=$2^5$) blocks. A block Id has 5 bits
- Note: Number of memory blocks/Number of cache frames = $2^5 / 2^3 = 2^2$ . Tag is 2 bits. In other words, $2^2$ memory blocks map to the same cache frame.
- Translate memory references into binary.




| Hex | 1E8 | 1EF | 0B9 | 1B8 | 0A6 | 0BE | 1C2 |
|---|---|---|---|---|---|---|---|
| bin | 111101000 | 111101111 | 010111001 | 110111000 | 010100110 | 010111110 | 111000010 |

- Note: It is important to do the hexadecimal to binary conversion in this example because address fields do not fall on hexit boundaries (not a multiple of 4). Take A4 Q2 as example, binary conversion is not required.

# Direct-Mapped Cache

Starting from an empty cache, This slide shows the contents of the cache following each memory reference.

Note: A ***conflict miss*** happens when a memory block maps to an already occupied cache frame. Earlier occupant has to be evicted.

| Tag | frame | offset |
|---|---|---|
| 2-bits | 3-bits | 4-bits |

| Hex | 1E8 | 1EF | 0B9 | 1B8 | 0A6 | 0BE | 1C2 |
|---|---|---|---|---|---|---|---|
| bin | 111101000 | 111101111 | 010111001 | 110111000 | 010100110 | 010111110 | 111000010 |

| Frame Index | data | Frame Index | data | Frame Index | data | Frame Index | data | Frame Index | data | Frame Index | data | Frame Index | data |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 000 | | 000 | | 000 | | 000 | | 000 | | 000 | | 000 | |
| 001 | | 001 | | 001 | | 001 | | 001 | | 001 | | 001 | |
| 010 | | 010 | | 010 | | 010 | | 010 | 01010 | 010 | 01010 | 010 | 01010 |
| 011 | | 011 | | 011 | 01011 | 011 | 11011 | 011 | 11011 | 011 | 01011 | 011 | 01011 |
| 100 | | 100 | | 100 | | 100 | | 100 | | 100 | | 100 | 11100 |
| 101 | | 101 | | 101 | | 101 | | 101 | | 101 | | 101 | |
| 110 | 11110 | 110 | 11110 | 110 | 11110 | 110 | 11110 | 110 | 11110 | 110 | 11110 | 110 | 11110 |
| 111 | | 111 | | 111 | | 111 | | 111 | | 111 | | 111 | |
| 1E8 → miss | | 1EF → hit | | 0B9 → miss | | 1B8 → conflict miss | | 0A6 → miss | | 0BE → conflict miss | | 1C2 → miss | |

Cache Hit Ratio = number of Hits/ Total number of memory references = 1/7

# 2-way Set Associative Cache Example

**Example 2**: The same example with cache 2-way set associative.

A computer has a Byte-addressable memory, size is 512 bytes. A 128-bytes ***2-way set associative cache***. Each frame is 16 bytes.

The system accesses the following 8 hex memory locations in this exact order: 1E8, 1EF, 0B9, 1B8, 0A6, 0BE, 1C2

Calculate the cache hit ratio for the given sequence.

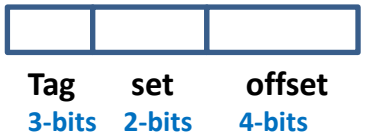
**Recall** The 3 fields in a 2-way set associative scheme:

- **Offset:** indicates the start position (byte offset) of the data item within the cache frame.
- **Set Index:** Set ID where frame containing the data is located in the cache.
- **Tag:** serves as a content identifier since many memory blocks are mapped to the same cache set.
- Set Index & Tag fields together aka cache line number or memory block Id

# 2-way Set Associative Cache

**Solution**: Start by *doing the following calculations* & address representation:

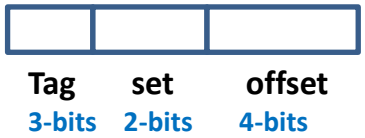
- Memory size is 512 (=$2^9$) bytes ➔ memory address is 9 bits.
- Block size (frame size) = 16 (=$2^4$) bytes ➔ offset is 4 bits
- Cache has 8 (=$2^3$) frames ➔organized as 4(=$2^2$) sets with 2 frames each. Set Index is 2 bits
- ***Tag take the remaining 3 bits***
- and memory has 512/16 = 32 (=$2^5$) blocks. A block Id has 5 bits
- Note: Number of memory blocks/Number of cache sets = $2^5$ / $2^2$ = $2^3$ Tag is 3 bits. In other words, $2^3$ memory blocks map to the same cache set.
- Translate memory references into binary




| Hex | 1E8 | 1EF | 0B9 | 1B8 | 0A6 | 0BE | 1C2 |
|---|---|---|---|---|---|---|---|
| bin | 111101000 | 111101111 | 010111001 | 110111000 | 010100110 | 010111110 | 111000010 |

# 2-way Set Associative Cache

Starting from an empty cache, This slide shows the contents of the cache following each memory reference.




| Hex | 1E8 | 1EF | 0B9 | 1B8 | 0A6 | 0BE | 1C2 |
|---|---|---|---|---|---|---|---|
| bin | 111101000 | 111101111 | 010111001 | 110111000 | 010100110 | 010111110 | 111000010 |

| Set Index | Data |
|---|---|
| 00 | |
| 01 | |
| 10 | 11110 |
| 11 | |

1E8 ➔ miss

| Set Index | Data |
|---|---|
| 00 | |
| 01 | |
| 10 | 11110 |
| 11 | |

1EF ➔ hit

| Set Index | Data |
|---|---|
| 00 | |
| 01 | |
| 10 | 11110 |
| 11 | 01011 |

0B9 ➔ miss

| Set Index | Data |
|---|---|
| 00 | |
| 01 | |
| 10 | 11110 |
| 11 | 01011<br>11011 |

1B8 ➔ miss

| Set Index | Data |
|---|---|
| 00 | |
| 01 | |
| 10 | 11110<br>01010 |
| 11 | 01011<br>11011 |

0A6 ➔ miss

| Set Index | Data |
|---|---|
| 00 | |
| 01 | |
| 10 | 11110<br>01010 |
| 11 | 01011<br>11011 |

0BE ➔ hit

| Set Index | Data |
|---|---|
| 00 | 11100 |
| 01 | |
| 10 | 11110<br>01010 |
| 11 | 01011<br>11011 |

1C2 ➔ miss

Cache Hit Ratio = number of Hits / Total number of memory references = 2/7

# Summary of Examples

- We have seen the process of ***referencing***, ***mapping*** and ***placing*** memory blocks to cache.
- We have seen the process for the same sequence of memory references over 2 types of cache: Direct-mapped and 2-way set associative.
- An associative cache reduces (but does not eliminate) conflict misses. In general, it gives a higher cache hit ratio but it depends on the nature of memory references (spatial locality). In general, ***more memory blocks will be in cache***.
- In a direct-mapped cache, ***handling a conflict miss*** is simple: Evict the current frame occupant and place the new one.
- How is a conflict miss handled in a set-associative cache?
- Set Associative caches implement a ***Replacement Policy***.

# Cache Replacement Policies
# Set-Associative Cache

- ***A Cache Replacement Policy*** is the strategy a cache uses to make room in a ***fully-occupied cache set*** for a new entry after a conflict miss. The cache must evict one of the existing entries.
- We will consider 2 of the most simple and popular policies:
  - ***FIFO - First In First Out***: A simple strategy. Needs to keep time of placement of each block. Evict the block who has been there the longest
  - ***LRU – Least Recently Used:*** A more efficient strategy but also more complex to implement. ***Discuss Why***?
- The fundamental problem with choosing an appropriate replacement policy is that it must predict which existing cache entry is least likely to be used in the future. Predicting the future is difficult, so there is no perfect method. In this course, you will be given the policy to use.

# 2-way Set Associative Cache
# FIFO Replacement Policy

Let's consider Example 2 seen earlier and see what happens if we have an additional memory reference. We note that 2 cache sets ($10_2$ and $11_2$) are fully-occupied. So any future entry to any of these sets would initiate the application of the Replacement Policy. Lets see the contents of the cache after the additional memory reference 1F0 = 111110000, mapping to cache set 11.

If we had a future memory reference (038=000111000) , it would replace the 2nd frame in the set (not shown)

| Hex | 1E8 | 1EF | 0B9 | 1B8 | 0A6 | 0BE | 1C2 |
|---|---|---|---|---|---|---|---|
| bin | 111101000 | 111101111 | 010111001 | 110111000 | 010100110 | 010111110 | 111000010 |

| Set Index | Data | Set Index | Data | Set Index | Data | Set Index | Data | Set Index | Data | Set Index | Data | Set Index | Data |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 00 | | 00 | | 00 | | 00 | | 00 | | 00 | | 00 | 11100 |
| 01 | | 01 | | 01 | | 01 | | 01 | | 01 | | 01 | |
| 10 | 11110 | 10 | 11110 | 10 | 11110 | 10 | 11110 | 10 | 11110<br>01010 | 10 | 11110<br>01010 | 10 | 11110<br>01010 |
| 11 | | 11 | | 11 | 01011 | 11 | 01011<br>11011 | 11 | 01011<br>11011 | 11 | 01011<br>11011 | 11 | 01011<br>11011 |

1E8 → miss | 1EF → hit | 0B9 → miss | 1B8 → miss | 0A6 → miss | 0BE → hit | 1C2 → miss

**From Previous Example-2**

| Set Index | Data |
|---|---|
| 00 | 11100 |
| 01 | |
| 10 | 11110<br>01010 |
| 11 | 11111<br>11011 |

1F0 → **conflict miss**

# Calculating Cache Performance

**Cache Miss Rate (m_r):** The fraction (usually expressed as a percentage) of memory accesses not found in a level of the cache.

**Miss Penalty (m_p):** Time (in cycles) to replace a block in a cache level.

**Cycles Per Instruction (CPI),** is the average time (in cycles) that a processor takes to execute an instruction without cache misses.

**Memory Access Rate**: ratio (or percentage) of instructions that access memory.

**Increase in CPI (Due to Cache Misses) = m_r x m_p x memory access rate**

**Note:** The increase in access time is calculated recursively for a multi-level cache.

# Calculating Cache Performance

**Example**

Assume that the miss rate of a data cache is 4%. If a processor has a CPI of 2 without any cache misses, and the miss penalty is 100 cycles for any miss, determine the average access time of the cache. Assume the frequency of all loads and stores is 36%.

Average Access time `= CPI + Increase in access time`

`= CPI + m_r x m_p x memory_access_rate`

`= 2 + (.04 x 100 x .36) = 3.44`

# Summary

- We have seen a review of the different cache mapping schemes and how a main memory address is interpreted accordingly.
- We have seen the process of ***referencing***, ***mapping*** and ***placing*** memory blocks in the cache.
- We have seen 2 examples using the same main memory architecture and the same sequence of memory references over 2 different cache mapping schemes.
- An associative cache reduces (but does not eliminate) conflict misses and in general gives a higher cache hit ratio but it all depends on the nature of memory references.
- Associative cache allows more memory blocks to stay in cache.
- Set Associative caches implement a ***Replacement Policy***.