

Caches

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

June 11, 2017

Overview I

- 1 Principle of locality
- 2 Basics
- 3 Cache hits and misses
- 4 A basic cache design
- 5 Memory to cache mapping
- 6 Cache tags
- 7 Valid bit
- 8 Cache hit
- 9 Cache miss
- 10 Cache example
- 11 Cache sizes
- 12 Cache performance
 - Example
- 13 Larger Block Sizes
- 14 Placement of bytes within a block
- 15 Building byte address

Overview II

- 16 Examples
- 17 Problem with direct mapping
- 18 Fully associative cache
- 19 Set associative cache
- 20 Set associative cache addressing
 - Example
- 21 Block replacement strategies
- 22 Writing to memory
 - Write-through cache
 - Write-back cache
- 23 Write misses
- 24 Cache organisations
- 25 Larger cache blocks and memory bandwidth

Principle of locality

- Programs access a small proportion of their address space at any time.
- **Temporal locality**
 - Items accessed recently are likely to be accessed again soon
 - E.g. Instructions in a loop, induction variables
- **Spatial locality**
 - Items near those accessed recently are likely to be accessed soon
 - E.g. Sequential instruction access, array data

Basics

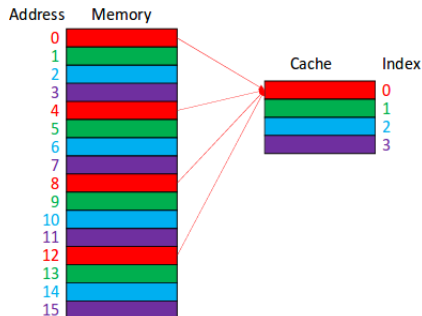
- The first time processor reads from an address in the main memory, we can make a copy of the data and put it into the cache memory
- Next time the data from the same address is needed, we can use its copy that we have stored in the cache
- The first read was slow, but then every subsequent read is much quicker
- We have taken advantage of the temporal locality.
- We can also take advantage of the spatial locality
- Instead of copying only the data from the requested memory address, we can copy the data residing nearby e.g. instead of a byte, we can copy the whole word.
- Next time any of the four bytes are needed, they are available from the cache
- Of course, the initial loads were slow, but we predict the data will be used soon likely more than once

Cache hits and misses

- A **cache hit** occurs if the data requested by the CPU appears somewhere in the cache
- If the data is not there, we have a **cache miss**
- Further, we can define a **hit rate** i.e. the fraction of memory accesses found in the cache.
- Analogously, we define a **miss rate** which is $1 - \text{hitrate}$.
- Typically, cache hit rates are above 95% which tells us that the vast majority of memory accesses are fast since they are handled by the cache.

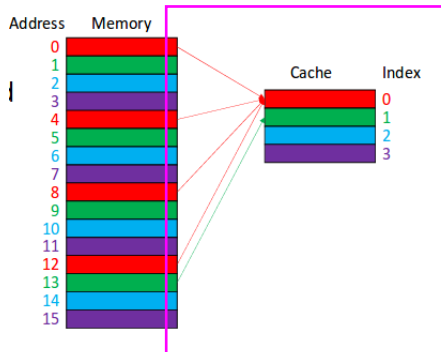
A basic cache design

- Each cache consists of blocks, which take advantage of the spatial locality.
- Let's assume for a moment that the block size is one byte (no spatial locality yet)
- The most basic design is so called direct-mapped cache where each memory location is mapped to exactly one location in the cache



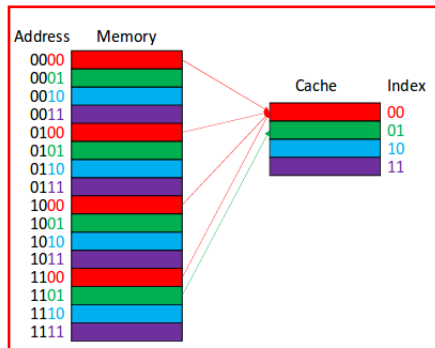
Memory to cache mapping I

- The memory address is mapped to the cache index using the mod (remainder) division
- The size of cache memory is 2^k
- Then, the memory address a is mapped to $a \% 2^k$
- E.g. $13 \Rightarrow 13 \% 4 = 1$



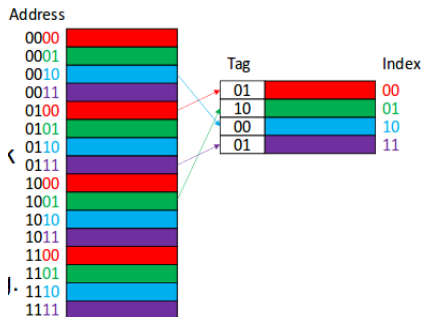
Memory to cache mapping II

- Instead of calculating mod division, could look at k least significant bits of the address
- E.g. 1101 \Rightarrow 01



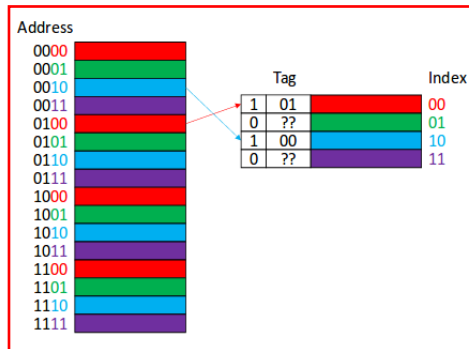
Cache tags

- Cache blocks need to be supplemented with tags
- A tag contains the address information necessary to establish whether the associated cache block corresponds to a requested address
- A tag together with the corresponding index form the corresponding memory address
- E.g. Tag - 01 and Index - 00 correspond to memory address: 0100.



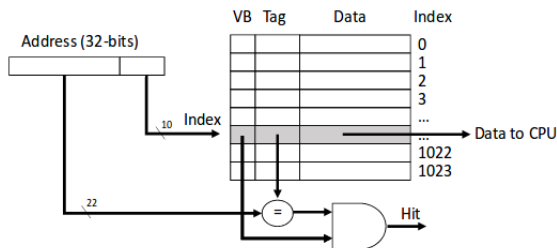
Valid bit

- Initially the cache memory contains invalid data in all blocks
- On the first load to the cache block the corresponding valid bit is set to 1.



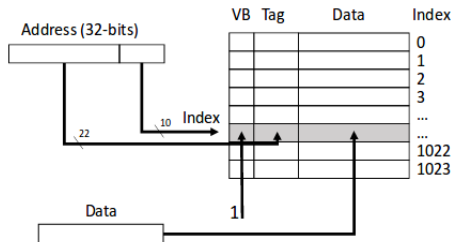
Cache hit

- On memory read, the address is sent to a cache controller
- The address is subdivided into lower k bits which index a block and upper bits for tag matching
- If the block is valid ($VB=1$) and its tag matches the upper bits of the address, the data is sent to the CPU



Cache miss

- A slow memory access is needed
- The simplest solution is to stall the pipeline until the data is fetched from memory and copied into cache



Cache example

- 8 blocks, direct-mapped:

Address	Binary Address	Hit/miss	Cache block
22	10 110	Miss	110
Address	Binary Address	Hit/miss	Cache block
26	11 010	Miss	010
Address	Binary Address	Hit/miss	Cache block
16	10 000	Miss	000
3	00 011	Miss	011
16	10 000	Hit	000

VB	Tag	Data	Index
1	10	Mem[16]	000
0	??		001
1	11	Mem[26]	010
1	00	Mem[3]	011
0	??		100
0	??		101
1	10	Mem[22]	110
0	??		111

Cache sizes

- Direct-mapped
- 1 byte per block
- 32 blocks
- handles 16-bit addresses

$$32 \times (8b + 1b + 11b) = 32 \times 20b = 640b$$

VB

Tag

Note, since there are 32 blocks, index has 5b so tag needs $16 - 5 = 11b$

Cache performance

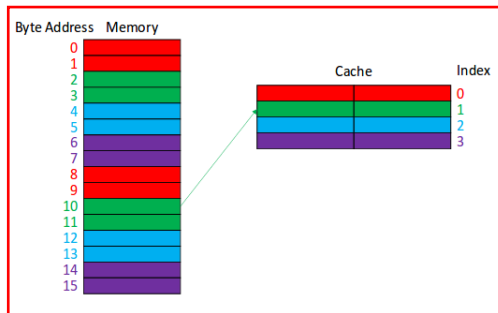
- **Hit time** - Time to access cache, usually 1-3 clock cycles
- **Miss penalty** - Time to move data from memory to cache, at least tens of clock cycles, often a lot longer
- **Average memory access time (AMAT)**
 - $AMAT = hit\ time + (miss\ rate \times miss\ penalty)$
 - The lower AMAT, the better
 - Since miss penalty is much greater than hit time, the best strategy to lower AMAT is to reduce miss rate or miss penalty

Example

- Say hit rate is 98%, hit time is one clock cycle and the miss penalty is 50 clock cycles. What is AMAT?
 - $AMAT = 1 + (0.02 \times 50) = 2$
 - If hit rate was perfect, the AMAT would be one clock cycle, with a hit rate just below 1, the AMAT has doubled
 - Spatial locality increases hit rate/reduce miss rate

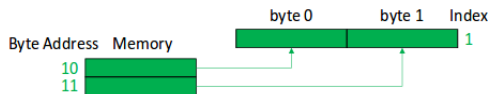
Larger Block Sizes

- Using block size of two:
- To map memory address to its block address we use integer division
- E.g.
 $10 \div 2 = 5$, $11 \div 2 = 5$
- Block address 5 belongs to cache block 1 since $5 \% 4 = 1$



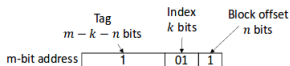
Placement of bytes within a block

- If a program reads byte 10 from the main memory, we copy both 10 and 11 from the cache
- The same will happen if the program requests byte 11.

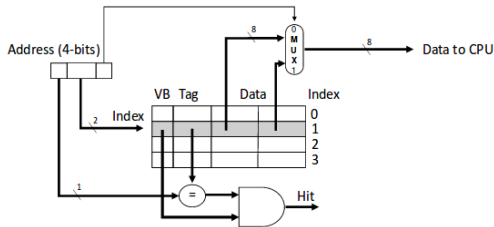


Building byte address

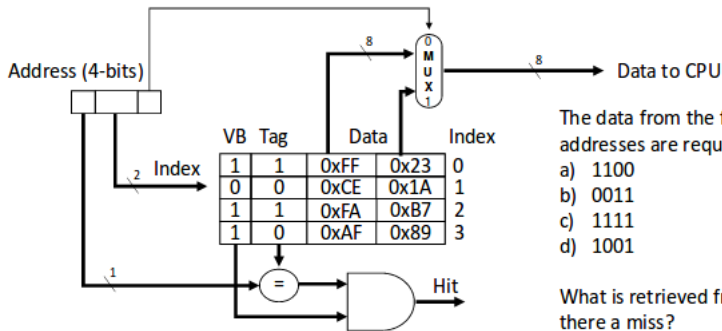
- For 2^k cache size and 2^n block size:
- In example:
 - 2^4 memory size
 - 2^2 cache size
 - 2^1 block size
 - Byte 11 (1011)



Note, block offset is memory address mod 2^n



Examples



The data from the following addresses are requested:

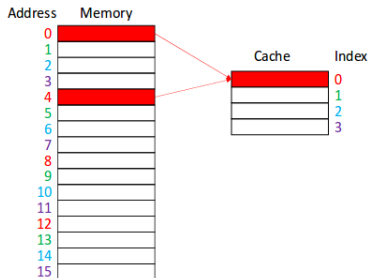
- a) 1100
- b) 0011
- c) 1111
- d) 1001

What is retrieved from cache or is there a miss?

- a) 0xFA
- b) miss - invalid
- c) miss - wrong tag
- d) 0x23

Problem with direct mapping

- Going back to the first example, and the load being 0,4,0,4,0,4 etc
- Since both map to the same cache block, all loads will be cache misses.
- The cache contains 4 blocks, and not using them efficiently

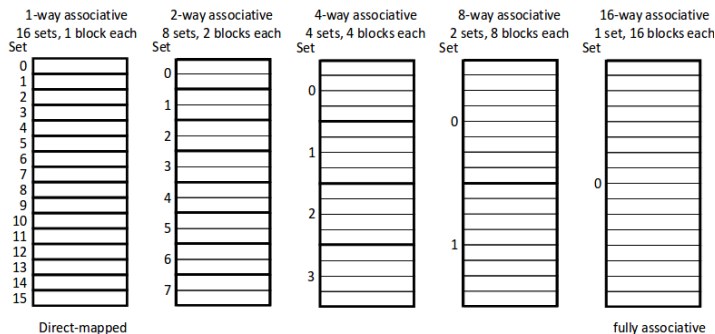


Fully associative cache

- Unlike direct-mapped cache, a **full associative cache** can map a memory address to any block in the cache and hence solves multiple misses issue.
- If all blocks are full, you could replace the one which was **least recently used**
- The luxury of fully associative cache comes with **significantly increased complexity of the hardware**
- **No index field** and therefore **entire address is used as a tag**
- Even worse, the **block can be stored anywhere in the cache**, so **all the tags** have to be compared against required address.

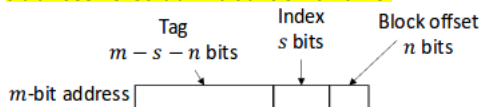
Set associative cache

- A compromise between the direct mapped cache and the fully associative cache is a **set associative cache**.
- Cache is divided into groups of blocks called **sets**
- Each memory address is mapped to only one set. However, the blocks of data can be placed anywhere within this set
- A 2^x **-way associative cache** has 2^x blocks in each set



Set associative cache addressing

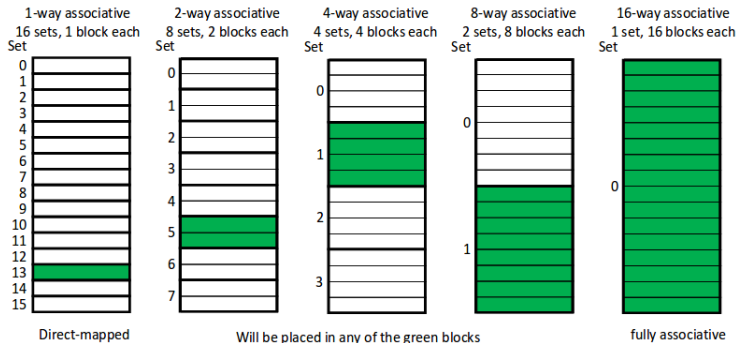
- Similar to direct-mapped.
- Instead of cache index, use a **set index**
- Assuming a cache has 2^s sets and each block 2^n bytes, the memory address is subdivided as follows:



- The computations of relevant fields are analogous to what was seen earlier.

Example

- Assuming 16 block design and 16 bytes per block where will the address 7129 be placed:
 - 7129 is 0...01101111011001:
 - For the 1-way cache, 1101
 - For the 2-way cache, 101
 - For the 4-way cache, 01
 - For the 8-way cache, 1



Block replacement strategies

- For associative caches, can use any non-empty (invalid) block for new data
- If all blocks are valid, replace the least recently used block
- This requires keeping statistics of block access
- For 2-way associative cache, nly one bit per set, call it LRU, the bit is inverted on every cache miss
- For highly associative caches, keeping track of the least recently used block may be expensive and often approximations are used.

Writing to memory

- More complicated than reading
- Assume the address we want to write is already in the cache, can update its value and avoid slow memory access
- Now the memory and cache contain inconsistent values
- This may create problems if the memory is shared with other devices

Write-through cache

- A simple solution is **write-through cache** which forces all writes to update both memories
- But makes writes take longer
 - If base CPI = 1, 10% of instructions are stores, write to memory takes 100 cycles
 - Effective CPI = $1 + 0.1 \times 100 = 11$
- **Write buffer** can help with the above problem
- It will hold data waiting to be written into memory
- CPU continues immediately
- Only stalls on write if buffer is already full

Write-back cache

- A **write-back cache** is an alternate option
- It **updates the memory only if the cache block needs to be replaced**
- We **would write the data to cache first and leave it inconsistent with the memory**
- **We would mark** such a cache block “dirty” to indicate inconsistency
- Any **subsequent load instruction accessing the same memory would be serviced by the cache**
- The “dirty” value will be stored in the main memory only **after the cache block in which it resides needs to be replaced.**
- The **writes can be also buffered as was the case with writ-through cache**
- **Write-back cache is potentially more efficient as it takes advantage that not all write operations need to access main memory**

Write misses

- Another scenario arises if we want to write to an address that is not contained in the cache - **write miss**
- Two policies:
 - **Write around:** (write-no-allocate) is advantageous when data is stored, but then is not immediately used again so there is no point of having in the cache yet.
 - **Write allocate:** is advantageous when data is needed soon

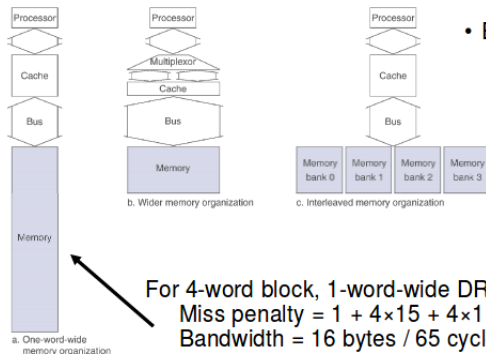
Cache organisations

- Data/Instruction caches:
 - Pros: No structural hazards between IF and MEM stages
 - Cons: Can be bad if instruction/data sets are unbalanced
- Cache hierarchies:
 - There is a trade-off between access time and hit rate
 - Smaller L1 cache can focus on access time
 - Larger L2 cache can focus on good hit rate

Larger cache blocks and memory bandwidth I

- Larger cache blocks impose additional miss penalties as we need to perform some number of individual memory accesses
- The miss penalty can be decreased by widening the memory and its interface to the cache
 - The cost is the disadvantage
- Another solution is the interleaved memory
 - Memory split into banks which can be accessed individually
 - Overlapping the latencies of accessing each word
 - Pipeline concept

Larger cache blocks and memory bandwidth II



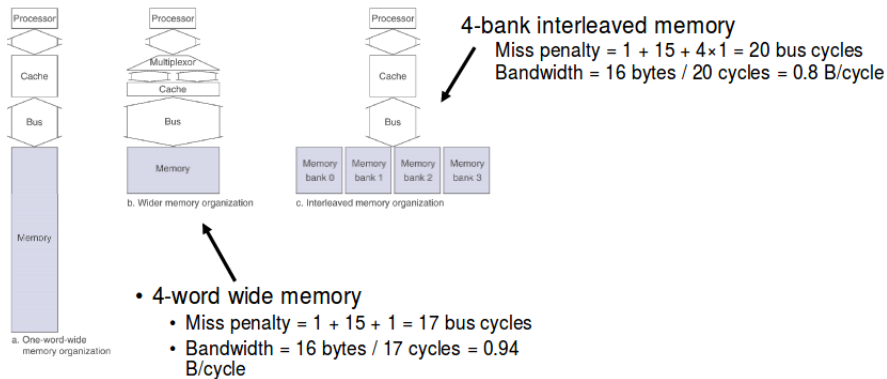
- Example cache block read
 - 1 bus cycle for address transfer
 - 15 bus cycles per DRAM access
 - 1 bus cycle per data transfer

For 4-word block, 1-word-wide DRAM

Miss penalty = $1 + 4 \times 15 + 4 \times 1 = 65$ bus cycles

Bandwidth = $16 \text{ bytes} / 65 \text{ cycles} = 0.25 \text{ B/cycle}$

Larger cache blocks and memory bandwidth III



The End