CSE 230

Computer Organization CSE 230

Dr. Gamal Fahmy

Lecture 8

Memory Hierarchy Design

Memory Hierarchy Design

- As time goes, programmers need unlimited memory
- Large memories are expensive, inefficient, and slow
- Memory hierarchy is the optimal solution for the cost-performance trade off in memory technologies
- ***Principle of locality***

Memory Hierarchy Design

- ***Principle of locality***
- ***Amdahl's law** states that the performance improvement to be gained from using some faster mode of execution is limited by the fraction of the time the faster mode can be used*
- ***Overall speed up=***

$$\frac{1}{(1-f)+\frac{f}{s}}$$

- ***For a computer that has 10% of its code run 90 faster, speed up = 1.1097***
- ***For a computer that has 90% of its code run 10 faster, speed up = 5.2631***

Memory Hierarchy Design

- **90/10 rule** comes from empirical observation: *"A program spends 90% of its time in 10% of its code"*
- we can predict with reasonable accuracy what *instructions* and *data* a program will use in the near future based on its accesses in the recent past.
- **Two different types** :

  Temporal locality: states that recently accessed items are likely to be accessed in the near future.

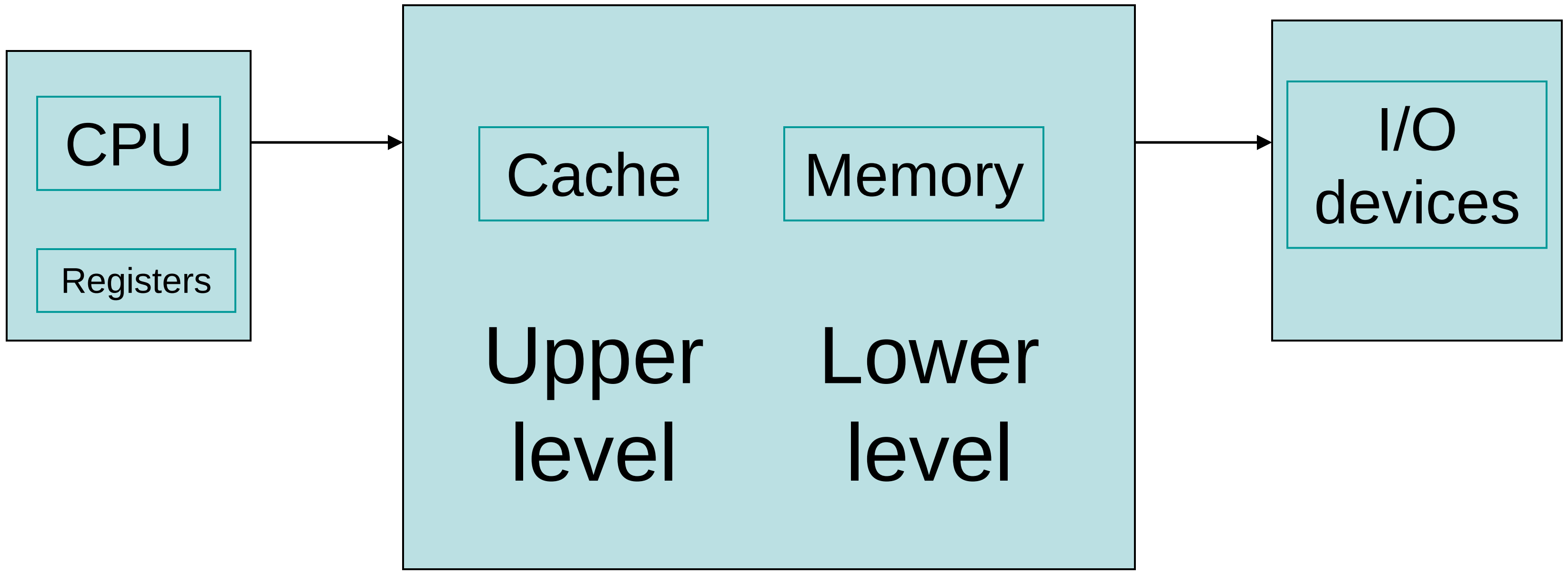
- **Spatial locality**: says that items whose addresses are near one another tend to be referenced close together in time.

Memory Hierarchy Design

- *Smaller memories hold the most recently accessed items close to the CPU and successively larger (and slower, and less expensive) memories as we move away from the CPU.*

- This type of organization is called **a memory hierarchy** .
- Two important levels of the memory hierarchy are **the cache** and **virtual memory.**

Memory Hierarchy Design



Size	200B	64KB	32MB	2GB
Speed	5ns	10ns	100ns	5ms

Hits, misses ?

Memory Hierarchy Design

- To evaluate the effectiveness of the memory hierarchy we can use the formula:
- ***Memory_stall_cycles =***

IC * Mem_Refs * Miss_Rate * Miss_Penalty

• where IC	= Instruction count
• Mem_Refs	= Memory References per Instruction
• Miss_Rate	= the fraction of accesses that are not in the cache
• Miss_Penalty	= the additional time to service the miss

Example

- A computer have 1.0 CPI, memory access (loads and stores) are 50% of the instructions, miss penalty is 25 clock-cycle and we have 2% miss rate, how much faster with no misses

- CPU execution =(CPU clock cycle + Memory stall cycles)*clock cycles
- Memory stalls cycles=IC* (Memory_access/Instruction) *miss rate*miss penalty
- CPU execution with no miss=(CPU clock cycle)*clock cycles

Memory Hierarchy Design

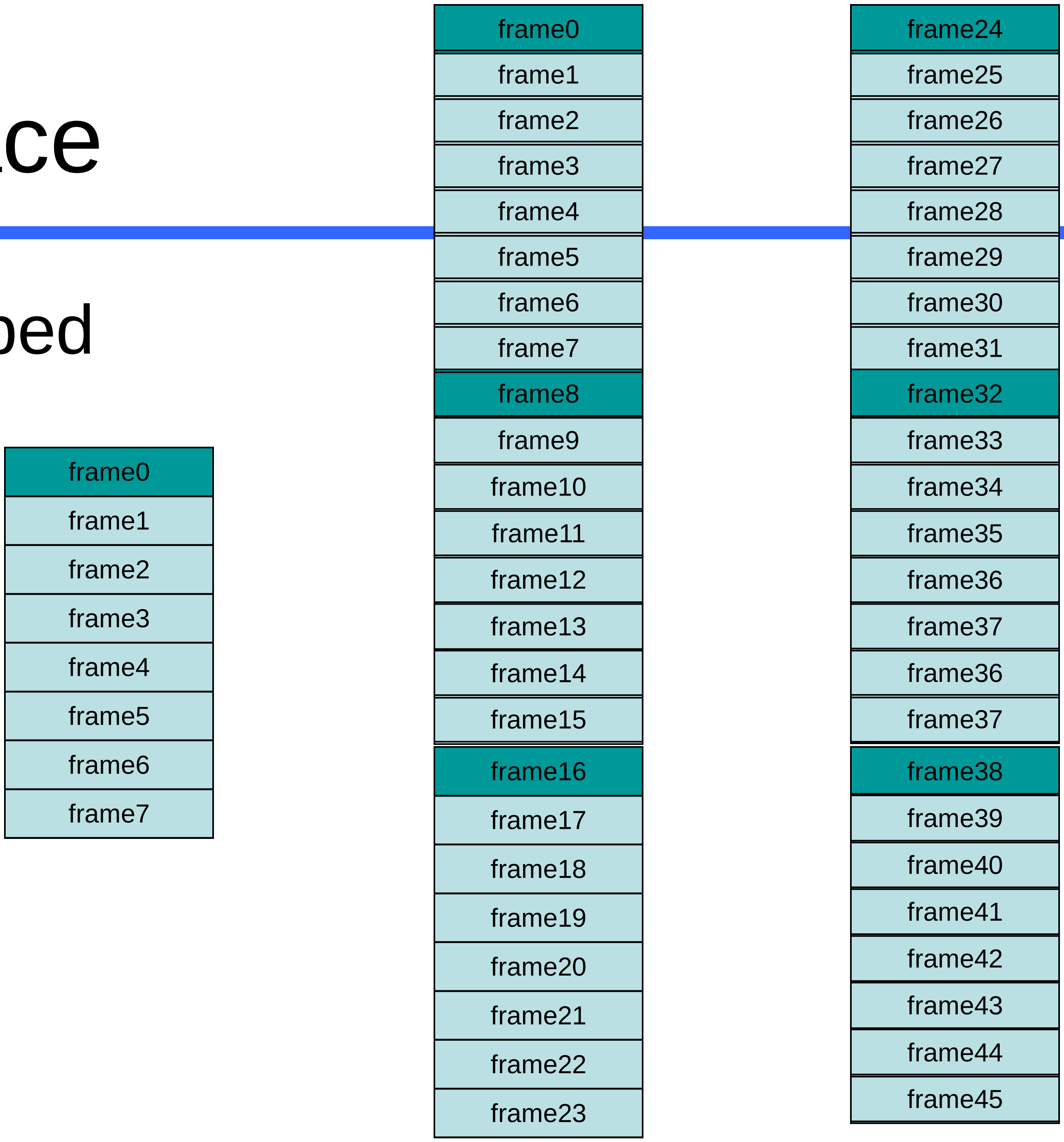
- Four main issues to consider when designing a hierarchical memory
  - ***Block Place***
  - ***Block ID***
  - ***Block replaced***
  - ***Cache Main memory interactions***

Block Place

- 3 methods to place blocks in the cache
  - Direct mapped: has only one slot
    - » (Block address) MOD (Number of blocks in cache)
  - Fully associative: can be any where in the memory
  - Set associative : can be within a set of places
    - » (Block address) MOD (Number of sets in cache)

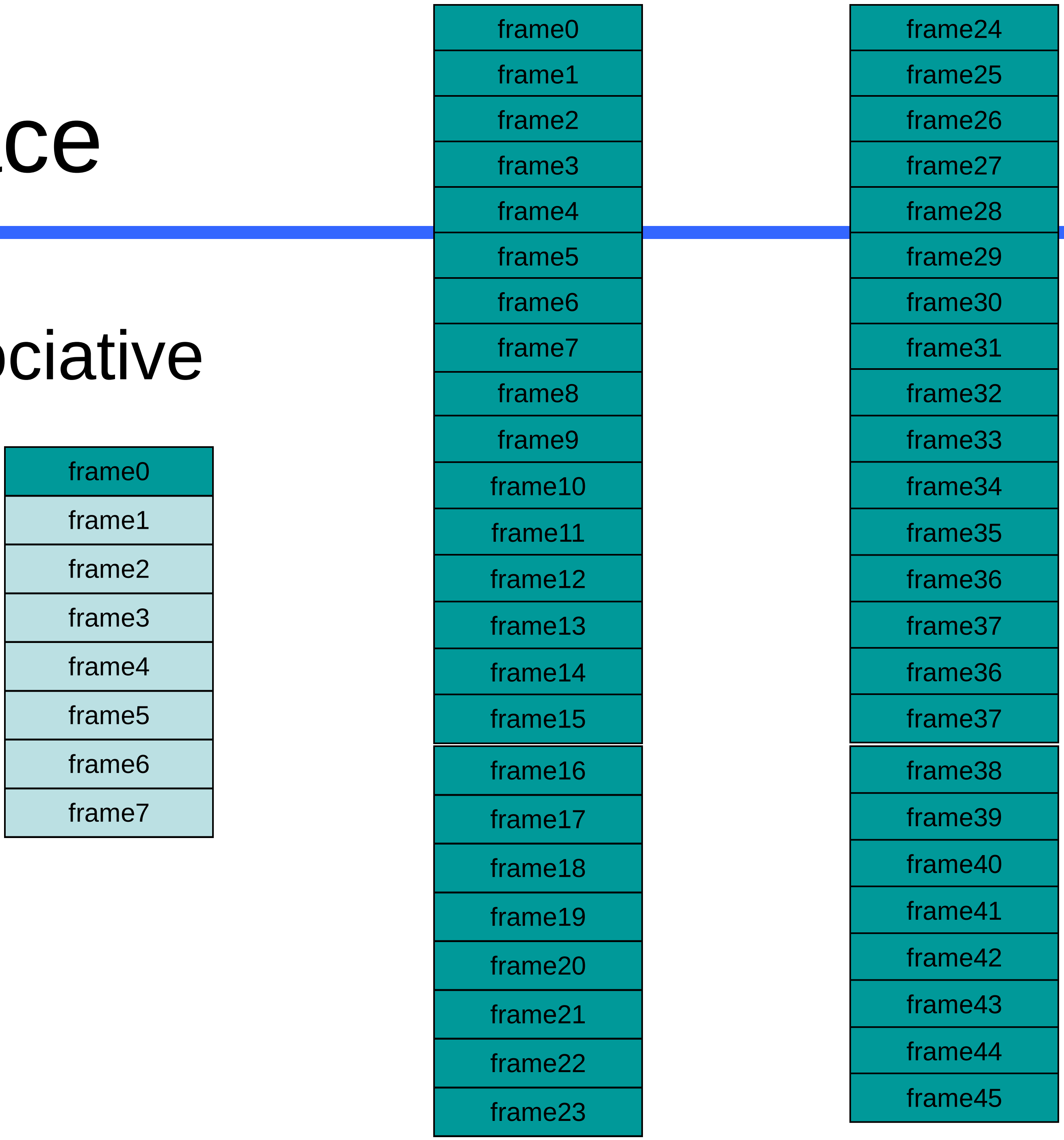
Block Place

Direct mapped



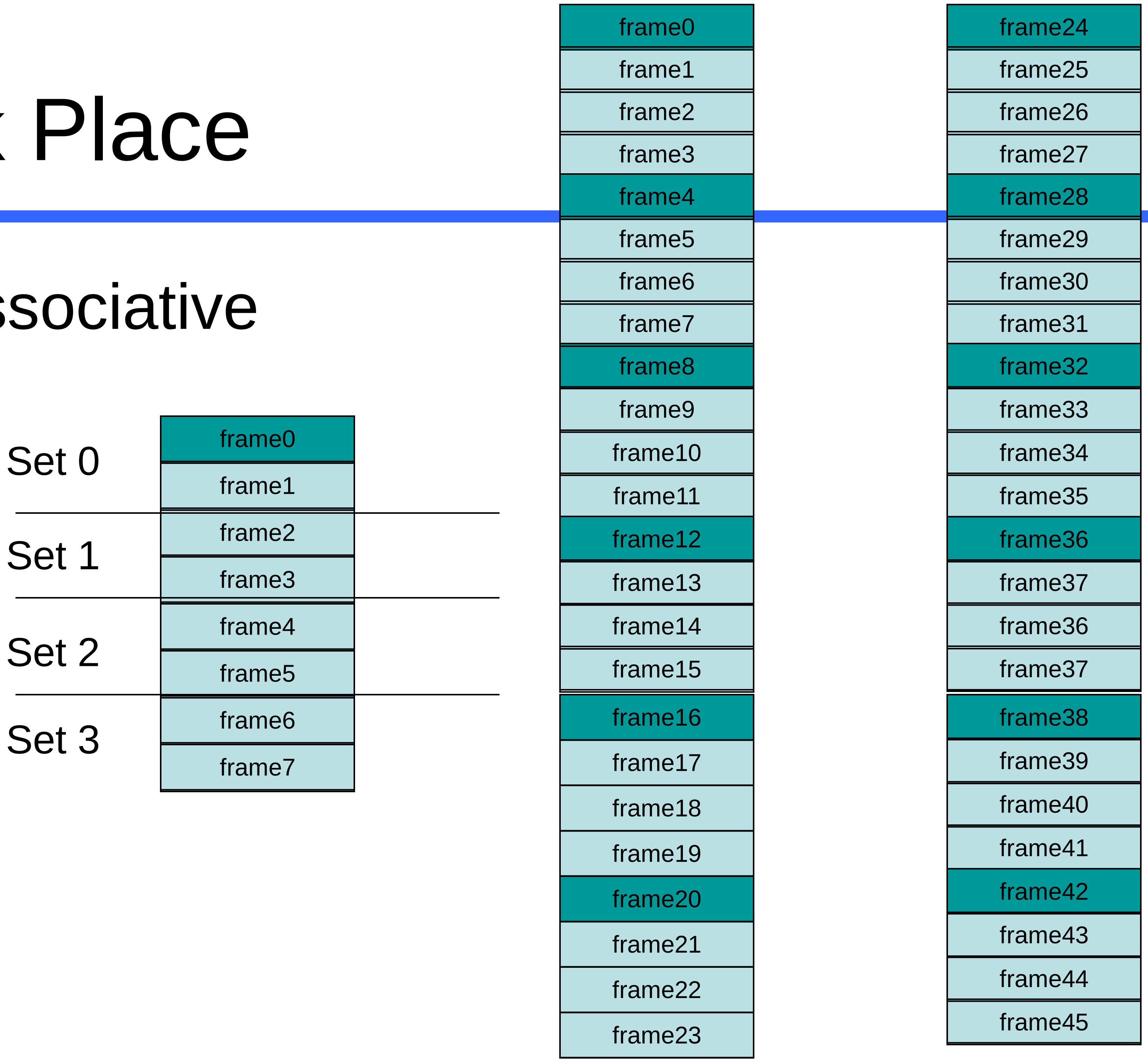
Block Place

Fully associative



Block Place

Set associative



Block Identification

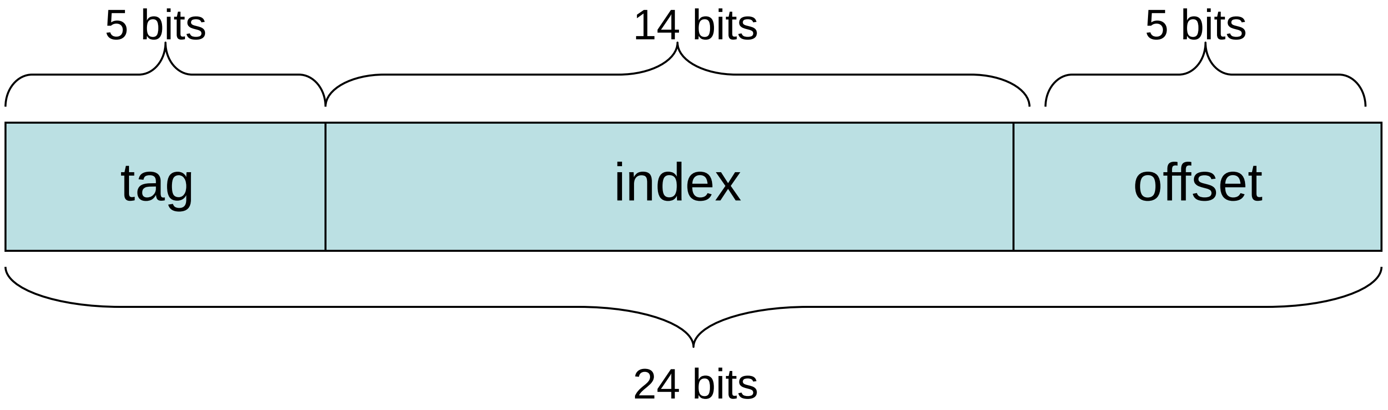
- **Cache memory consists of two portions:**
- **Directory**
  - **Address Tags (checked to match the block address from CPU)**
  - **Control Bits (indicate that the content of a block is valid)**

  RAM
  - **Block Frames (contain data of blocks)**

- **For an address structure, that has 16 MB memory size, 512 KB cache size, and 32 Block size, what is the address for the fully associative methodology, direct mapped, and set associative (set size is 2 blocks)**

Block Identification

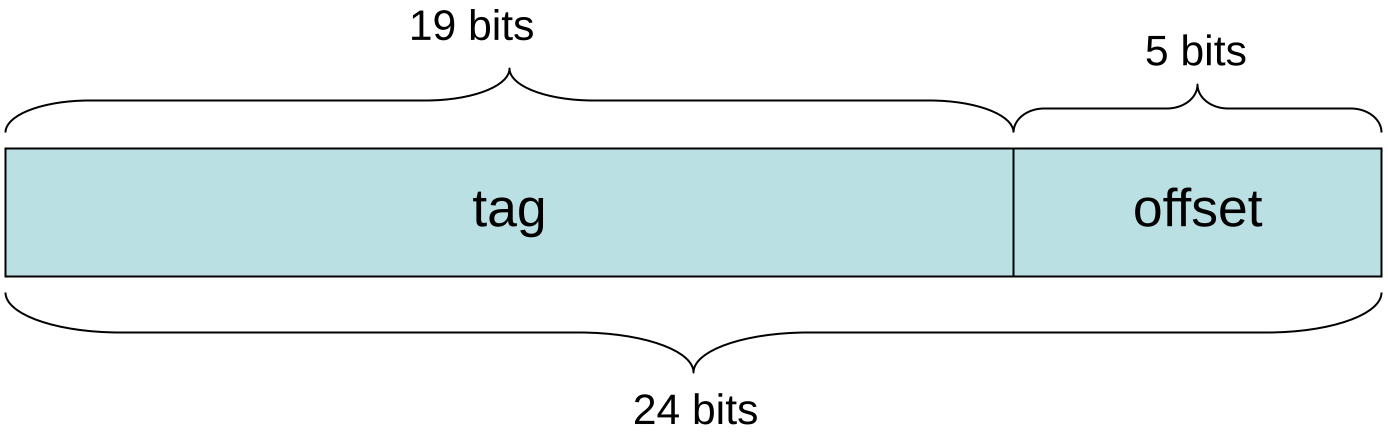
- Direct mapped
  - » Memory size=16 MB=2** 24 bits
  - » Block size=32 B=2**5 bits
  - » Number 0f blocks in cache= cache size/Block size=2**14 bits
  - » Number of bits in tag=24-5-14=5



Block Identification

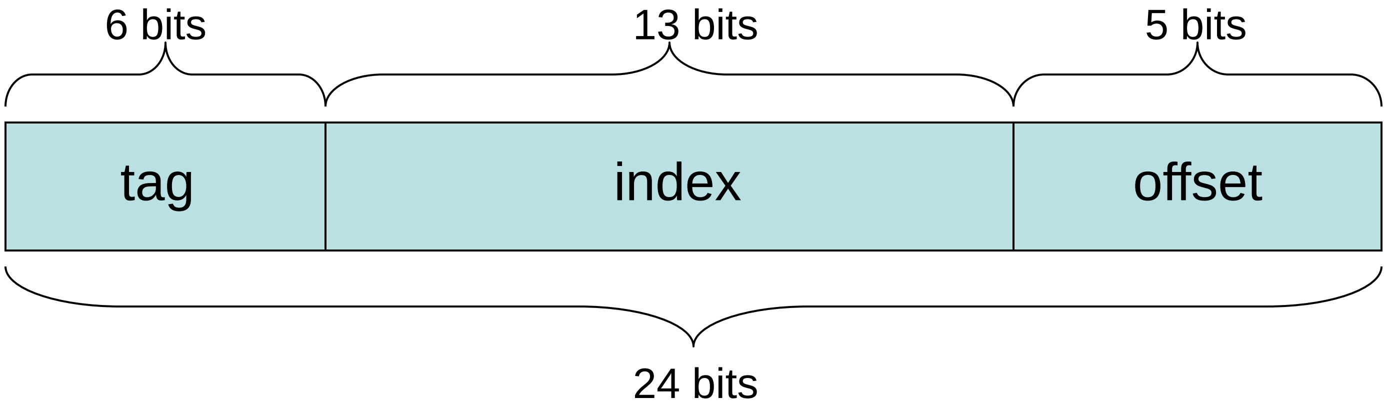
- Fully associative
  - » Memory size=16 MB=2** 24 bits
  - » Block size=32 B=2**5 bits

  - » Number of bits in tag=24-5=19



Block Identification

- Set Associative
  - » Memory size=16 MB=2** 24 bits
  - » Block size=32 B=2**5 bits
  - » Number 0f sets in cache= cache size/ (set size * Block size) = 512KB/(2*32 B) = 2**13 bits
  - » Number of bits in tag=24-13-5=6



Block Replacement

- ***When a miss occurs, the cache controller must select a block to be replaced with the desired data.***
- ***With direct-mapped placement the decision is simple because there is no choice: only one block frame is checked for a hit and only that block can be replaced***
- ***With fully-associative or set-associative placement, there are more than one block to choose from on a miss***
- **Strategies**
  - » ***First In First Out (FIFO)***
  - » ***Most-Recently Used (MRU)***
  - » ***Least-Frequently Used (LFU)***
  - » ***Most-Frequently Used (MFU)***
  - » ***Random***
  - » ***Least-Recently Used (LRU)***

Block Replacement

- ***Most two popular strategies***
  - ***Random*** - to spread allocation uniformly, candidate blocks are randomly selected.
    Advantage: simple to implement in hardware
    Disadvantage: ignores principle of locality

  - ***Least-Recently Used (LRU)*** - The block replaced is the one that has been unused for the longest time.

    Advantage: takes locality into account

    Disadvantage: as the number of blocks to keep track of increases, LRU becomes more expensive (harder to implement, slower and often just approximated).

Example

- A computer with 256K memory and 4K cache that is organized in a set associative manner, with 4 block frames per set and 64 words per block. The cache is 10 times faster. If cache is initially empty, suppose we fetch 4352 words from locations 0,1….4351 in order, then repeat it 14 more times. Specify tag, index and offset, and estimate the speedup from cache using LRU

Memory size=256 KW=2**18
Block size =64 W=2**6
Number of sets in cache=4KW/(4*64W)=2**4
Number of bits in Tag=18-4-6=8

Example

Cache Structure

S0	S1	S2	S3	s4	s5	S6	S7	S8	S9	S10	S11	S12	S13	S14	S15
F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0	F0
F1	F1	F1	F1	F1	F1	F1	F1	F1	F1	F1	F1	F1	F1	F1	F1
F2	F2	F2	F2	F2	F2	F2	F2	F2	F2	F2	F2	F2	F2	F2	F2
F3	F3	F3	F3	F3	F3	F3	F3	F3	F3	F3	F3	F3	F3	F3	F3

Cache is 10 times faster then main memory. Assuming **Cache Access Time = *t***, then **Memory Access Time = *10*t***

We have 15 iterations, the total number of fetched blocks=4352/64=68

Total time for fetches without cache=Ninstr.*Memory access time*Niter=4352*10**t**15=652800**t*
Total time with for fetches with cache= ***Time for Fetches From Cache (on hits) +Time for Fetches From Memory (on misses);***

Example

- **Time for Fetches From Cache (on hits) =**

  Ninst. * Cache Access Time * Niter;
- **Time for Fetches From Memory(on misses) =**

  Nmisses * Miss Penalty;

- **Miss Penalty = Block Size * Memory Access Time = *64 * 10 * t***
- First iteration, Nmiss=68
- Second iteration, Nmiss=20
- Third iteration, Nmiss=20 and so on
- Nmisses = 68 + 20*14 = *348*
- **Total Time for Fetches With Cache = 4352*15*t + 348*(64*10*t)= 65280*t + 222720*t= *288000*t***

Example

- **Speedup = Total Time for Fetches without Cache/ Total Time for Fetches with Cache= 652800*t / 288000*t =2.*26***
- Repeat same problem using MRU
- **Take home quiz**

Memory Interaction with Cache

- Reads dominate processor cache accesses. All instruction accesses are reads, and most instructions do not write to memory.
- The read policies for a miss are:
  Read Through **-** reading a word from main memory to CPU
  No Read Through **-** reading a block from main memory to cache and then from cache to CPU

Memory Interaction with Cache

- The **write policies** on write *hit* often distinguish cache designs:
- ***Write Through*** - the information is written to both the block in the cache and to the block in the lower-level memory.
- ***Advantage:***
  - read miss never results in writes to main memory
  - easy to implement
  - main memory always has the most current copy of the data (consistent)
- ***Disadvantage:***
  - write is slower
  - every write needs a main memory access
  - as a result uses more memory bandwidth

Memory write on a hit (cont.)

- ***Write back*** - the information is written only to the block in the cache. The modified cache block is written to main memory only when it is replaced.

- ***Advantage:***
  - writes occur at the speed of the cache memory
  - multiple writes within a block require only one write to main memory
  - as a result uses less memory bandwidth
- ***Disadvantage:***
  - harder to implement
  - main memory is not always consistent with cache
  - reads that result in replacement may cause writes to main memory

Memory write on a miss (cont.)

- ***Write Allocate*** - the block is loaded on a write miss, followed by the write-hit action.
- ***No Write Allocate*** - the block is modified in the main memory and not loaded into the cache.
- Although either write-miss policy could be used with write through or write back, **write-back** caches generally use **write allocate** (hoping that subsequent writes to that block will be captured by the cache) and **write-through** caches often use **no-write allocate** (since subsequent writes to that block will still have to go to memory).

Memory Interaction with Cache

Possible combinations of interaction policies with main memory on write.

Write hit policy	**Write miss policy**
Write Through	Write Allocate
Write Through	No Write Allocate
Write Back	Write Allocate
Write Back	No Write Allocate

Memory Interaction with Cache

- ***Write Through with Write Allocate:***
- on hits it writes to cache and main memory
- on misses it updates the block in main memory and brings the block to the cache
- Bringing the block to cache on a miss does not make a lot of sense in this combination because the next hit to this block will generate a write to main memory anyway (according to Write Through policy)

Memory Interaction with Cache

- ***Write Through with No Write Allocate:***
- on hits it writes to cache and main memory;
- on misses it updates the block in main memory not bringing that block to the cache;
- Subsequent writes to the block will update main memory because Write Through policy is employed. So, some time is saved not bringing the block in the cache on a miss because it appears useless anyway.

Memory Interaction with Cache

- ***Write Back with Write Allocate:***
- on hits it writes to cache setting “dirty” bit for the block, main memory is not updated;

- on misses it updates the block in main memory and brings the block to the cache;

- Subsequent writes to the same block, if the block originally caused a miss, will hit in the cache next time, setting dirty bit for the block. That will eliminate extra memory accesses and result in very efficient execution compared with Write Through with Write Allocate combination.

Memory Interaction with Cache

- ***Write Back with No Write Allocate:***

- on hits it writes to cache setting “dirty” bit for the block, main memory is not updated;

- on misses it updates the block in main memory not bringing that block to the cache;

- Subsequent writes to the same block, if the block originally caused a miss, will generate misses all the way and result in very inefficient execution.

Memory Interaction with Cache

Possible combinations of interaction policies with main memory on write.

Write hit policy	**Write miss policy**
4 Write Through	Write Allocate
1 Write Through	**No Write Allocate**
2 Write Back	**Write Allocate**
3 Write Back	No Write Allocate