Inf2C - Computer Systems
Lecture 13
Memory Hierarchy and Caches

Boris Grot

School of Informatics

University of Edinburgh

Previous lecture: multi-cycle processor

- Execution time:

```
(inst count) x (cycles/inst) x (cycle time)
```

- Multi-cycle processor: reduce the cycle time by breaking up an instruction's execution into multiple simple operations (1 or 2 per cycle)
  – Control FSM sequences the cycles
  – Reuse components (memory, ALU) to reduce "cost"

Memory requirements

- Programmers wish for memory to be
  - Large
  - Fast
  - Random access
- Wish not achievable with 1 kind of memory
  - Issues of cost and technical feasibility
- Idea of a **memory hierarchy**: approximate the "ideal" large+fast memory through a combination of different kinds of memories

Memory examples

Technology	Typical access time	Price per GB
SRAM	1-10 ns	£1000
DRAM	~100 ns	£10
Flash SSD	~100 μs	£1
Magnetic disk	~10 ms	£0.1

Which of these is "main memory"? **DRAM**

Memory hierarchy overview

- Use a combination of memory kinds
  - Smaller amounts of expensive but fast memory closer to the processor
  - Larger amounts of cheaper but slower memory farther from the processor
- Idea is not new:

"Ideally one would desire an indefinitely large memory capacity such that any particular … word would be immediately available… we are … forced to recognize the possibility of constructing a hierarchy of memories, each of which has greater capacity than the preceding but which is less quickly accessible."

A. W. Burks, H. H. Goldstine, and J. von Neumann - 1946

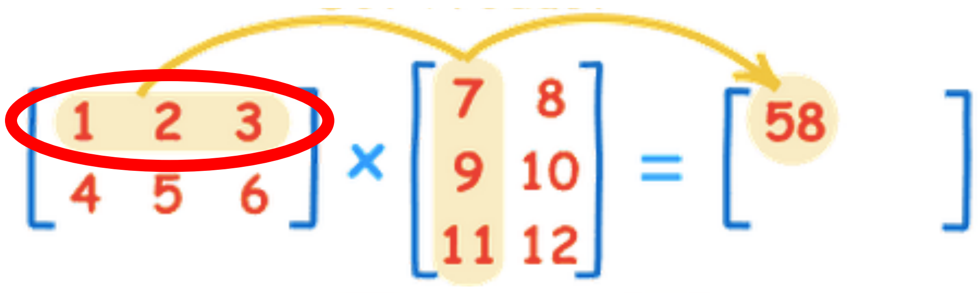
Why is a memory hierarchy effective?

- Temporal Locality:
  - A recently accessed memory location (instruction or data) is likely to be accessed again in the near future
- Spatial Locality:
  - Memory locations (instructions or data) close to a recently accessed location are likely to be accessed in the near future
- Why does locality exist in programs?
  - Instruction reuse: loops, functions
  - Data working sets: arrays, temporary variables, objects

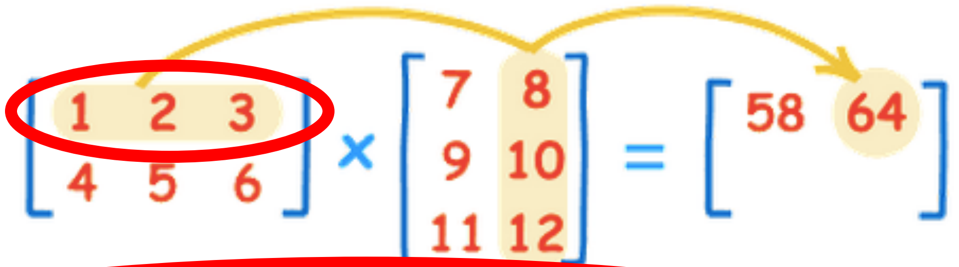
Example of Temporal & Spatial Locality

Matrix – matrix multiplication:

Spatial locality



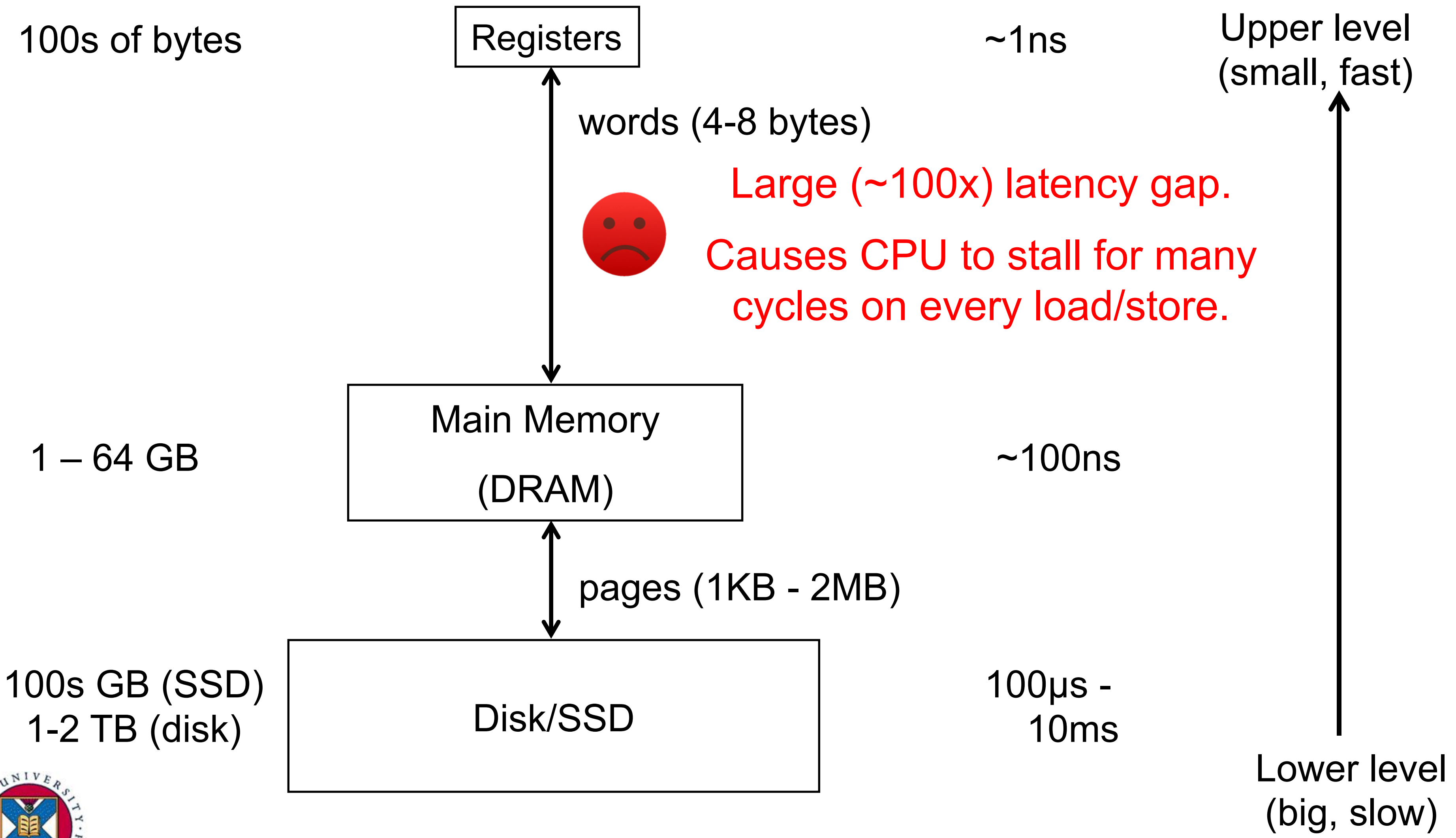
Temporal locality



```
for i = 1 to M
  for j = 1 to N
    for k = 1 to P
      c[i,j] = c[i,j] + a[i,k] * b[k,j]
```

Temporal & spatial locality in the code itself

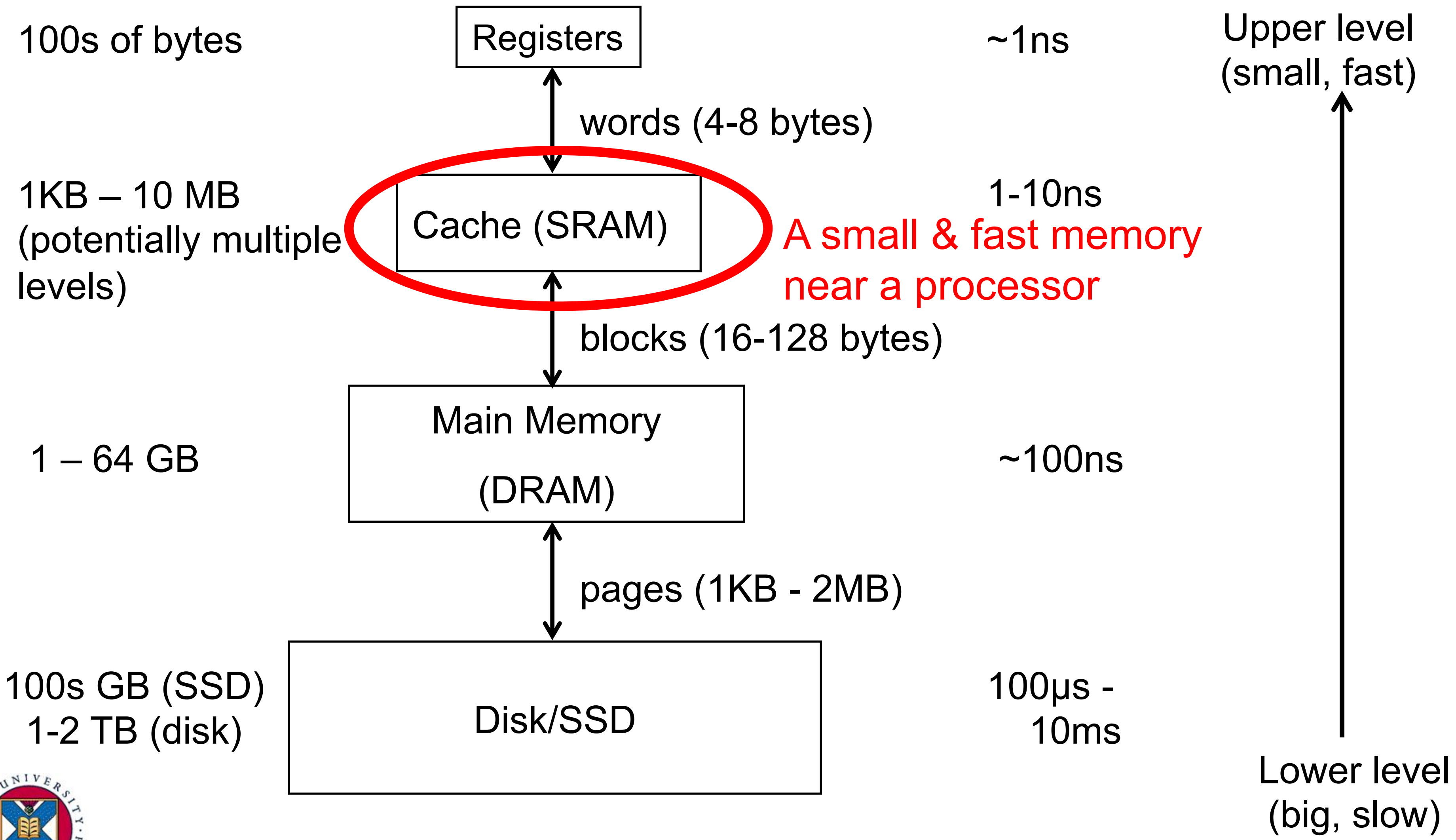
Levels of the memory hierarchy



Large (~100x) latency gap.

Causes CPU to stall for many cycles on every load/store.

Levels of the memory hierarchy



Memory hierarchy in a modern processor

- Small, fast **cache** next to a processor backed up by larger & slower cache(s) and main memory give the impression of a single, large, fast memory
- Take advantage of temporal locality
  - If access data from slower memory, move it to faster memory
  - If data in faster memory unused recently, move it to slower memory
- Take advantage of spatial locality
  - If need to move a word from slower to faster memory, move adjacent words at same time
  - Gives rise to **block** & **pages**: units of storage within the memory hierarchy composed of multiple contiguous words

Control of data transfers in hierarchy

- Q. Should the SW or HW be responsible for moving data between levels of the memory hierarchy?

- A. It depends: there is a trade-off between ease of programming, hardware complexity, and performance.
  - *SW (compiler):* between registers and cache/main memory
  - *HW:* between caches and main memory (SW is usually unaware of caches)
  - *SW (Operating System):* between main memory and disk

HW-managed transfers between levels

- Occurs between cache memory and main memory levels
- Programmer & processor both oblivious to where data resides
  - Just issue loads & stores to "memory"
- Cache Hardware manages transfers between levels
  - Data moved or copied between levels automatically in response to the program's memory accesses
  - Memory always has a copy of cached data, but data in the cache may be more recent
    - This creates interesting problems.
      Discussed in Computer Architecture and Parallel Architectures ☺

Cache terminology

- Block (or line): the unit of data stored in the cache
  - Typically in the range of 32-128 bytes
- Hit: data is found (this is what we want to happen)
  - Memory access completes quickly
- Miss: data not found
  - Must continue the search at the next level of the memory hierarchy (could be another cache or main memory)
  - After data is eventually located, it is copied to the memory level where the miss happened

More cache terminology

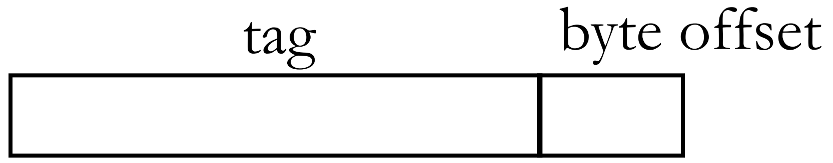
- Hit rate (hit ratio): fraction of accesses that are hits at a given level of the memory hierarchy
- Miss rate (miss ratio): fraction of accesses that are misses at a given level   (= 1 – hit rate)
- Allocation: placement of a new block into the cache, which typically results in an eviction of another block.
- Eviction: displacement of a block from the cache, which commonly happens when a new block is allocated in its place.

Cache basics

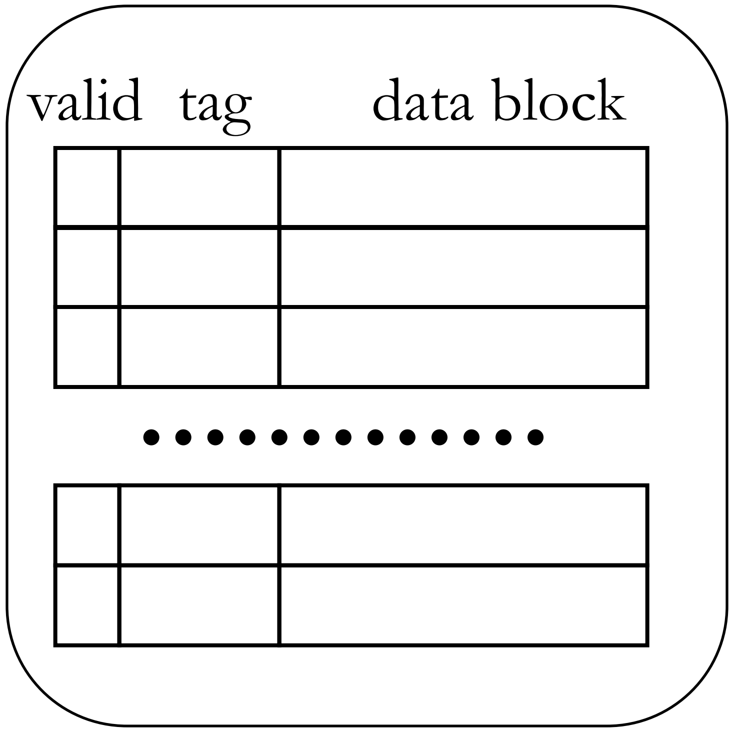
- Data are identified in (main) memory by their full 32-bit address
- Problem: how to map a 32-bit address to a much smaller memory, such as a cache?
- Answer: associate with each data block in cache:
  - a **tag** word, indicating the address of the main memory block it holds
  - a **valid bit**, indicating the block is in use

Fully-associative cache

requested address:



Cache



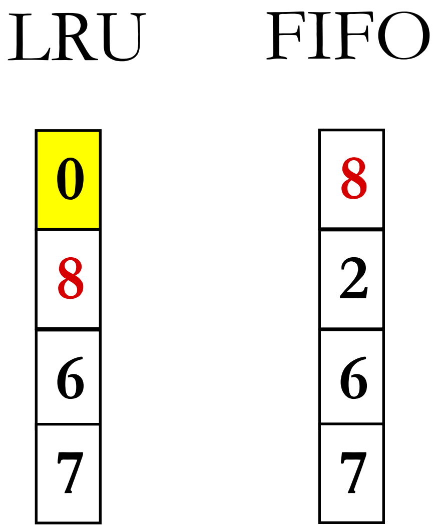
Correct cache block identified by matching tags

Byte offset selects word/byte within block

Address tag can potentially match tag of *any* cache block

Cache Replacement

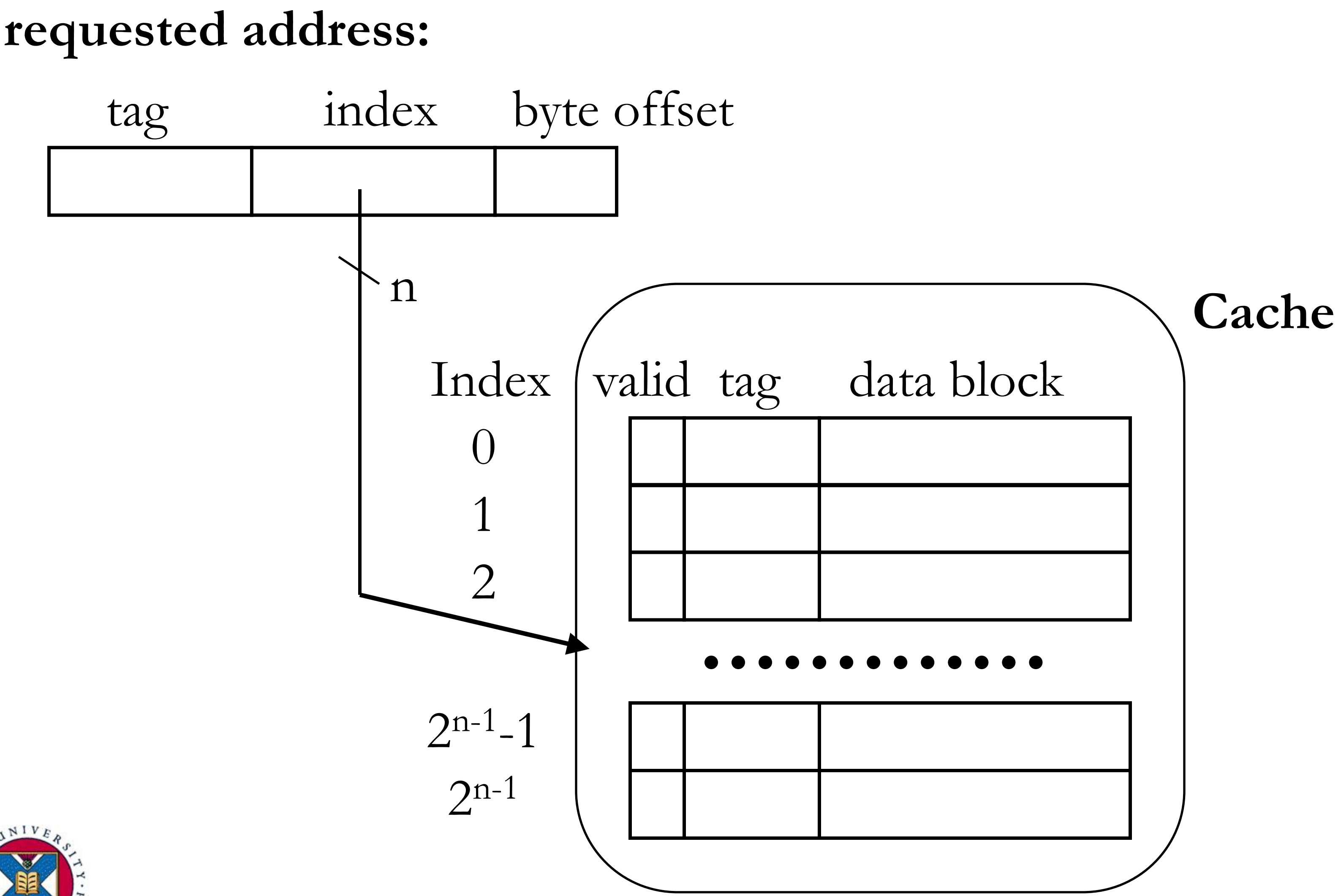
- Least Recently Used (LRU)
  - Evict the cache block that hasn't been accessed longest
  - Relies on past behaviour as a predictor of the future
- FIFO – replace in same order as filled
  - Simple to implement
- Example:
  - Cache with 4 blocks
  - Access addresses: 0 2 6 0 7 8



Direct-mapped cache

- In a fully-associative cache, search for matching tags is either very slow, or requires a very expensive memory type called Content Addressable Memory (CAM)
- By restricting the cache location where a data item can be stored, we can simplify the cache
- In a **direct-mapped** cache, a data item can be stored in one location only, determined by its address
  - Use some of the address bits as index to the cache array

Address mapping for direct-mapped cache



Example problem

Given a 4 KB direct-mapped cache with 4-byte blocks and 32-bit addresses.

Question: How many tag, index, and offset bits does the address decompose into?

Example problem

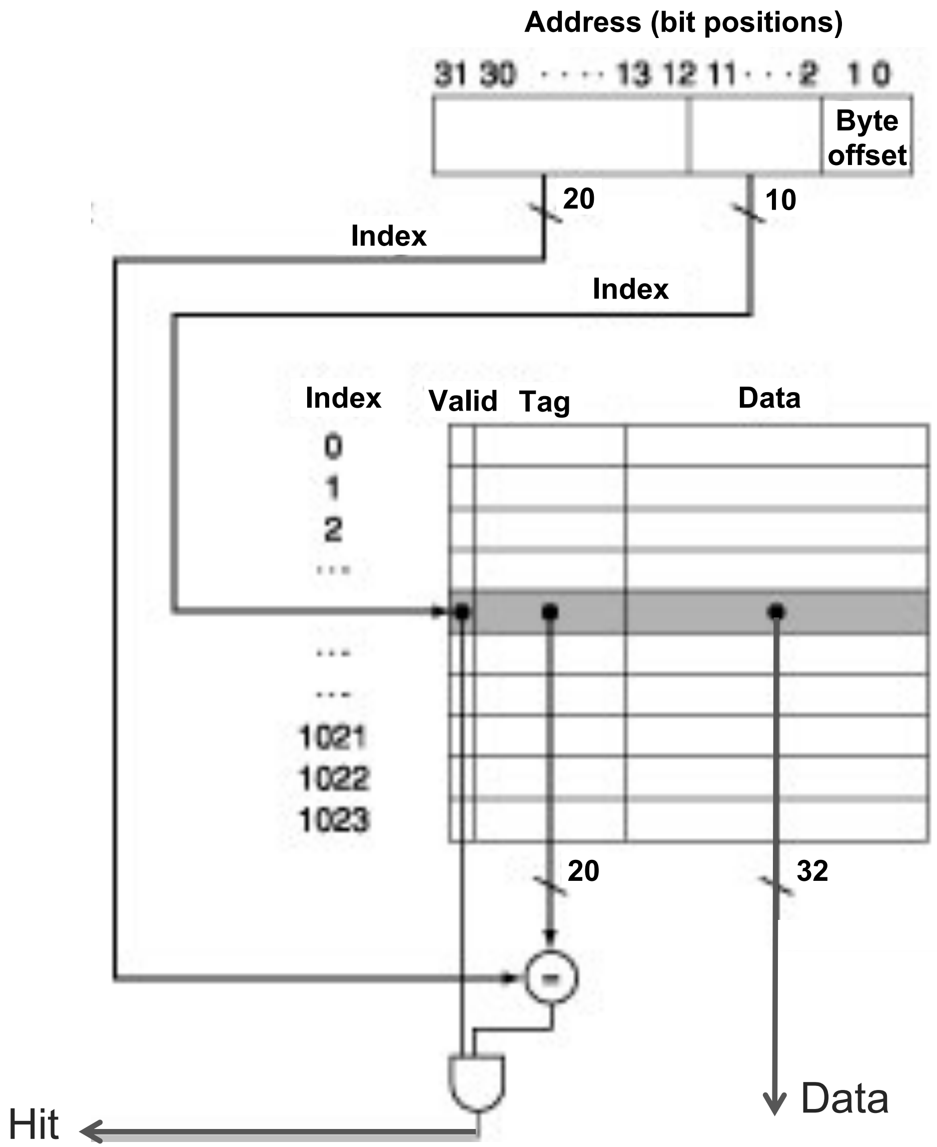
Given a 4 KB direct-mapped cache with 4-byte blocks and 32-bit addresses.

Question: How many tag, index, and offset bits does the address decompose into?

Answer:

- 4 KB / 4 bytes per block = 1K blocks
  - Requires a 10-bit index
- 4-byte block: requires a 2-bit offset
- Tag: 32 – 10 – 2 = 20 bits

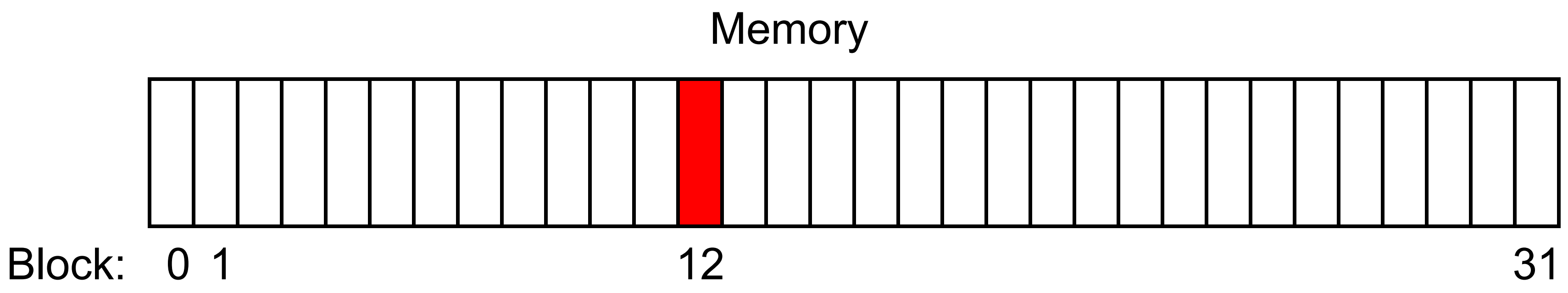
Direct-mapped cache in detail

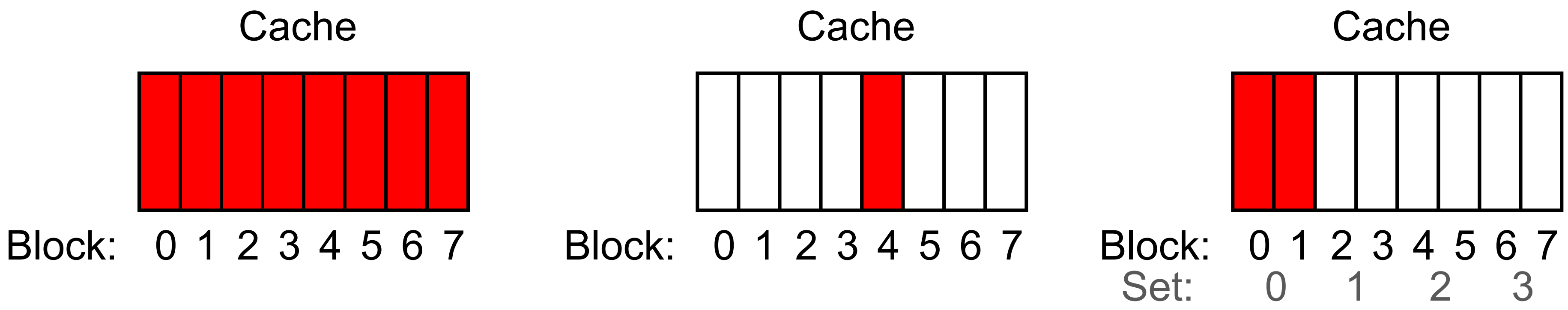


Cache Associativity Options

- Fully Associative
  - The block can go into any location in the cache
  - Good: Most flexible approach → lowest miss rate
  - Bad: Must search the whole cache to find the block (bad for speed and power)
- Direct mapped
  - The block can only go into one location in the cache
  - Good: very simple hardware (fast and low power)
  - Bad: Blocks mapping to the same location (**thrashing**) → increased miss rates
- Set Associative
  - Split the cache into groups (**sets**) of *m* blocks each → **m-way set-associative**
  - A given block can only go into one set (based on block address), but within that set it can go anywhere
  - Typical degree of associativity is 2 – 16

Cache Block Placement





Fully associative:

block 12 can go anywhere in the cache

Direct mapped:

block 12 can only go into block 4 (12 mod 8)

Set associative:

block 12 can go anywhere in set 0 (12 mod 4)

Example problem

Given a 4 KB, 4-way set-associative cache with 4-byte blocks and 32-bit addresses.

Question: How many tag, index, and offset bits does the address decompose into?

Answer:

– 4 KB / 4 bytes per block = 1K blocks

– But.. there are 4 ways per set → 256 sets

  ▪ Requires an 8-bit index to select the set

– 4-byte block: requires a 2-bit offset

– Tag: 32 – 8 – 2 = 22 bits

Writing to caches: on a hit

- Write through – write to both cache and memory
  - Good: memory and cache always synchronized
  - Bad: writes are slow and require memory bandwidth
- Write back – write to cache only
  - Each cache block has a dirty bit, set if the block has been written to
  - When a dirty cache block is replaced, it is written to memory
  - Good: writes are fast and generate little memory traffic
  - Bad: memory can have stale data for some time

Writing to caches: on a miss

- Write allocate – bring the block into the cache and write to it
  – Useful if locality exists
- Write no-allocate – do not bring the block into the cache; modify data only in memory
  – Useful if no locality
  – Guarantees that cache and memory are synchronized (have the same value for an address)

Coursework 2: cache simulator

Goals: understand caches, get practice with C

Your job: using C, write a configurable cache simulator that supports the following parameters:

- Number of blocks
- Block size
- Associativity
- Replacement policy

Input: list of addresses (in a file)

Output: cache configuration details and hit/miss stats

Due: Wed, Nov 28, 4pm

It’s only a simulator!

- **Simulator:** Not the same as real hardware!
  - Don’t think bits & bytes; think data types!
  - E.g., is `bit` the best data type for indicating that a cache block is valid?

- The cache will not store data
  - Cache size refers only to the size of the data portion

Coursework 2 Tips

- Start the coursework by understanding lecture slides, and Tutorial 4 (questions 2 & 3)
  - Don’t ignore the text book – it’s very useful
  - Read others’ questions on Piazza
    - Read previous questions to avoid double posting
- You must allocate memory dynamically
- You must have a clean, modular simulator design
- Your code must compile & run on DICE with the specified flags (and only those flags)