CSC 230 - Summer 2018

Caching

Bill Bird

Department of Computer Science
University of Victoria

July 22, 2018

Memory Review (1)

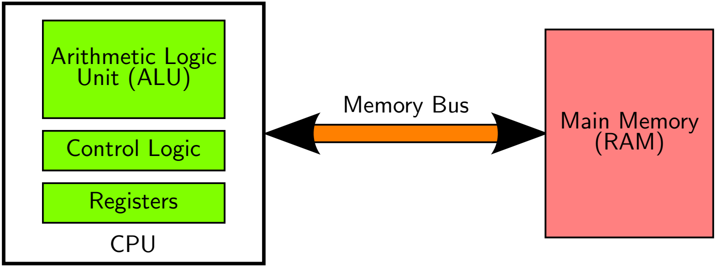
Recall that a memory system whose memory locations are 8 bits wide is called **byte addressable**. There is no general requirement that a memory be byte addressable, but most modern architectures use byte addressable memories for data.

Also recall that if a memory has addresses which are k bits wide and data locations which are d bits wide, the number of valid addresses is 2^k and the maximum capacity of the memory (in bits) is $2^k \cdot 2^d = 2^{k+d}$.

Memory Review (2)

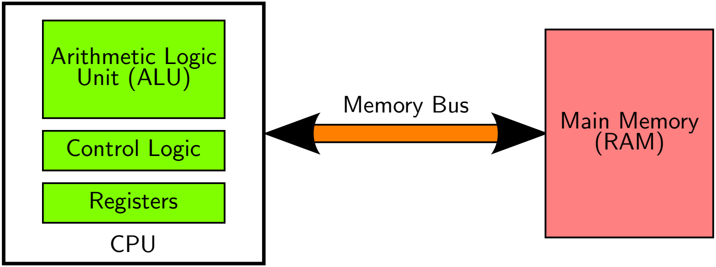
The component containing the memory controller and associated logic is the **memory management unit** (MMU). In some cases (e.g. AVR), the MMU may consist only of control circuitry for memory banks. On more complicated architectures, the MMU may also contain circuitry for abstractions like virtual memory (which will be covered in the next lecture).

Processors and Memory (1)



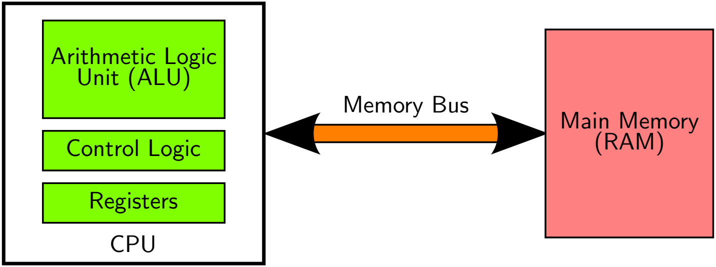
In a von Neumann architecture, a single main memory is used to store both programs and data. Main memory is regarded as 'external' to the CPU (at least conceptually) and, unlike onboard registers, requires special logic to access.

Processors and Memory (2)



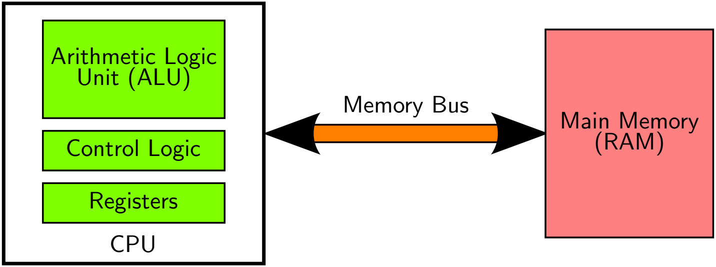
In CISC architectures, the 'special logic' is not much different from register-based operations, but in RISC architectures memory access is limited to load and store instructions.

Processors and Memory (3)



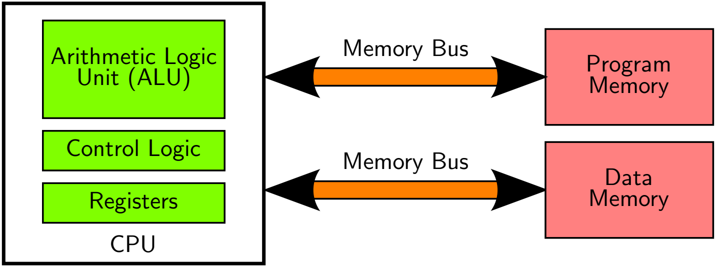
In either case, the CPU must constantly fetch from memory as it executes code, since the program itself is stored in memory. One consequence of the von Neumann model is that, with the arrangement depicted above, instructions and data must share a memory channel, so two memory-related pipeline stages (e.g. fetch and memory load) cannot execute simultaneously.

Processors and Memory (4)



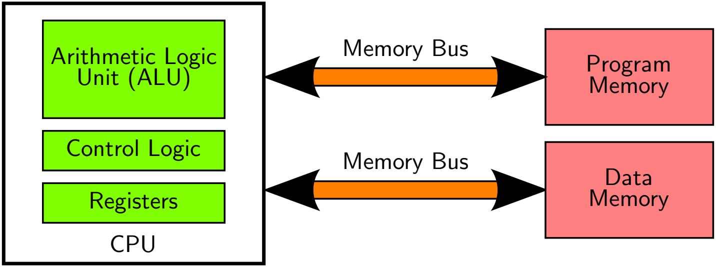
(It is possible to add hardware support for two simultaneous memory accesses to remedy this issue)

Processors and Memory (5)



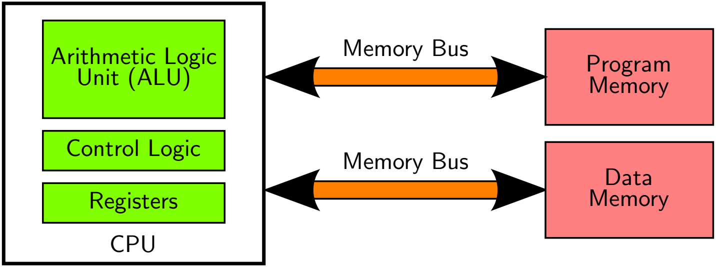
In a Harvard architecture, separate memories are used for program instructions and data, with the added benefit of separate pathways for instructions and data.

Processors and Memory (6)



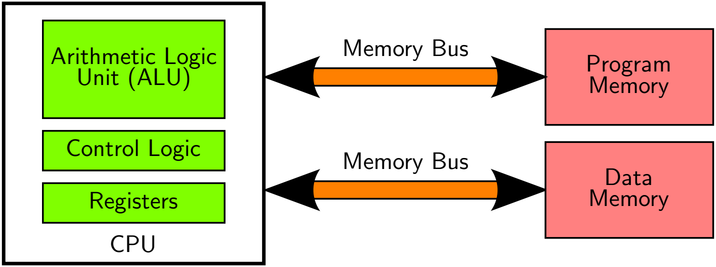
Embedded processors (like the ATmega2560) have memory circuitry which runs at the same speed as the CPU. Such memory is expensive, but the cost is reduced by the (comparatively) slow CPU speed and the small amount of memory needed (8KB for the ATmega2560).

Processors and Memory (7)



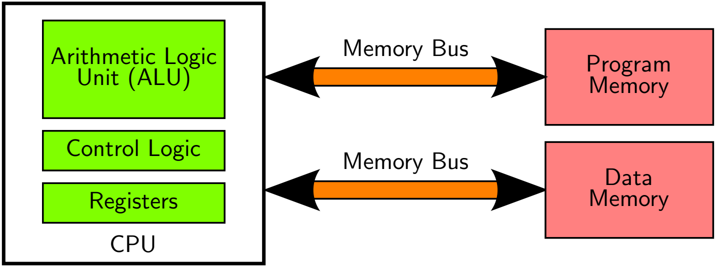
Processors designed for high performance settings (such as the processors in laptops and desktops) generally have much higher clock speeds (in the gigahertz range) and several gigabytes of main memory.

Processors and Memory (8)



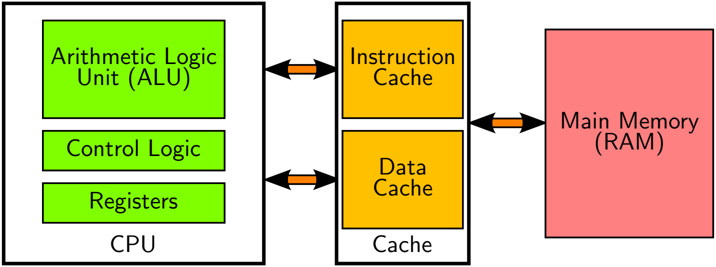
Memory circuitry which matched the speed of a modern desktop processor would be prohibitively expensive in the quantities needed for a desktop's main memory, so the speed of memory tends to be an order of magnitude lower than the speed of the processor.

Processors and Memory (9)



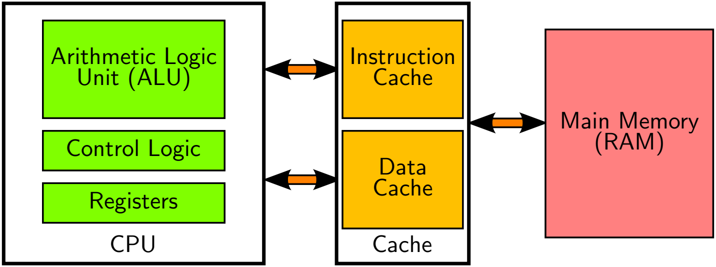
As a result, accessing main memory may require dozens of CPU clock cycles. Since the use of memory is an integral part of a stored program architecture, there is no general way to reduce memory access. Even if loads and stores can be avoided, every instruction must be fetched from memory before execution.

Processors and Memory (10)



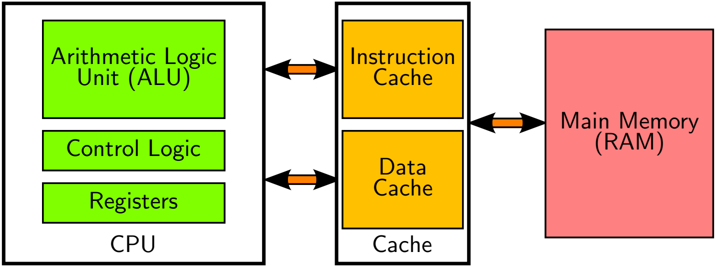
To mitigate this, a **cache** can be added between the processor and memory. The cache stores a small subset of the larger main memory to reduce the number of reads and writes to the slower memory.

Processors and Memory (11)



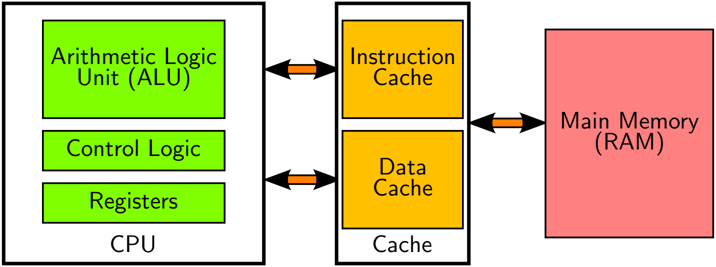
The most basic type of cache is a special region of main memory which is implemented with very fast circuitry. This model requires the programmer or compiler to actively track which data is stored in the fast cache area and which data is stored in the slower area.

Processors and Memory (12)



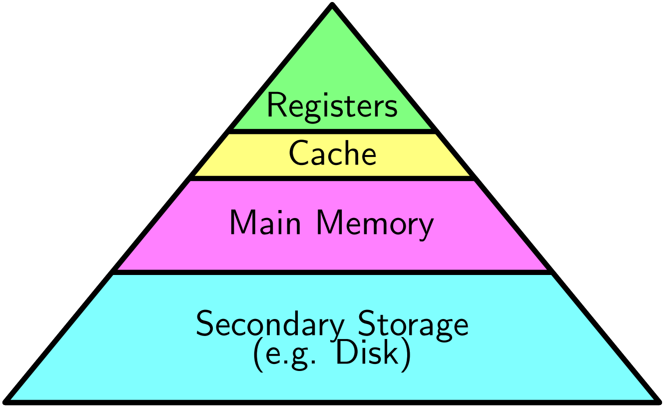
Caching systems used on modern processors are usually implemented to allow arbitrary blocks of main memory to be cached, and to be relatively transparent to the programmer.

Processors and Memory (13)



The x86 architecture uses two separate cache structures for program instructions and data. Modern x86 processors also use several layers of cache (instead of a single cache). The split cache structure makes the x86 architecture a hybrid of the von Neumann and Harvard models.

Processors and Memory (14)



Cache can be considered to be a component of the larger **memory heirarchy**, which stratifies the various types of storage available to the system. Generally, the upper levels are small, fast and expensive and the lower levels are very large, slow and cheap.

Locality (1)

Caching systems rely on two types of **locality** to speed up memory access. These are heuristics only, but they tend to hold very strongly in practice.

Spatial Locality: If an address M is accessed, assume that addresses close to M (such as $M + 1$ or $M - 10$) will also be accessed.

Temporal Locality: If an address M is accessed, assume that M will be accessed again soon.

Locality (2)

Spatial locality is strongly exhibited by the following programming constructs.

Data Loads and Stores:

- Arrays (when allocated as contiguous blocks, as in C or Java).
- Stack access (since the stack is essentially an array)

Instruction Loads:

- Linear (non-branching) code: a sequence of instructions without branches will all need to be loaded from memory and executed.
- Localized branching code (such as the code containing relative branches with very short jumps, such as those produced by the AVR branch instructions).

Locality (3)

Temporal locality is strongly exhibited by the following programming constructs.

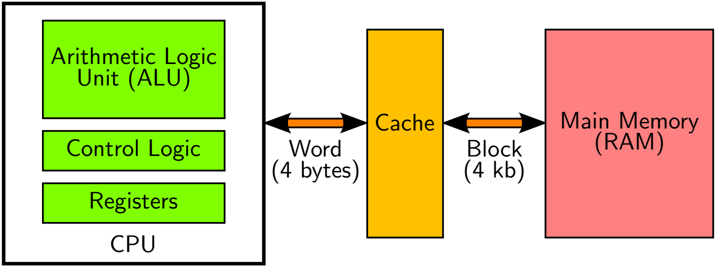
Data Loads and Stores:

- Stack Access: Every `PUSH` will have a corresponding `POP`, and the more recent the `PUSH` operation, the sooner the `POP` will occur.
- Local Variables and Operands: Algorithms which need large numbers of local variables (beyond what can be held in registers) tend to be data-intensive.

Instruction Loads:

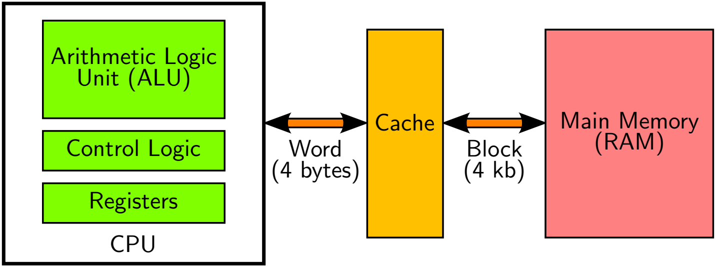
- Loops: The same instructions must be loaded and executed repeatedly during a loop.
- Functions: Since functions are often used to reduce code duplication, it is likely that a function could be executed frequently.

Cache Mechanics (1)



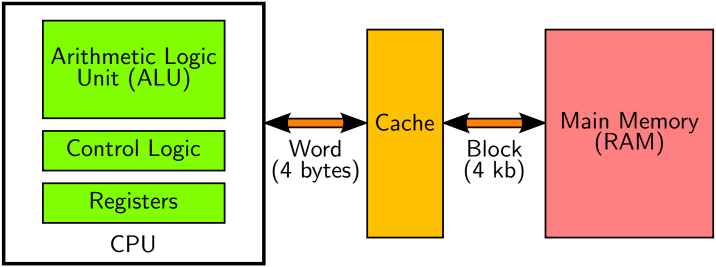
A cache is normally divided into a set of **cache lines**. Each cache line stores a contiguous block of memory.

Cache Mechanics (2)



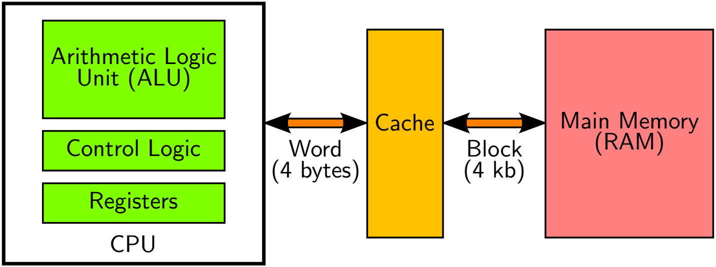
Since each request to main memory is slow, reading or writing entire blocks of memory at a time tends to be faster than reading or writing individual words due to decreased latency.

Cache Mechanics (3)



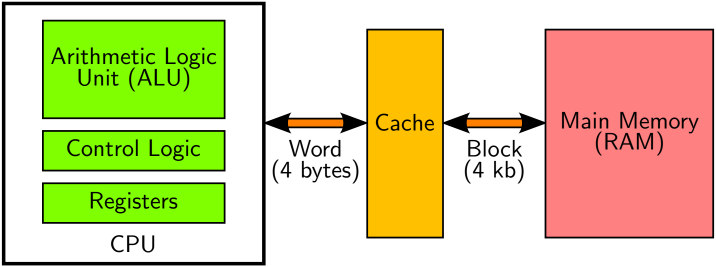
On the other hand, the CPU tends to access memory in smaller chunks, usually one word. For example, in the AVR architecture, memory loads are either 16 bits (loading from program memory) or 8 bits (loading from data memory).

Cache Mechanics (4)



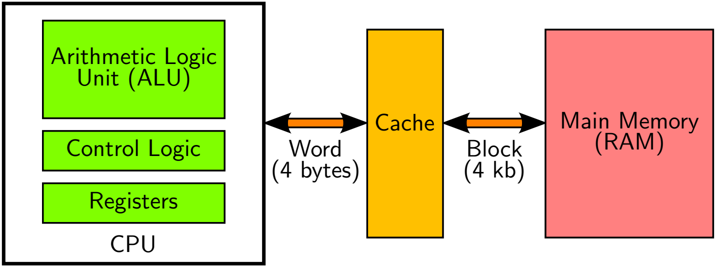
We normally conceptualize the cache as 'sitting between' the CPU and memory, such that all memory requests must go through the cache's processing circuitry.

Cache Mechanics (5)



A major problem which arises with any cache system (including caching mechanisms used in software or over networks) is the issue of **invalidation**. Since the cache, by its nature, stores duplicate copies of data, it is necessary to carefully track when both copies of the data are changed, to ensure that they do not disagree.

Cache Mechanics (6)

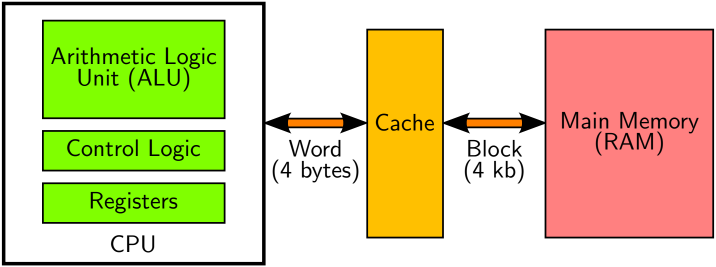


When caching is used to speed up algorithms (for example, by caching recent search results in a dictionary structure), managing the invalidation problem is crucial to ensure that the data is consistent.

“ There are only two hard problems in Computer Science: cache invalidation and naming things. ”

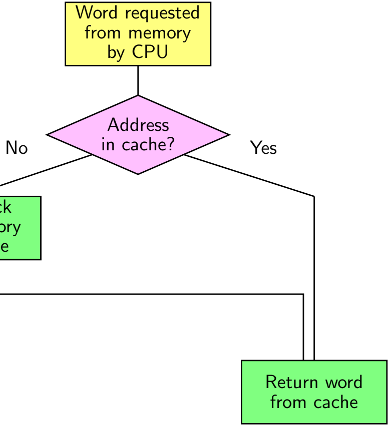
- P. Karlton

Cache Mechanics (7)



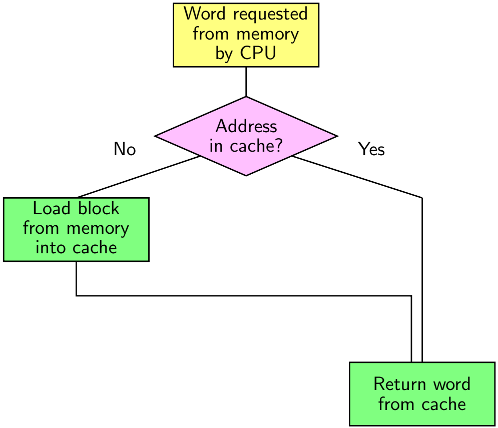
In the model used here, with the cache interposed between the CPU and memory, invalidation is less of a concern as long as only one CPU is present, but we still need to define access policies to govern how data is moved between the cache and memory.

Cache Mechanics (8)



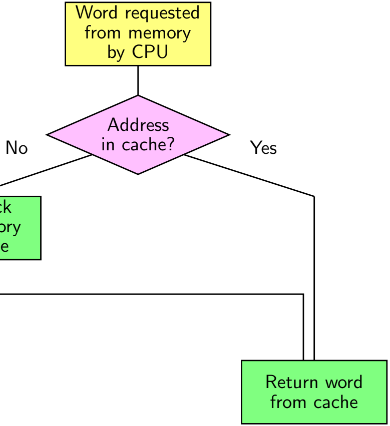
The flow chart above shows a possible process for handing a read request (like a load instruction or the 'fetch' phase of the instruction pipeline). If the requested data is in the cache, it is immediately returned to the CPU.

Cache Mechanics (9)



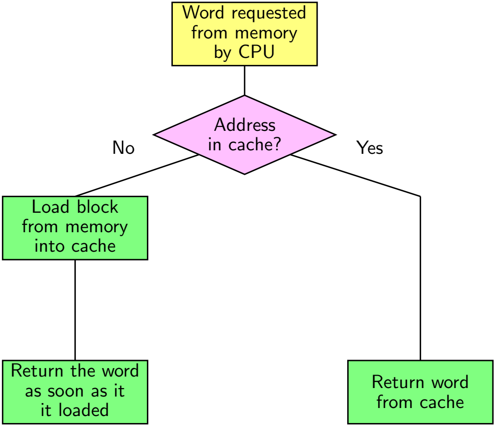
If the requested data is not in the cache, the block containing the address is loaded into the cache, and the word is then returned to the CPU.

Cache Mechanics (10)



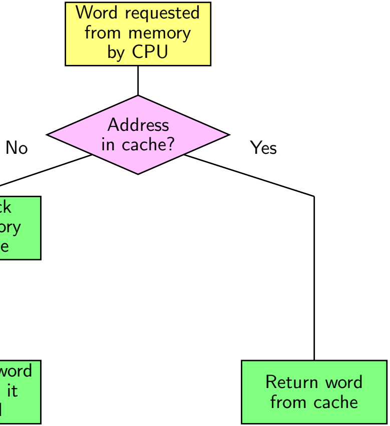
This policy can be called **load and forward**.

Cache Mechanics (11)



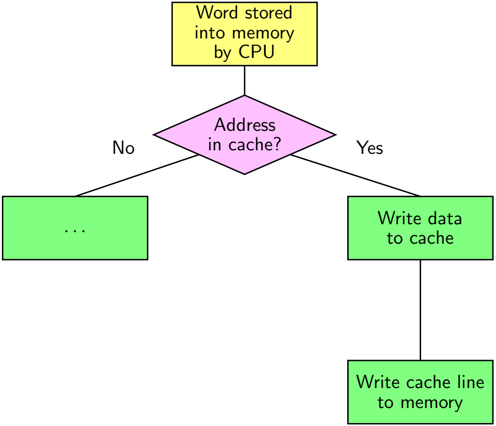
An alternative is **load through**, which returns the requested word as soon as it arrives from main memory (which may occur before the cache line has been completely filled). The CPU can then continue processing while the rest of the cache line is filled in the background.

Cache Mechanics (12)



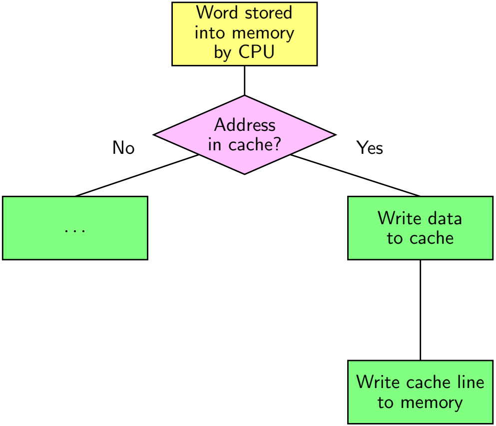
Question: Assuming that blocks are received from memory in order (that is, low addresses first), which words in the block would benefit least (or most) from the load through policy?

Cache Mechanics (13)



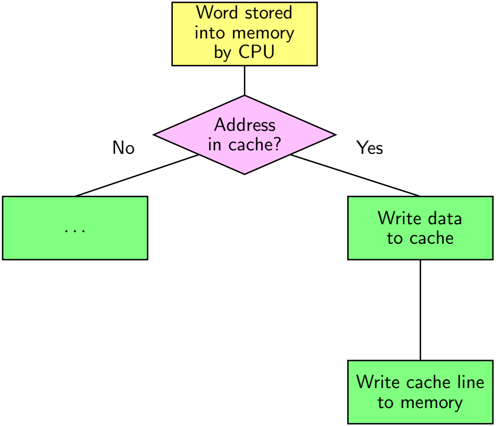
For store requests, similar policies can be defined. The policies for cases where the data is in the cache (**cache hits**) are independent from those where the data is not in the cache (**cache misses**).

Cache Mechanics (14)



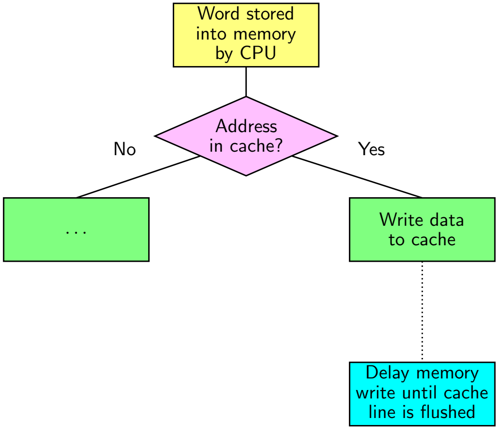
Under the **write through** (WT) policy, if a write request is received for data which is in the cache, the cache is updated and an update is also dispatched to main memory.

Cache Mechanics (15)



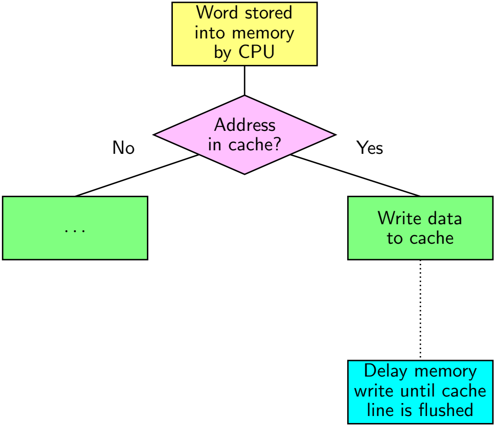
With a write through policy, the CPU may have to wait for the memory write to complete before continuing, but not always. In general, write through is assumed to be slow because of this issue.

Cache Mechanics (16)



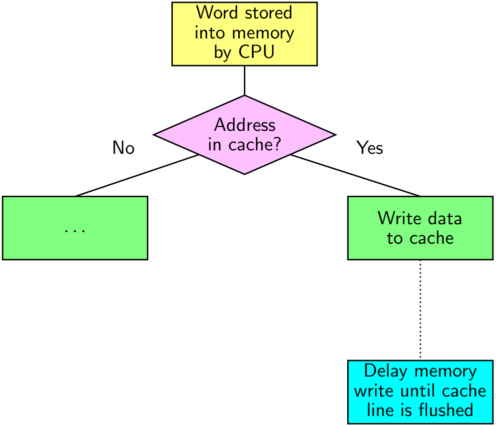
Alternatively, under the **write back** (WB) policy, only the cached copy is updated. Later, when the cache line containing the modified data is flushed from the cache, the entire line can be written back to memory.

Cache Mechanics (17)



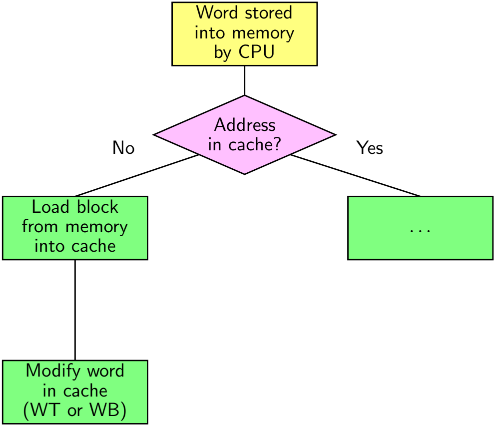
Under the write back policy, the data in memory may differ from the memory in the cache, but the CPU will never see the difference (since all requests for the modified data will be cache hits).

Cache Mechanics (18)



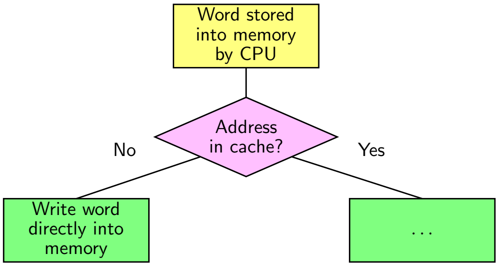
Question: How could the write back policy cause problems in a machine with two processors (each with their own cache) and one shared memory?

Cache Mechanics (19)



For write requests which result in cache misses, the **write allocate** policy is to load the relevant block into the cache, then treat the write as if it was a cache hit (with either write through or write back).

Cache Mechanics (20)



The **write no-allocate** policy just dispatches the write request to memory, without bringing the data into the cache.

Cache Addressing (1)

00000		01000		10000	
00001		01001		10001	
00010		01010		10010	
00011		01011		10011	
00100		01100		10100	
00101		01101		10101	
00110		01110		10110	
00111		01111		10111	

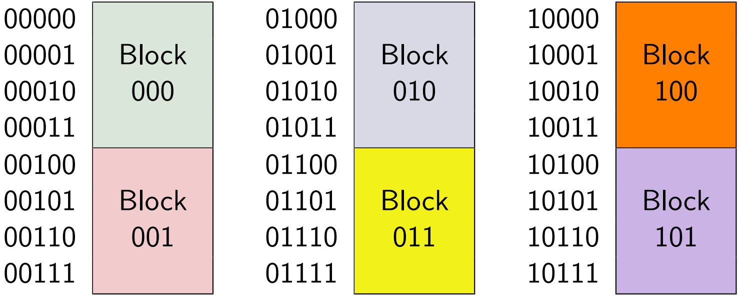
Consider the memory above, with the binary address of each memory location shown. Suppose that each location stores 8 bits.

Cache Addressing (2)

00000		01000		10000	
00001		01001		10001	
00010		01010		10010	
00011		01011		10011	
00100		01100		10100	
00101		01101		10101	
00110		01110		10110	
00111		01111		10111	

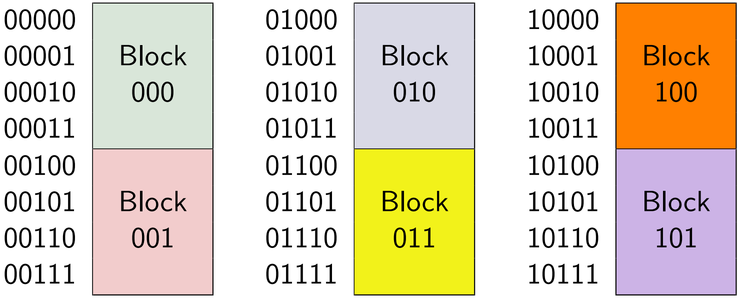
Question: Suppose that a cache structure which stores 32-bit (4 byte) blocks is used with this memory. How should the memory be divided into blocks, and how should the blocks be identified?

Cache Addressing (3)



The most convenient option is to split the 5-bit address into two pieces: The most significant 3 bits become a ‘tag’ to label each block, and the least significant two bits are used to distinguish the different slots inside the block

Cache Addressing (4)



For example, if address 10110 was read, the cache would be checked for block 101, and if the block was found, the byte at index 10 would be the desired value.

Associative Mapping (1)

Tag (10 bits)	Index (6 bits)

Consider a system which has byte-addressable memory with 16 bit memory addresses and suppose a cache is designed with a block size of $64 = 2^6$ bytes. In this system, each block would be identified by a 10 bit tag.

Associative Mapping (2)

	V	**D**	Tag (10 bits)	Data (512 bits)
Line 0				
Line 1				
Line 2				
Line 3				
Line 4				
...				...
Line 15				

Now consider a simple cache which maintains a set of up to 16 blocks (for a total of $16 \cdot 64 = 1024$ bytes $=$ 1kb of storage). Each block is identified by its 10 bit tag.

Associative Mapping (3)

	V	**D**	Tag (10 bits)	Data (512 bits)
Line 0				
Line 1				
Line 2				
Line 3				
Line 4				
...			...	
Line 15				

When the CPU reads or writes a particular address, the set of blocks in the cache is searched, and if a block matching the address is found, the copy in the cache is used instead of going to memory.

Associative Mapping (4)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	
Line 4	0			
...				...
Line 15	1	0	0000011000	

A **valid bit** (V) is used to indicate which cache lines contain valid blocks. Cache lines can be invalidated for a variety of reasons, so the cache may not always be full. For example, when operating systems switch between running programs, the entire cache might be invalidated.

Associative Mapping (5)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	
Line 4	0			
. . .			. . .	
Line 15	1	0	0000011000	

This model is called an **associative mapped cache**, since it is conceptually similar to an associative array which maps tags to blocks. Note that there is no internal ordering to the blocks in the cache. To find a particular block, every cache line may need to be inspected.

Associative Mapping (6)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	
Line 4	0			
...			...	
Line 15	1	0	0000011000	

For example, if the CPU requests a load from memory address `0xe582` $= (1110010110000010)_2$, the leading 10 bits (1110010110) would be used as the tag. When the cache is searched, the cache line shaded in orange would be found.

Associative Mapping (7)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	
Line 4	0			
...			...	
Line 15	1	0	0000011000	

The lower 6 bits of the address (000010) would be used to index into the data block stored in the cache line, and the byte at that location would be returned to the CPU.

Associative Mapping (8)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	
Line 4	0			
...				...
Line 15	1	0	0000011000	

Searching through the set of tags is done by the hardware in parallel (not with a linear search algorithm). Circuitry for this task becomes increasingly complicated and unwieldy as the number of cache lines increases.

Associative Mapping (9)

	V	**D**	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	
Line 4	0			
...				...
Line 15	1	0	0000011000	

If the cache is used with a **write through** policy for memory writes, then any write will modify the corresponding entry in the cache, as well as the actual location in memory (although the memory write will occur in the background and not delay the CPU's execution).

Associative Mapping (10)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	
Line 4	0			
...			...	
Line 15	1	0	0000011000	

If the cache is used with a **write back** policy for memory writes, then if the written value is in an active cache line, it **will not** be written back to memory until the cache line is invalidated. Instead, the cache line is updated with the changed value and marked as **dirty**.

Associative Mapping (11)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	
Line 4	0			
. . .				. . .
Line 15	1	0	0000011000	

The **dirty bit** (D) denotes cache lines which have been modified in the cache (but not in main memory). When a dirty cache line is invalidated, the data in the cache line must be written back to memory (no such requirement exists when the D bit is not set).

Associative Mapping (12)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	0xe1, 0xe6, 0xa6, 0xbb, ...
Line 4	0			
...			...	
Line 15	1	0	0000011000	

Example: Assuming the block in line 3 has the contents shown, write the value 0x06 to address 0xe582.

Associative Mapping (13)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	0	1110010110	0xe1, 0xe6, 0xa6, 0xbb, ...
Line 4	0			
...				...
Line 15	1	0	0000011000	

As before, the address 0xe582 would be split into a 10 bit tag $(1110010110)_2$ and a 6 bit index $(000010)_2$ and the cache line in orange would be found.

Associative Mapping (14)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	0			
Line 3	1	**1**	1110010110	0xe1, 0xe6, 0x06, 0xbb, ...
Line 4	0			
...				...
Line 15	1	0	0000011000	

The index $(000010)_2 = (2)_{10}$ would be used to identify the byte to modify and the byte would be changed to 0x06. The dirty bit would be set on the cache line to indicate the modification.

Associative Mapping (15)

	V	**D**	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	1	0	0101001100	
Line 3	1	0	1110010110	
Line 4	1	0	1100111110	
...			...	
Line 15	1	0	0000011000	

If the cache has any entries which are invalid, then new blocks can be loaded into those cache lines without any repercussions.

Associative Mapping (16)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	1	0	0101001100	
Line 3	1	0	1110010110	
Line 4	1	0	1100111110	
...				...
Line 15	1	0	0000011000	

Question: What if all 16 cache lines are full?

Associative Mapping (17)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	1	0	0101001100	
Line 3	1	0	1110010110	
Line 4	1	0	1100111110	
...				...
Line 15	1	0	0000011000	

When the cache is full and a new block is loaded, a **replacement policy** is used to choose a cache line to invalidate so that the new block can be accommodated. When a cache line is invalidated, it will be written back to memory if its dirty bit is set (otherwise, it will just be discarded).

Associative Mapping (18)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	1	0	0101001100	
Line 3	1	0	1110010110	
Line 4	1	0	1100111110	
...			...	
Line 15	1	0	0000011000	

Since an associative mapped cache has no restriction on which cache line a particular block can occupy, any of the 16 cache lines is a candidate for removal. Various replacement policies can be used to optimize performance.

Associative Mapping (19)

	V	**D**	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	1	0	0101001100	
Line 3	1	0	1110010110	
Line 4	1	0	1100111110	
...			...	
Line 15	1	0	0000011000	

In theory, an optimal replacement policy will invalidate the cache line which will be least useful in the future. However, it is impossible to predict this.

Associative Mapping (20)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	1	0	0101001100	
Line 3	1	0	1110010110	
Line 4	1	0	1100111110	
...				...
Line 15	1	0	0000011000	

Some simple policies are to invalidate the **least frequently used** (LFU) or **least recently used** (LRU) blocks. Alternatively, the cache can be treated as a queue, and first-in first-out order can be used to discard a block (that is, the **oldest** block is discarded).

Associative Mapping (21)

	V	D	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	1	0	0101001100	
Line 3	1	0	1110010110	
Line 4	1	0	1100111110	
...				...
Line 15	1	0	0000011000	

Finally, a randomly selected block can be discarded in the absence a more clever replacement policy.

Associative Mapping (22)

	V	**D**	Tag (10 bits)	Data (512 bits)
Line 0	1	0	1011000110	
Line 1	1	0	1111110011	
Line 2	1	0	0101001100	
Line 3	1	0	1110010110	
Line 4	1	0	1100111110	
...			...	
Line 15	1	0	0000011000	

Generally, the least recently used (LRU) policy is a reasonably good compromise between practical and efficient.

Associative Mapping (23)

Associative caches have the advantage of allowing instantaneous lookup of a tag, allowing blocks to occupy any cache line and allowing the use of a tuned replacement algorithm to improve performance.

However, the circuitry required for associative caches is too complicated to be practical at the scales needed for modern processors.

Associative Mapping Example (1)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	0			
Line 1	0			
Line 2	0			
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Consider an associative mapped cache with 4 cache lines and 4 byte (32 bit) blocks, with a main memory which has 8 bit addresses and 8 bit memory locations.

Associative Mapping Example (2)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	0			
Line 1	0			
Line 2	0			
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

The table on the right is a compact representation of the first 32 memory addresses. For example, the highlighed cell is the byte at address 0x15.

Associative Mapping Example (3)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	0			
Line 1	0			
Line 2	0			
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
`0x00`	`00`	`ab`	`06`	`01`
`0x04`	`04`	`08`	`15`	`16`
`0x08`	`c5`	`c2`	`30`	`af`
`0x0C`	`de`	`ad`	`be`	`ef`
`0x10`	`23`	`42`	`20`	`06`
`0x14`	`a5`	`df`	`a5`	`df`
`0x18`	`02`	`30`	`02`	`25`
`0x1C`	`06`	`10`	`bb`	`17`

...

Exercise: Using the **write allocate** and **write back** policies, along with the **least recently used** replacement policy, draw the state of the cache and memory after the following operations.

1. Load from `0x01`
2. Store `0xFF` into `0x02`
3. Store `0x99` into `0x08`
4. Load from `0x05`
5. Store `0xAA` into `0x15`
6. Load from `0x13`

Associative Mapping Example (4)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	0			
Line 1	0			
Line 2	0			
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

(The choice of load and forward vs. load through is not relevant to this example, since the state of the cache will be the same under either policy. In general, if you are asked to do an example like this, the load policy will not be relevant.)

Associative Mapping Example (5)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	0	000000	00 ab 06 01
Line 1	0			
Line 2	0			
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 1 (Load from `0x01`): Address `0x01` $= (000000\ 01)_2$ lies in block `0x00`, which is not cached. The block is loaded into an available cache line (in this case, line 0) with the tag 000000. The value at index 01 is then read and returned to the CPU.

Associative Mapping Example (6)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	1	000000	00 ab ff 01
Line 1	0			
Line 2	0			
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 2 (Store 0xFF into 0x02): Address 0x02 = $(000000\ 10)_2$ lies in block 0x00, which is in the cache. The value at index 10 is updated to 0xFF in the cache and the dirty bit is set to indicate that the line has been modified.

Associative Mapping Example (7)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	1	000000	00 ab ff 01
Line 1	0			
Line 2	0			
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

No memory write is performed due to the write back policy (the entire block will be written when it is removed from the cache). This step therefore requires no memory reads or writes at all.

Associative Mapping Example (8)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	1	000000	00 ab ff 01
Line 1	1	1	000010	99 c2 30 af
Line 2	0			
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 3 (Store `0x99` into `0x08`): Address `0x08` = $(000010\ 00)_2$ lies in block `0x08`, which is not cached. By the write allocate policy, it must be loaded into the cache, and by the write back policy, no memory write is performed. The value at index 00 is updated to `0x99` after the line is loaded and the dirty bit is immediately set on the loaded line.

Associative Mapping Example (9)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	1	000000	00 ab ff 01
Line 1	1	1	000010	99 c2 30 af
Line 2	1	0	000001	04 08 15 16
Line 3	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 4 (Load from 0x05): Address 0x05 = $(000001\ 01)_2$ lies in block 0x04, which is not cached. The block is loaded into an empty slot and the value at index 01 is returned to the CPU.

Associative Mapping Example (10)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	1	000000	00 ab ff 01
Line 1	1	1	000010	99 c2 30 af
Line 2	1	0	000001	04 08 15 16
Line 3	1	1	000101	a5 aa a5 df

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 5 (Store `0xAA` into `0x15`): Address `0x15` $= (000101\ 01)_2$ lies in block `0x14`, which is not cached. The block is loaded into the last available line and the value at index 01 is modified. The dirty bit is set on the loaded line. This step involves a memory load (to load the data into the cache) but no memory writes.

Associative Mapping Example (11)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	1	000000	00 ab ff 01
Line 1	1	1	000010	99 c2 30 af
Line 2	1	0	000001	04 08 15 16
Line 3	1	1	000101	a5 aa a5 df

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 6 (Load from `0x13`): Address `0x13` $= (000100\ 11)_2$ lies in block `0x10`, which is not cached. In order to load the block which starts at `0x10`, one of the existing lines must be ejected from the cache.

Associative Mapping Example (12)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	1	000000	00 ab ff 01
Line 1	1	1	000010	99 c2 30 af
Line 2	1	0	000001	04 08 15 16
Line 3	1	1	000101	a5 aa a5 df

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 6 (continued): Since the cache is using a **least recently used** replacement policy, the line which has been used least recently (shown in orange) is selected for removal.

Associative Mapping Example (13)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	1	000000	00 ab ff 01
Line 1	1	1	000010	99 c2 30 af
Line 2	1	0	000001	04 08 15 16
Line 3	1	1	000101	a5 aa a5 df

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	ff	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 6 (continued): The selected cache line has the D bit set, which means that the (entire) line must be written back to memory before deleting it from the cache. Remember that this step is **only** necessary if the D bit is set.

Associative Mapping Example (14)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	0	000100	23 42 20 06
Line 1	1	1	000010	99 c2 30 af
Line 2	1	0	000001	04 08 15 16
Line 3	1	1	000101	a5 aa a5 df

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	ff	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 6 (continued): Once the line is written back to memory, block 0x10 can be read into the freed line and the value at index 11 can be returned to the CPU.

Associative Mapping Example (15)

Cache

	V	D	Tag (6 bits)	Data (32 bits)
Line 0	1	0	000100	23 42 20 06
Line 1	1	1	000010	99 c2 30 af
Line 2	1	0	000001	04 08 15 16
Line 3	1	1	000101	a5 aa a5 df

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	ff	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Question: How many memory reads and writes were needed for the entire process? How does this compare to the number which would be required without a cache? You may count a block read/write as one operation.

Associative Mapping Example (16)

Exercise: With the same initial memory values as the previous example (and an initially empty cache), show the state of the cache and memory after the following sequence of operations. Use the **write allocate** and **write back** policies with a least recently used (LRU) replacement policy.

1. Load from `0x01`
2. Store `0xFF` into `0x02`
3. Store `0x99` into `0x08`
4. Store `0xCC` into `0x00`
5. Store `0x00` into `0x0A`
6. Load from `0x05`
7. Store `0xAA` into `0x15`
8. Load from `0x13`
9. Load from `0x1e`
10. Load from `0x02`

Associative Mapping Example (17)

After completing the exercise on the previous slide, consider the following questions.

- How many memory reads/writes were needed to complete the 10 operations?
- How many memory reads/writes would be needed if the write no-allocate policy was used instead of the write allocate policy?
- How many memory reads/writes would be needed if the **least frequently used** (LFU) replacement policy was used instead of LRU?

Direct Mapping (1)

Tag (7 bits)	Slot (3 bits)	Index (6 bits)

An alternative is **direct mapping**, which assigns each block of memory to a particular cache line. Using the same addressing scheme as before (16 bit addresses, 8 bit memory locations), consider a cache which uses 64 byte blocks and has 8 cache lines, numbered 0 through 7.

Direct Mapping (2)

Tag (7 bits)	Slot (3 bits)	Index (6 bits)

The 10 bit block index is now divided into a 7 bit tag and a 3 bit slot index, which determines the cache line that each block will occupy.

Direct Mapping (3)

Tag (7 bits)	Slot (3 bits)	Index (6 bits)

Normally, the slot index is taken to be the low order bits of the 10 bit tag. This ensures that the set of blocks with the same index will be relatively far apart in memory.

Direct Mapping (4)

	V	**D**	Tag (7 bits)	Data (512 bits)
Line 000				
Line 001				
Line 010				
Line 011				
Line 100				
Line 101				
Line 110				
Line 111				

The cache is otherwise laid out congruently to an associative mapped cache, with V and D bits.

Direct Mapping (5)

	V	D	Tag (7 bits)	Data (512 bits)
Line 000	0			
Line 001	0			
Line 010	0			
Line 011	1	0	1010111	`0xe1, 0xe6, 0xa6, 0xbb, ...`
Line 100	0			
Line 101	0			
Line 110	1	0	1110011	`0x01, 0x06, 0x01, 0x16, ...`
Line 111	0			

The actual address of a block can be recovered by combining the 7 bit tag with the slot number. For example, the 64 byte block in slot 011 occupies addresses $(1010111011000000)_2$ - $(1010111011111111)_2$.

Direct Mapping (6)

	V	D	Tag (7 bits)	Data (512 bits)
Line 000	0			
Line 001	0			
Line 010	0			
Line 011	1	0	1010111	`0xe1, 0xe6, 0xa6, 0xbb, ...`
Line 100	0			
Line 101	0			
Line 110	1	0	1110011	`0x01, 0x06, 0x01, 0x16, ...`
Line 111	0			

In a direct mapped cache, there is only one possible replacement policy: A block is invalidated and removed from the cache when a new block with the same slot number is loaded. Since each block has a predetermined slot, it **must** be placed in that slot when loaded.

Direct Mapping Example (1)

Cache

	V	**D**	Tag (4 bits)	Data (32 bits)
Line 00	0			
Line 01	0			
Line 10	0			
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Consider a direct mapped cache with 4 cache lines and 4 byte (32 bit) blocks, with a main memory which has 8 bit addresses and 8 bit memory locations.

Direct Mapping Example (2)

Cache

	V	**D**	Tag (4 bits)	Data (32 bits)
Line 00	0			
Line 01	0			
Line 10	0			
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Notice that in this case, the tag for each cache line is only 4 bits long, since each block can be identified by a 6 bit tag, which is split into a 2 bit slot number (formed from the lower 2 bits) and a 4 bit cache line tag.

Direct Mapping Example (3)

Cache

	V	**D**	Tag (4 bits)	Data (32 bits)
Line 00	0			
Line 01	0			
Line 10	0			
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Exercise: Using the **write allocate** and **write back** policies, draw the state of the cache and memory after the following operations.

1. Load from `0x01`
2. Store `0xFF` into `0x02`
3. Store `0x99` into `0x08`
4. Load from `0x05`
5. Store `0xAA` into `0x15`
6. Load from `0x13`

Direct Mapping Example (4)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	0	0000	00 ab 06 01
Line 01	0			
Line 10	0			
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 1 (Load from `0x01`**)**: Address `0x01` $= (0000\ 00\ 01)_2$ lies in block `0x00`, which is not cached. The block has slot index 00, so it must be inserted into cache line 00. The value at index 01 of the block is then returned to the CPU.

Direct Mapping Example (5)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	1	0000	00 ab ff 01
Line 01	0			
Line 10	0			
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 2 (Store `0xFF` into `0x02`): Address `0x02` = $(0000\ 00\ 10)_2$ lies in block `0x00`, which is cached. As in the earlier example, due to the write back policy, the modification is only made in the cache line and the dirty bit is set.

Direct Mapping Example (6)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	1	0000	00 ab ff 01
Line 01	0			
Line 10	1	1	0000	99 c2 30 af
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 3 (Store 0x99 **into** 0x08**)**: Address 0x08 = $(0000\ 10\ 00)_2$ lies in block 0x08, which is not cached. Due to the write allocate policy, the block must be loaded before the write is performed. The block is loaded into line 10, as required by its slot index. The newly loaded line is then modified and its dirty bit is set.

Direct Mapping Example (7)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	1	0000	00 ab ff 01
Line 01	1	0	0000	04 08 15 16
Line 10	1	1	0000	99 c2 30 af
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 4 (Load from `0x05`**)**: Address `0x05` $= (0000\ 01\ 01)_2$ lies in block `0x04`, which is not cached. The block is loaded into slot 01 and the value at index 01 is returned to the CPU.

Direct Mapping Example (8)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	1	0000	00 ab ff 01
Line 01	1	0	0000	04 08 15 16
Line 10	1	1	0000	99 c2 30 af
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 5 (Store `0xAA` into `0x15`): Address `0x15` $= (0001\ 01\ 01)_2$ lies in block `0x14`, which is not cached. By the write allocate policy, the block must be loaded. However, the block is assigned to index 01, which is full. Therefore, the current contents of line 01 must be removed from the cache. Since the dirty bit is not set on line 01, it is deleted immediately.

Direct Mapping Example (9)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	1	0000	00 ab ff 01
Line 01	1	1	0001	a5 aa a5 df
Line 10	1	1	0000	99 c2 30 af
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 5 (continued): Block `0x14` is then loaded into slot 01. To write the value `0xAA` to address `0x15`, the value at index 01 of the block is modified and the dirty bit of line 01 is set.

Direct Mapping Example (10)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	1	0000	00 ab ff 01
Line 01	1	1	0001	a5 aa a5 df
Line 10	1	1	0000	99 c2 30 af
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 6 (Load from `0x13`): Address `0x13` = $(0001\ 00\ 11)_2$ lies in block `0x10`, which is not cached. Block `0x10` must be loaded into slot 00, which is occupied and therefore must be flushed.

Direct Mapping Example (11)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	1	0000	00 ab ff 01
Line 01	1	1	0001	a5 aa a5 df
Line 10	1	1	0000	99 c2 30 af
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	ff	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 6 (continued): Since the dirty bit is set on line 00, the entire line is written to memory.

Direct Mapping Example (12)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Line 00	1	0	0001	23 42 20 06
Line 01	1	1	0001	a5 aa a5 df
Line 10	1	1	0000	99 c2 30 af
Line 11	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	ff	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 6 (continued): After the line is written to memory, block 0x10 is loaded into slot 01 and the value at index 11 is returned to the CPU

Set-Associative Mapping (1)

Associative mapping is flexible and efficient (in terms of memory accesses), but costly, and does not scale to the number of cache lines needed by modern caches. Direct mapping is relatively inexpensive (assigning blocks to cache lines only requires simple switching circuitry) and can scale easily, but is inflexible and can yield poor behavior if certain sets of blocks are used frequently.

A compromise is the **set-associative** mapping scheme, which uses a direct mapping approach to assign each block to a slot in the cache, but allows each slot to contain multiple cache lines, which are administered with an associative model. Each slot is assigned a fixed-size set of k cache lines and the cache is normally called *k-way set associative*. For example, an 8-way set associative cache would have 8 cache lines per slot.

Set-Associative Mapping (2)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Slot 00	0			
	0			
Slot 01	0			
	0			
Slot 10	0			
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Consider the system above, with the same properties as the previous examples (8 bit memory addresses and 8 bit memory cells), but with a 2-way set associative cache with 4 slots (giving a total of 8 cache lines).

Set-Associative Mapping (3)

Cache

	V	**D**	Tag (4 bits)	Data (32 bits)
Slot 00	0			
	0			
Slot 01	0			
	0			
Slot 10	0			
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Exercise: Using the **write allocate** and **write back** policies, along with the **least recently used** replacement policy, draw the state of the cache and memory after the following operations.

1. Load from 0x01
2. Store 0xFF into 0x02
3. Store 0x99 into 0x08
4. Load from 0x05
5. Store 0xAA into 0x15
6. Load from 0x13

Set-Associative Mapping (4)

Cache

	V	**D**	Tag (4 bits)	Data (32 bits)
Slot 00	0			
	0			
Slot 01	0			
	0			
Slot 10	0			
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Note that the LRU policy is specified since, within the set of lines contained in a particular slot, the scheme is associative and therefore requires a replacement policy. However, for this example, the policy is never needed.

Set-Associative Mapping (5)

Cache

	V	**D**	Tag (4 bits)	Data (32 bits)
Slot 00	0			
	0			
Slot 01	0			
	0			
Slot 10	0			
	0			
Slot 11	0			
	0			

Memory

Block Start	$+0$	$+1$	$+2$	$+3$
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Also, note that this cache is actually larger than the one used by previous example (8 lines instead of 4), so the 'better' performance on this example is not necessarily due to the set-associative scheme.

Set-Associative Mapping (6)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Slot 00	1	0	`0000`	`00 ab 06 01`
	0			
Slot 01	0			
	0			
Slot 10	0			
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
`0x00`	`00`	`ab`	`06`	`01`
`0x04`	`04`	`08`	`15`	`16`
`0x08`	`c5`	`c2`	`30`	`af`
`0x0C`	`de`	`ad`	`be`	`ef`
`0x10`	`23`	`42`	`20`	`06`
`0x14`	`a5`	`df`	`a5`	`df`
`0x18`	`02`	`30`	`02`	`25`
`0x1C`	`06`	`10`	`bb`	`17`

...

Step 1 (Load from `0x01`): Address `0x01` $= (0000\ 00\ 01)_2$ lies in block `0x00`, which is not cached. The block has slot index 00, so it must be inserted into cache line 00. The value at index 01 of the block is then returned to the CPU.

Set-Associative Mapping (7)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Slot 00	1	1	0000	00 ab ff 01
	0			
Slot 01	0			
	0			
Slot 10	0			
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 2 (Store `0xFF` **into** `0x02`**)**: Address `0x02` $= (0000\ 00\ 10)_2$ lies in block `0x00`, which is cached. As in the earlier example, due to the write back policy, the modification is only made in the cache line and the dirty bit is set.

Set-Associative Mapping (8)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Slot 00	1	1	0000	00 ab ff 01
	0			
Slot 01	0			
	0			
Slot 10	1	1	0000	99 c2 30 af
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 3 (Store 0x99 **into** 0x08**)**: Address 0x08 = $(0000\ 10\ 00)_2$ lies in block 0x08, which is not cached. The block is loaded into slot 10 and the write is performed to the cached copy only.

Set-Associative Mapping (9)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Slot 00	1	1	0000	00 ab ff 01
	0			
Slot 01	1	0	0000	04 08 15 16
	0			
Slot 10	1	1	0000	99 c2 30 af
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 4 (Load from 0x05): Address 0x05 = $(0000\ 01\ 01)_2$ lies in block 0x04, which is not cached. The block is loaded into slot 01 and the value at index 01 is returned to the CPU.

Set-Associative Mapping (10)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Slot 00	1	1	0000	00 ab ff 01
	0			
Slot 01	1	0	0000	04 08 15 16
	1	1	0001	a5 aa a5 df
Slot 10	1	1	0000	99 c2 30 af
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 5 (Store 0xAA into 0x15): Address 0x15 = $(0001\ 01\ 01)_2$ lies in block 0x14, which is not cached. Since one of the lines in slot 01 is still available, the block can be loaded into line 01. The value of address 0x15 is then changed and the dirty bit is set.

Set-Associative Mapping (11)

Cache

	V	D	Tag (4 bits)	Data (32 bits)
Slot 00	1	1	0000	00 ab ff 01
	1	0	0001	23 42 20 06
Slot 01	1	0	0000	04 08 15 16
	1	1	0001	a5 aa a5 df
Slot 10	1	1	0000	99 c2 30 af
	0			
Slot 11	0			
	0			

Memory

Block Start	+0	+1	+2	+3
0x00	00	ab	06	01
0x04	04	08	15	16
0x08	c5	c2	30	af
0x0C	de	ad	be	ef
0x10	23	42	20	06
0x14	a5	df	a5	df
0x18	02	30	02	25
0x1C	06	10	bb	17

...

Step 6 (Load from `0x13`): Address `0x13` $= (0001\ 00\ 11)_2$ lies in block `0x10`, which is not cached. Since one of the lines in slot 00 is still available, the block is loaded into line 00 and the value at index 11 is returned to the CPU.

Sources

- Slides by B. Bird, 2017 - 2018.
- The diagrams on Slides 29 - 41 are based on Figure 7-25 of *Computer Architecture and Organization* by Miles Murdocca and Vincent Heuring (Wiley, 2007).