Table of contents

Introduction

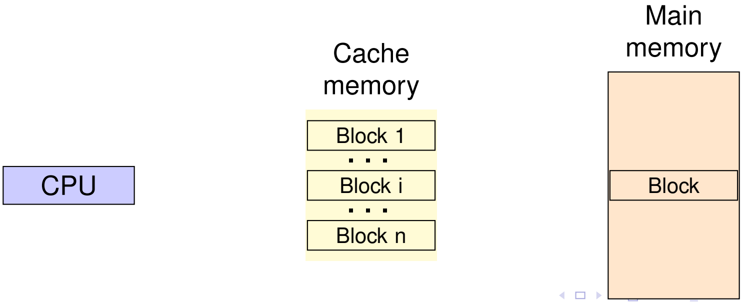
Cache

Name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor

- SRAM technology $\rightarrow$ fast, high cost per bit, low capacity

Divided into blocks

- contain data from consecutive memory locations
- all blocks have the same capacity



Introduction

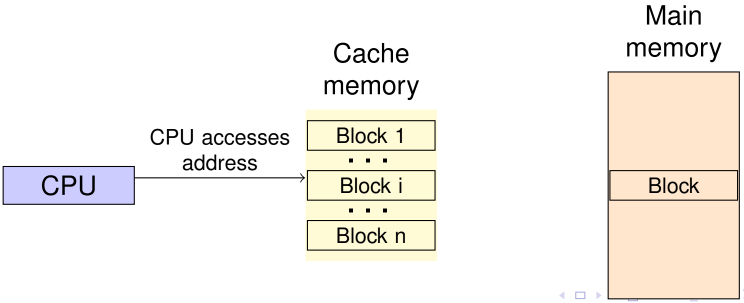
Cache

Name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor

- SRAM technology $\rightarrow$ fast, high cost per bit, low capacity

Divided into blocks

- contain data from consecutive memory locations
- all blocks have the same capacity



Introduction

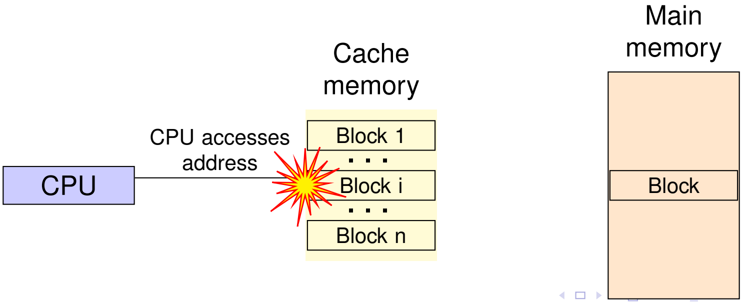
Cache

Name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor

- SRAM technology $\rightarrow$ fast, high cost per bit, low capacity

Divided into blocks

- contain data from consecutive memory locations
- all blocks have the same capacity



Introduction

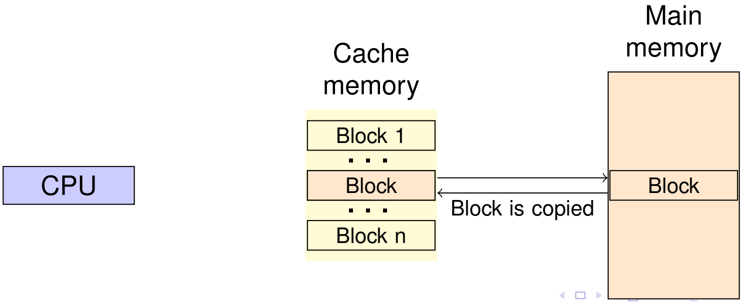
Cache

Name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor

- SRAM technology $\rightarrow$ fast, high cost per bit, low capacity

Divided into blocks

- contain data from consecutive memory locations
- all blocks have the same capacity



Introduction

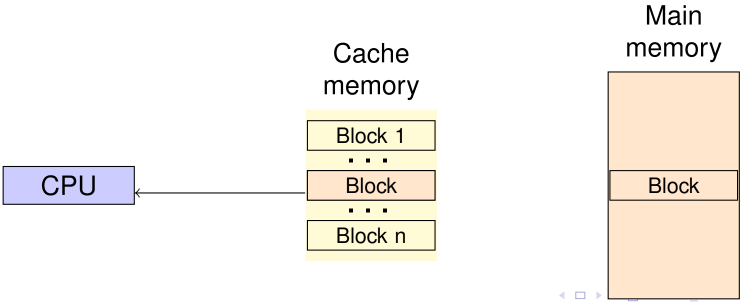
Cache

Name given to the highest or first level of the memory hierarchy encountered once the address leaves the processor

- SRAM technology $\rightarrow$ fast, high cost per bit, low capacity

Divided into blocks

- contain data from consecutive memory locations
- all blocks have the same capacity



Introduction

Operation

- When the processor finds a requested data item in the cache, it is called a cache hit
- When the processor does not find a data item in the cache, a cache miss occurs
- A fixed-size collection of data, block, is retrieved from the main memory and placed into the cache
- The principle of locality expresses that data contained in the block is likely to be accessed in the near future

Operation

- When the processor finds a requested data item in the cache, it is called a cache hit
- When the processor does not find a data item in the cache, a cache miss occurs
- A fixed-size collection of data, block, is retrieved from the main memory and placed into the cache
- The principle of locality expresses that data contained in the block is likely to be accessed in the near future

Operation

- When the processor finds a requested data item in the cache, it is called a cache hit
- When the processor does not find a data item in the cache, a cache miss occurs
- A fixed-size collection of data, block, is retrieved from the main memory and placed into the cache
- The principle of locality expresses that data contained in the block is likely to be accessed in the near future

Introduction

Operation

- When the processor finds a requested data item in the cache, it is called a cache hit
- When the processor does not find a data item in the cache, a cache miss occurs
- A fixed-size collection of data, block, is retrieved from the main memory and placed into the cache
- The principle of locality expresses that data contained in the block is likely to be accessed in the near future

Introduction

Operation

- When the processor finds a requested data item in the cache, it is called a cache hit
- When the processor does not find a data item in the cache, a cache miss occurs
- A fixed-size collection of data, block, is retrieved from the main memory and placed into the cache
- The principle of locality expresses that data contained in the block is likely to be accessed in the near future

Organization in blocks

Main memory $\Leftrightarrow$ big block store

- cache memory stores a copy of some of them

$$\text{Address provided by the CPU} \begin{cases} \bullet \text{ main memory block} \\ \bullet \text{ block offset} \end{cases}$$

Example

- Word: 1 byte
- Cache size: 32 bytes
- Block size: 4 bytes
- Main memory size: 256 bytes

Organization in blocks

Main memory $\Leftrightarrow$ big block store

- cache memory stores a copy of some of them

$$\text{Address provided by the CPU} \left\{ \begin{array}{l} \bullet \text{ main memory block} \\ \\ \bullet \text{ block offset} \end{array} \right.$$

Example

- Word: 1 byte

$$\left. \begin{array}{l} \bullet \text{ Cache size: 32 bytes} \\ \bullet \text{ Block size: 4 bytes} \end{array} \right\} \text{8 blocks}$$

- Main memory size: 256 bytes

Organization in blocks

Main memory $\Leftrightarrow$ big block store

- cache memory stores a copy of some of them

$$\text{Address provided by the CPU} \quad \left\{ \begin{array}{l} \bullet \text{ main memory block} \\ \\ \bullet \text{ block offset} \end{array} \right.$$

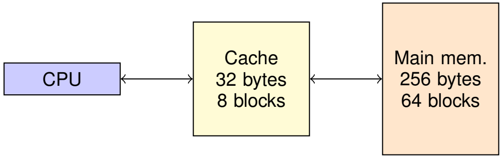
Example

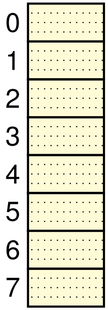
- Word: 1 byte

$$\left. \begin{array}{l} \bullet \text{ Cache size: 32 bytes} \\ \bullet \text{ Block size: 4 bytes} \end{array} \right\} \text{8 blocks}$$

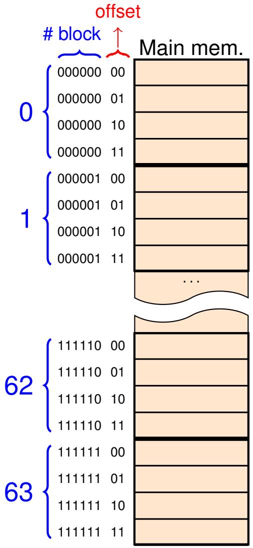
- Main memory size: 256 bytes $\Rightarrow$ 64 blocks

Organization in blocks





- Word: 1 byte
- Cache size: 32 bytes
- Block size: 4 bytes
- Main memory size: 256 bytes



Features

Cache controller

Manages the cache memory

1. Determines whether a memory access operation is a hit or a miss

Design characteristics

2. Placement strategy
   - Where can a block be placed in the cache memory?
3. Replacement strategy
   - Which block should be replaced on a miss?
4. Write strategy
   - What happens on a write?

Placement strategies

Where can a block be placed in the cache memory?

- The most popular scheme is *set associative*, where a *set* is a group of blocks in the cache $\rightarrow$ a block is first mapped onto a set, and then the block can be placed anywhere within the set
- *n* blocks in a set $\rightarrow$ *n*-way set associative
- End points of set associative
  - *Direct mapped* cache: one block per set
  - *Fully associative* cache: only one set

Direct mapped placement

Easiest placement scheme

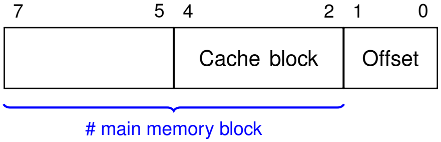
$$\boxed{\text{cache block} = (\text{main memory block})\ \text{MOD}\ \underbrace{(\#\ \text{of cache blocks})}_{2^x}}$$

$$\Updownarrow$$

$$\text{cache block} = x \text{ least significant bits of the main memory block}$$

Example

Address issued by the CPU



$$\text{FEh} = 1111\,1110 \longrightarrow \underbrace{111\ \overbrace{111}}\ 10$$

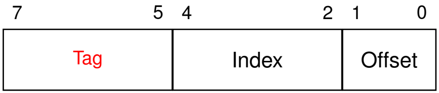
$$\text{33h} = 0011\,0011 \longrightarrow \underbrace{001\ \overbrace{100}}\ 11$$

Direct mapped placement

Address bits

- Offset → Word offset in the block
- Index → Select the set
- Tag → Identify main memory block in the cache

Address issued by the CPU



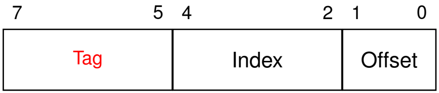
- Each cache block has a valid bit assigned

Direct mapped placement

Address bits

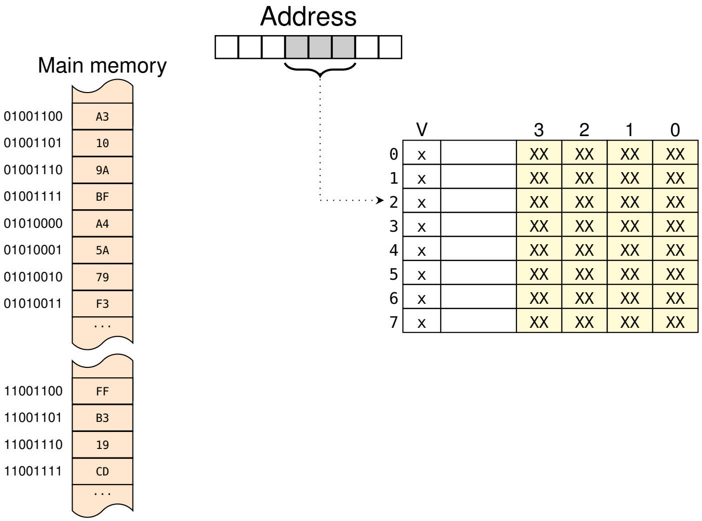
- Offset → Word offset in the block
- Index → Select the set
- Tag → Identify main memory block in the cache

Address issued by the CPU

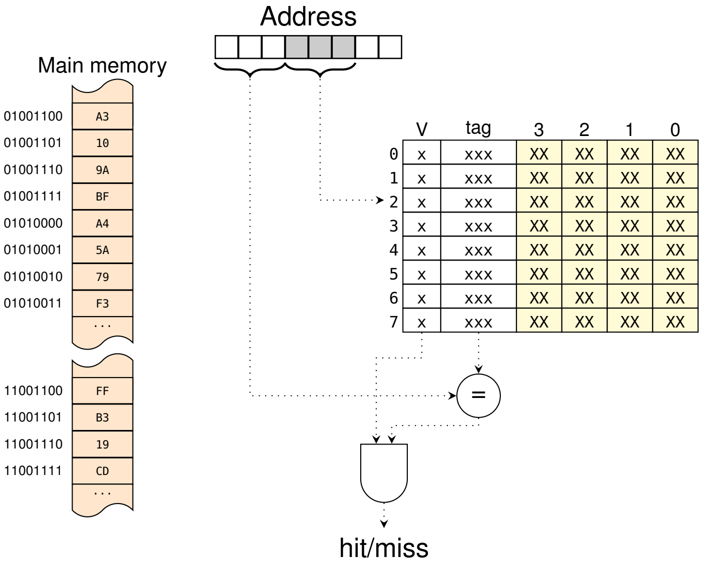


- Each cache block has a valid bit assigned

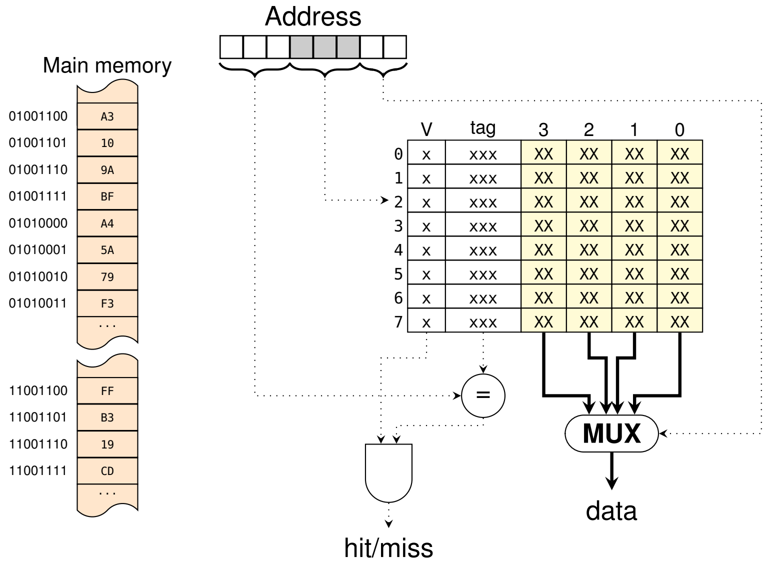
Direct mapped placement



Direct mapped placement

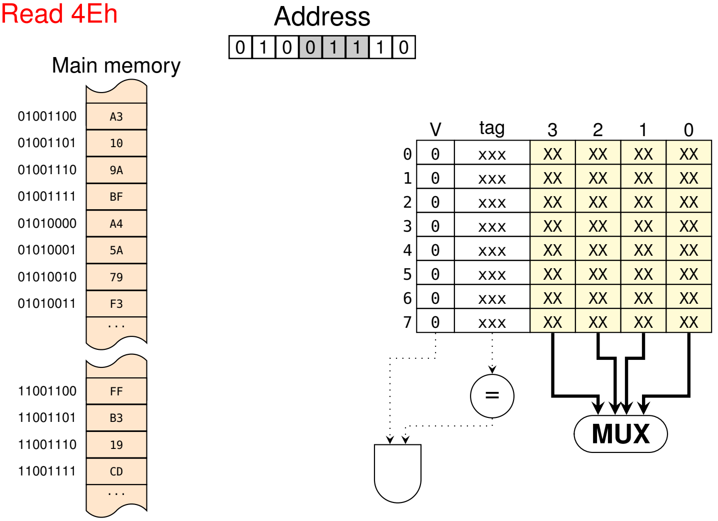


Direct mapped placement



Direct mapped placement

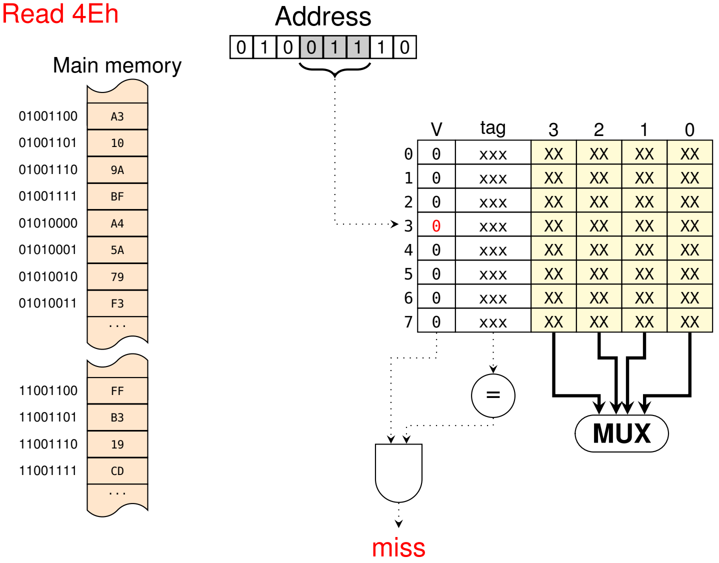
Read 4Eh



	V	tag	3	2	1	0
0	0	xxx	XX	XX	XX	XX
1	0	xxx	XX	XX	XX	XX
2	0	xxx	XX	XX	XX	XX
3	0	xxx	XX	XX	XX	XX
4	0	xxx	XX	XX	XX	XX
5	0	xxx	XX	XX	XX	XX
6	0	xxx	XX	XX	XX	XX
7	0	xxx	XX	XX	XX	XX

Direct mapped placement

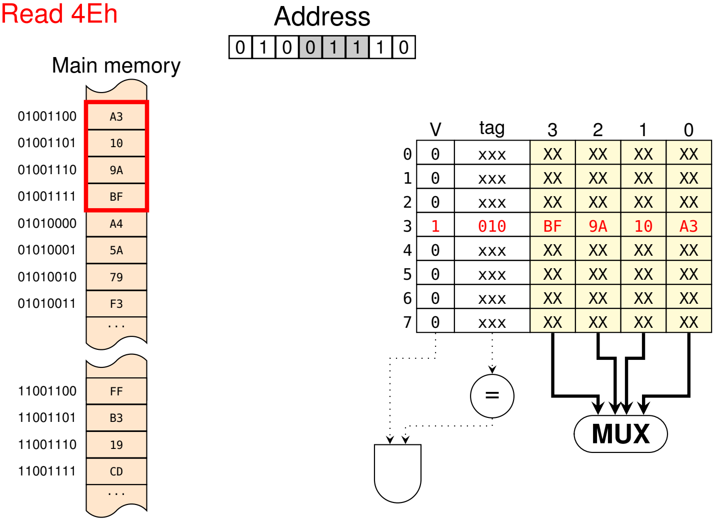
Read 4Eh



miss

Direct mapped placement

Read 4Eh

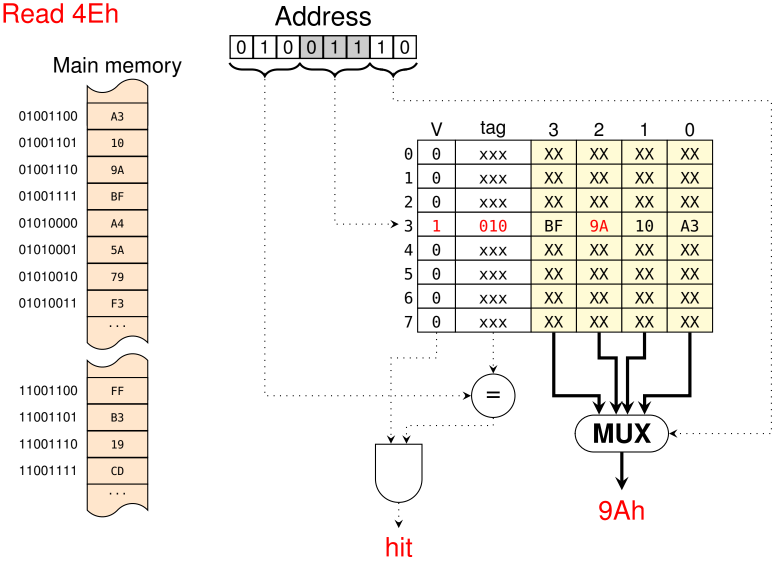


01001100	A3
01001101	10
01001110	9A
01001111	BF
01010000	A4
01010001	5A
01010010	79
01010011	F3
	...
11001100	FF
11001101	B3
11001110	19
11001111	CD
	...

	V	tag	3	2	1	0
0	0	xxx	XX	XX	XX	XX
1	0	xxx	XX	XX	XX	XX
2	0	xxx	XX	XX	XX	XX
3	1	010	BF	9A	10	A3
4	0	xxx	XX	XX	XX	XX
5	0	xxx	XX	XX	XX	XX
6	0	xxx	XX	XX	XX	XX
7	0	xxx	XX	XX	XX	XX

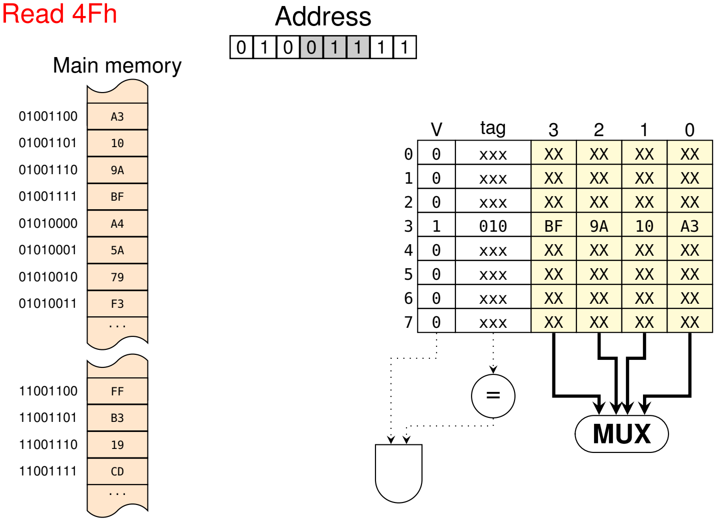
Direct mapped placement

Read 4Eh



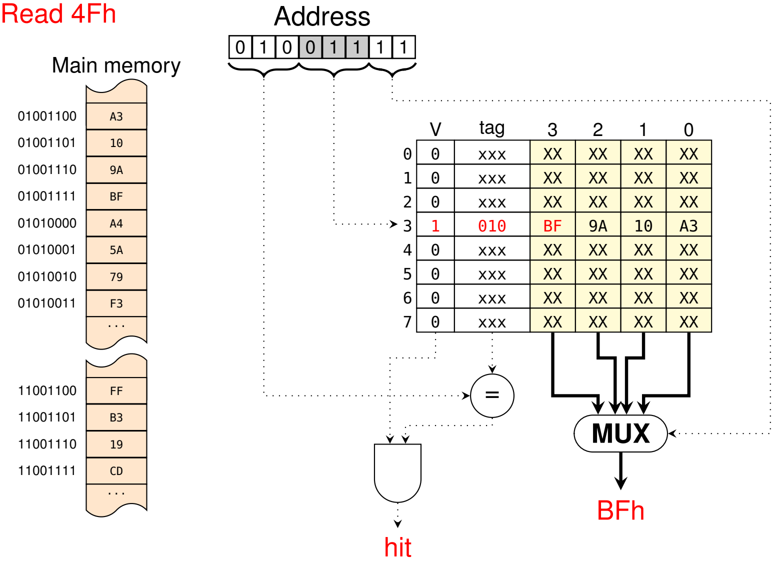
Direct mapped placement

Read 4Fh



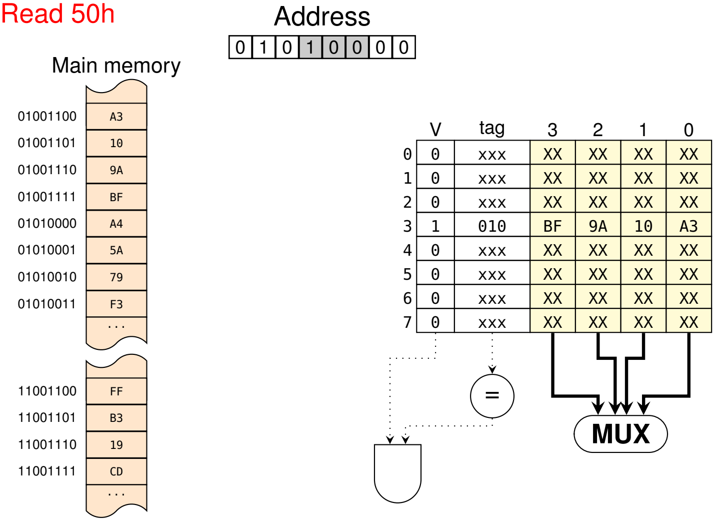
Direct mapped placement

Read 4Fh



Direct mapped placement

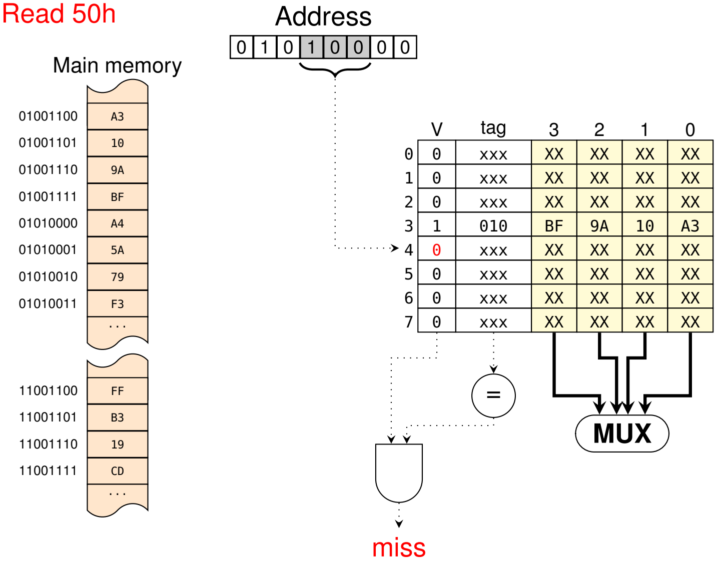
Read 50h



	V	tag	3	2	1	0
0	0	xxx	XX	XX	XX	XX
1	0	xxx	XX	XX	XX	XX
2	0	xxx	XX	XX	XX	XX
3	1	010	BF	9A	10	A3
4	0	xxx	XX	XX	XX	XX
5	0	xxx	XX	XX	XX	XX
6	0	xxx	XX	XX	XX	XX
7	0	xxx	XX	XX	XX	XX

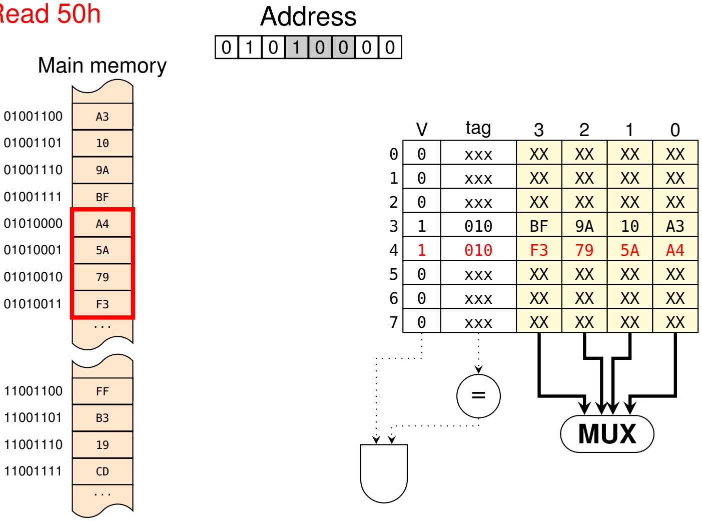
Direct mapped placement

Read 50h



Direct mapped placement

Read 50h



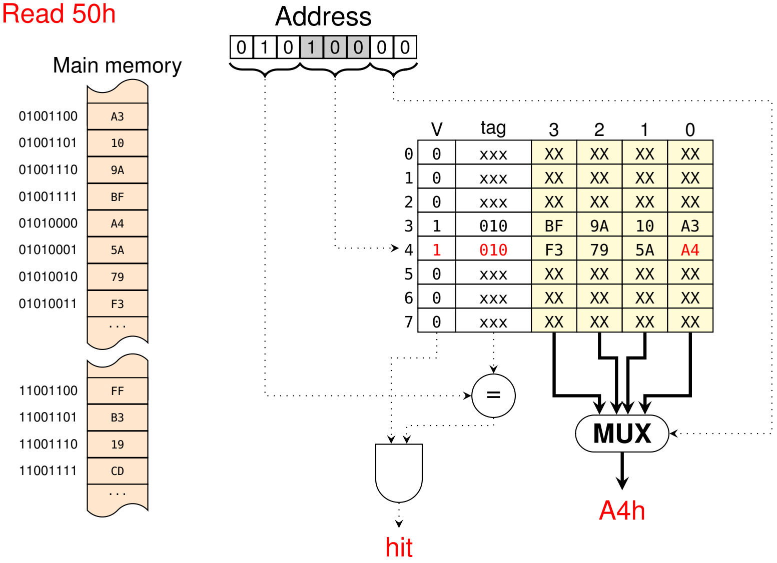
Main memory

01001100	A3
01001101	10
01001110	9A
01001111	BF
01010000	A4
01010001	5A
01010010	79
01010011	F3
	...
11001100	FF
11001101	B3
11001110	19
11001111	CD
	...

	V	tag	3	2	1	0
0	0	xxx	XX	XX	XX	XX
1	0	xxx	XX	XX	XX	XX
2	0	xxx	XX	XX	XX	XX
3	1	010	BF	9A	10	A3
4	1	010	F3	79	5A	A4
5	0	xxx	XX	XX	XX	XX
6	0	xxx	XX	XX	XX	XX
7	0	xxx	XX	XX	XX	XX

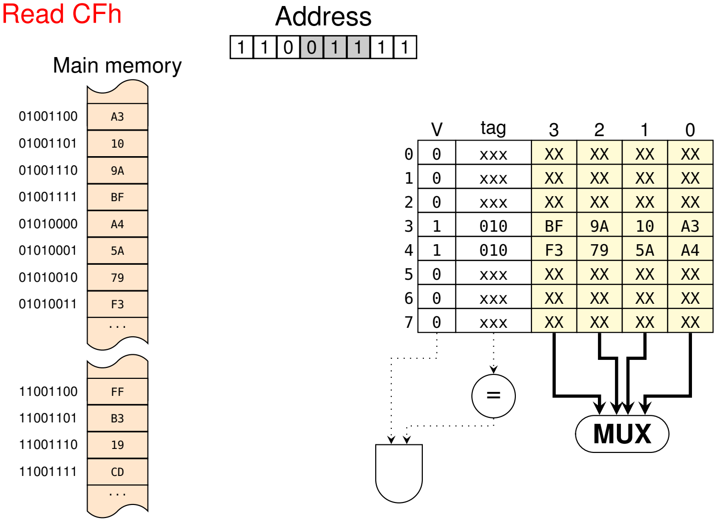
Direct mapped placement

Read 50h



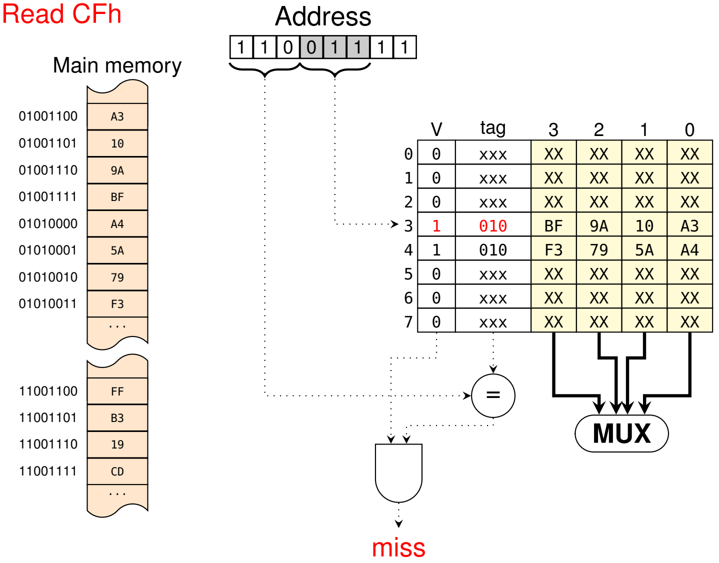
Direct mapped placement

Read CFh



Direct mapped placement

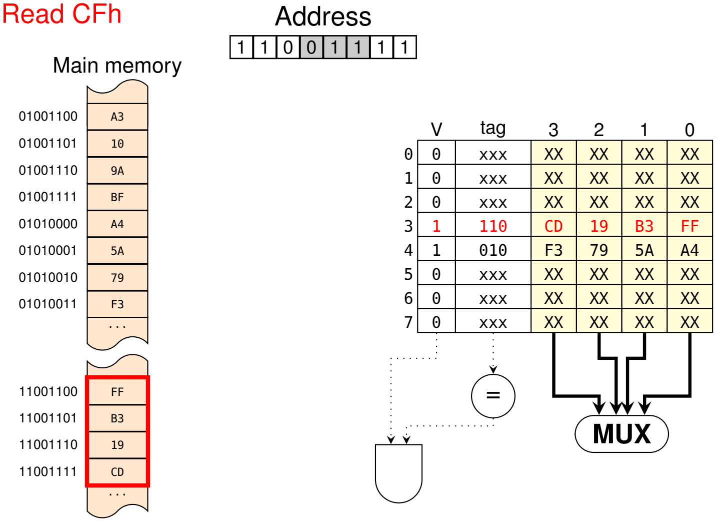
Read CFh



	V	tag	3	2	1	0
0	0	xxx	XX	XX	XX	XX
1	0	xxx	XX	XX	XX	XX
2	0	xxx	XX	XX	XX	XX
3	1	010	BF	9A	10	A3
4	1	010	F3	79	5A	A4
5	0	xxx	XX	XX	XX	XX
6	0	xxx	XX	XX	XX	XX
7	0	xxx	XX	XX	XX	XX

Direct mapped placement

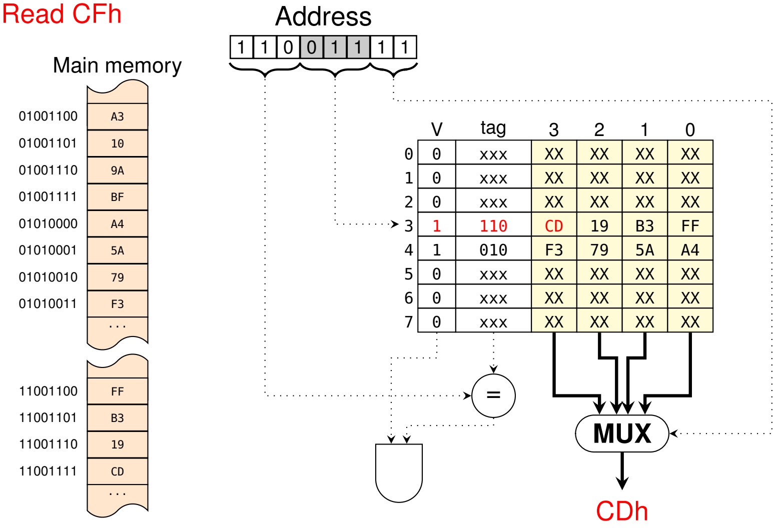
Read CFh



	V	tag	3	2	1	0
0	0	xxx	XX	XX	XX	XX
1	0	xxx	XX	XX	XX	XX
2	0	xxx	XX	XX	XX	XX
3	1	110	CD	19	B3	FF
4	1	010	F3	79	5A	A4
5	0	xxx	XX	XX	XX	XX
6	0	xxx	XX	XX	XX	XX
7	0	xxx	XX	XX	XX	XX

Direct mapped placement

Read CFh



	V	tag	3	2	1	0
0	0	xxx	XX	XX	XX	XX
1	0	xxx	XX	XX	XX	XX
2	0	xxx	XX	XX	XX	XX
3	1	110	CD	19	B3	FF
4	1	010	F3	79	5A	A4
5	0	xxx	XX	XX	XX	XX
6	0	xxx	XX	XX	XX	XX
7	0	xxx	XX	XX	XX	XX

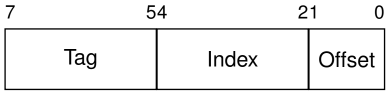
Fully associative placement

Direct mapped placement

Simple, but no flexibility

Example

- Word: 1 byte
- Cache size: 32 bytes
- Block size: 4 bytes
- Main memory size: 256 bytes



$$
\begin{array}{rlcl}
37\mathrm{h} = & 0011\,0111 & \longrightarrow & 001\ \boxed{101}\ 11 \\
\mathrm{F4h} = & 1111\,0100 & \longrightarrow & 111\ \boxed{101}\ 00 \\
56\mathrm{h} = & 0101\,0110 & \longrightarrow & 010\ \boxed{101}\ 10
\end{array}
$$

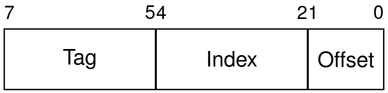
all to the same cache block

Fully associative placement

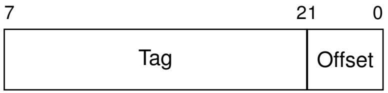
Total freedom

Any cache block can be used

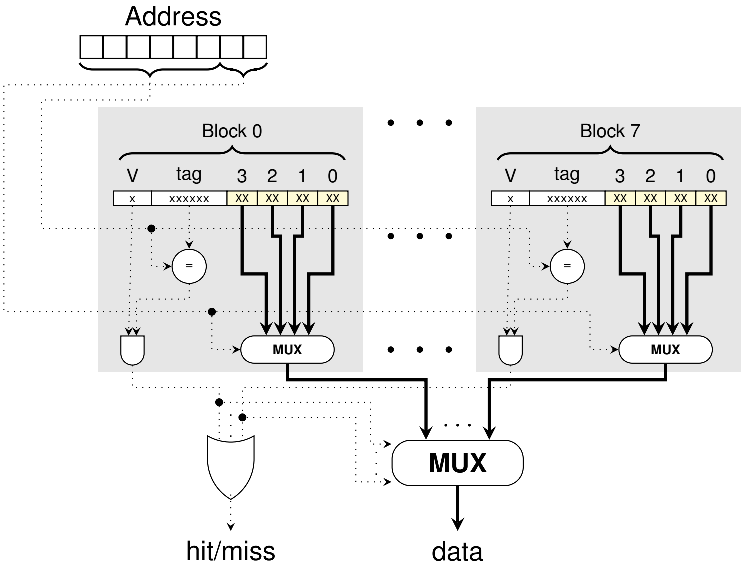
Direct mapped placement



Fully associative placement



Fully associative placement



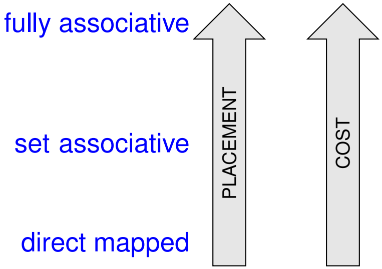
Set associative placement

Fully associative placement problem

Efficient, but high cost

- number of comparators

$$\text{placement policy} \begin{cases} \text{direct mapped} \Rightarrow \text{one for the whole cache} \\ \text{fully associative} \Rightarrow \text{one per block} \end{cases}$$



Set associative placement

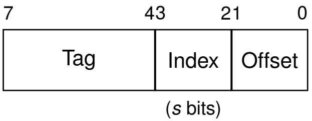
Key

Cache blocks are grouped in sets

- direct mapped placement to the set
- fully associative placement inside the set

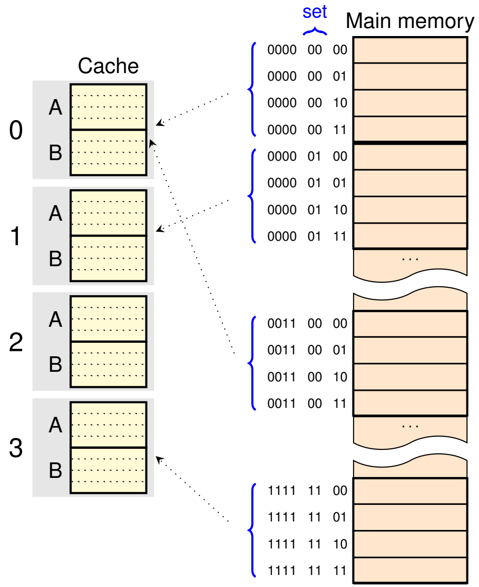
Example

Address issued by the CPU

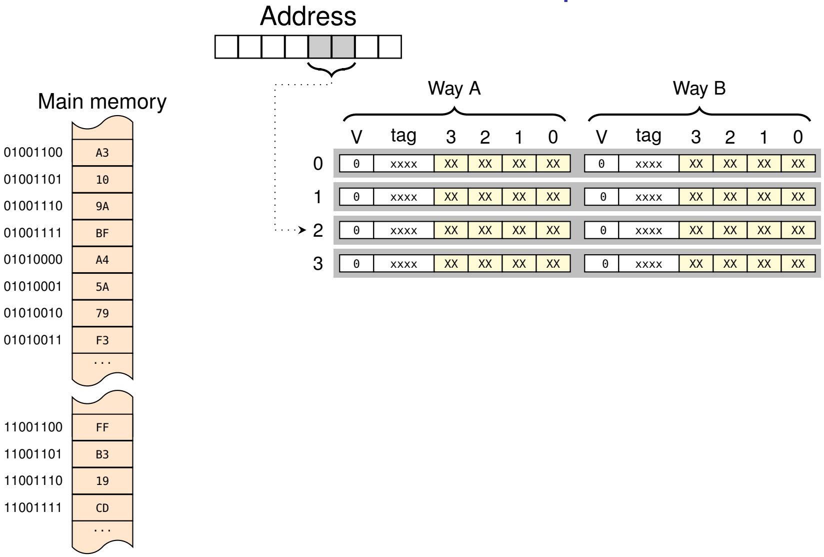


Set associative placement

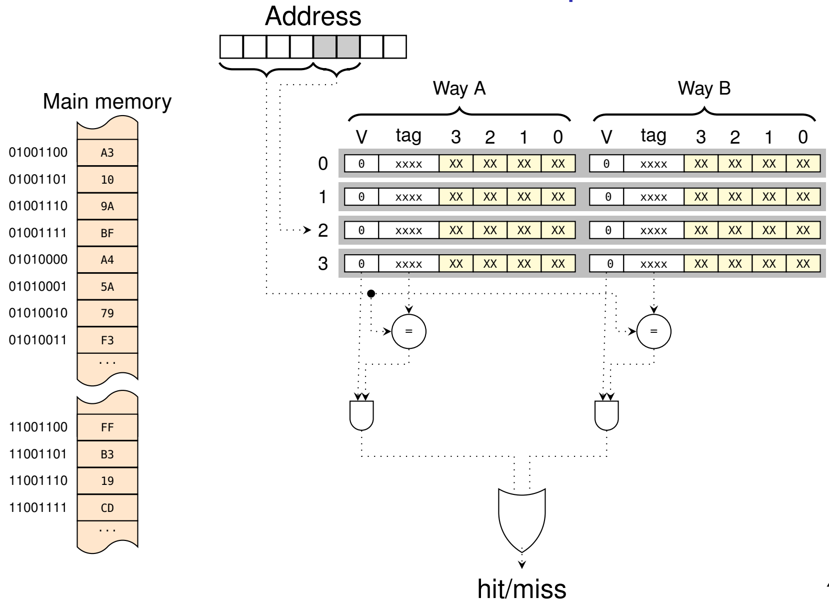
- Blocks directly assigned to the set
- Each set has two ways
- Tag is defined by the bits not used for the set



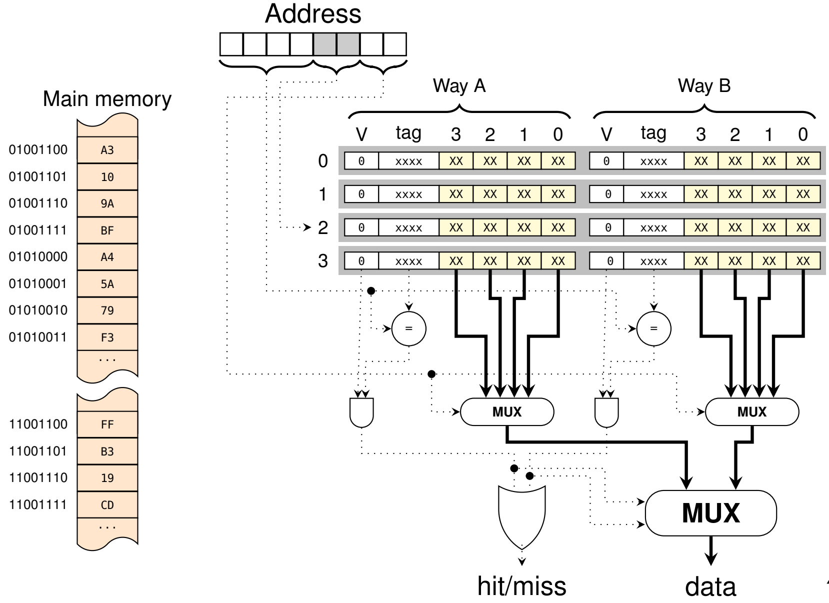
Set associative placement



Set associative placement



Set associative placement



Set associative placement

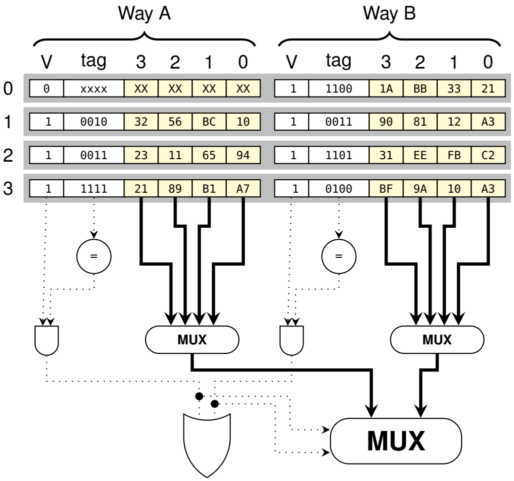
Read 4Ch

Address

0	1	0	0	1	1	0	0

Main memory

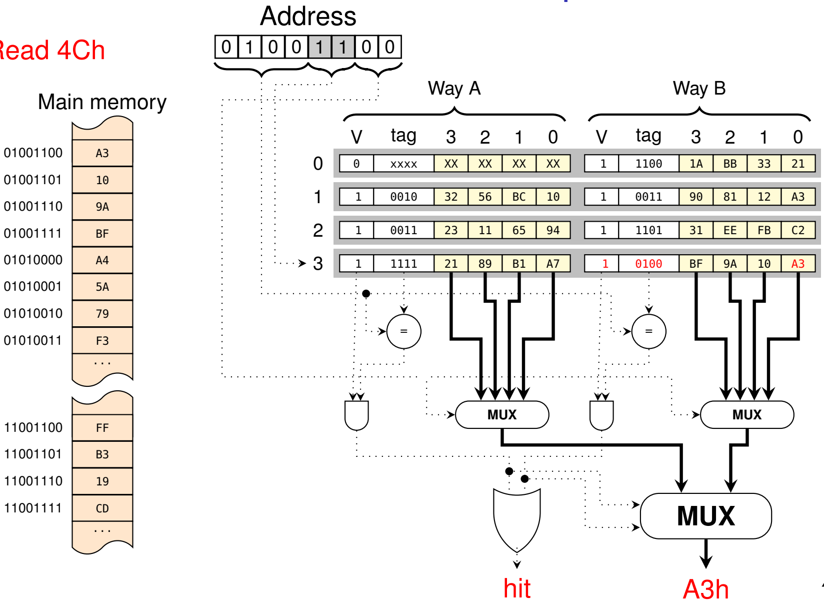
01001100	A3
01001101	10
01001110	9A
01001111	BF
01010000	A4
01010001	5A
01010010	79
01010011	F3
	...
11001100	FF
11001101	B3
11001110	19
11001111	CD
	...



	Way A V	tag	3	2	1	0	Way B V	tag	3	2	1	0
0	0	xxxx	XX	XX	XX	XX	1	1100	1A	BB	33	21
1	1	0010	32	56	BC	10	1	0011	90	81	12	A3
2	1	0011	23	11	65	94	1	1101	31	EE	FB	C2
3	1	1111	21	89	B1	A7	1	0100	BF	9A	10	A3

Set associative placement

Read 4Ch

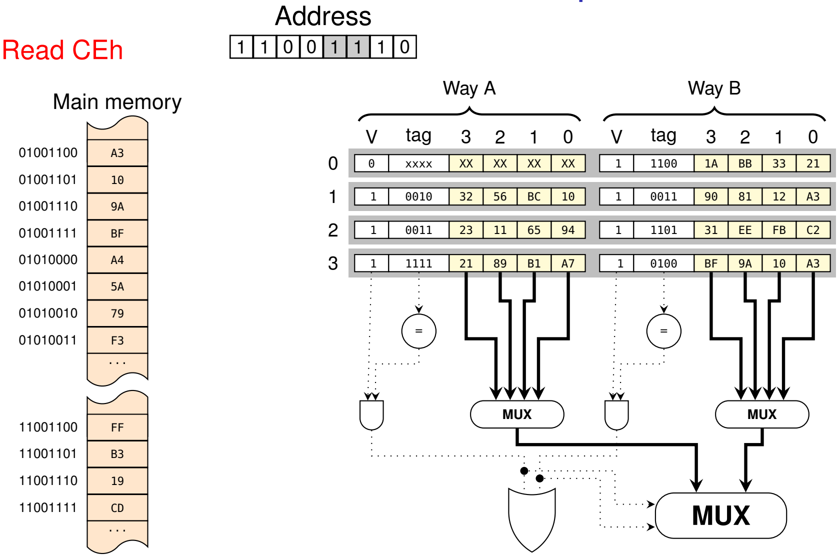


01001100	A3
01001101	10
01001110	9A
01001111	BF
01010000	A4
01010001	5A
01010010	79
01010011	F3
	...
11001100	FF
11001101	B3
11001110	19
11001111	CD
	...

	V	tag	3	2	1	0	V	tag	3	2	1	0
0	0	xxxx	XX	XX	XX	XX	1	1100	1A	BB	33	21
1	1	0010	32	56	BC	10	1	0011	90	81	12	A3
2	1	0011	23	11	65	94	1	1101	31	EE	FB	C2
3	1	1111	21	89	B1	A7	1	0100	BF	9A	10	A3

Set associative placement

Read CEh

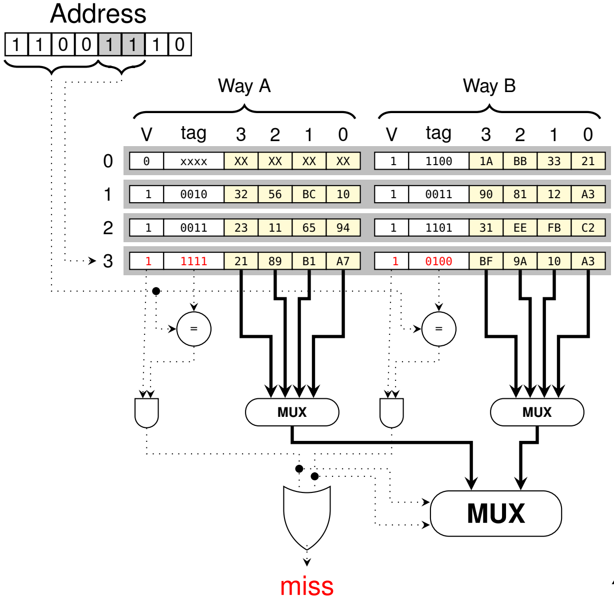


Set associative placement

Read CEh

Main memory

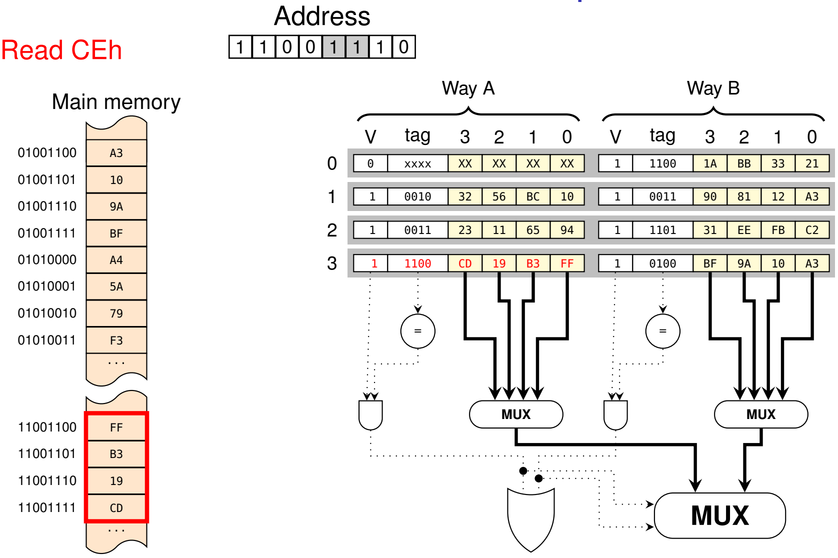
01001100	A3
01001101	10
01001110	9A
01001111	BF
01010000	A4
01010001	5A
01010010	79
01010011	F3
	...
11001100	FF
11001101	B3
11001110	19
11001111	CD
	...



	Way A V	Way A tag	Way A 3	Way A 2	Way A 1	Way A 0	Way B V	Way B tag	Way B 3	Way B 2	Way B 1	Way B 0
0	0	xxxx	XX	XX	XX	XX	1	1100	1A	BB	33	21
1	1	0010	32	56	BC	10	1	0011	90	81	12	A3
2	1	0011	23	11	65	94	1	1101	31	EE	FB	C2
3	1	1111	21	89	B1	A7	1	0100	BF	9A	10	A3

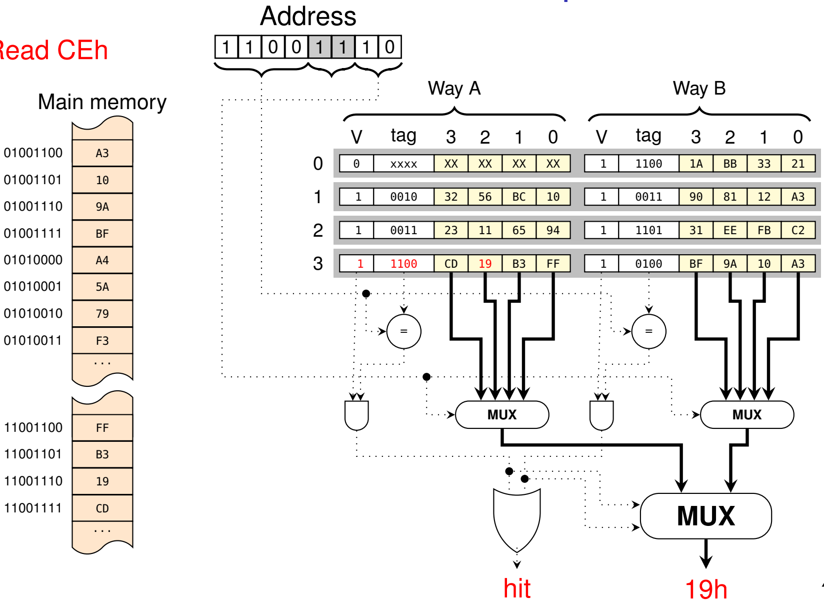
Set associative placement

Read CEh



Set associative placement

Read CEh



Main memory

01001100	A3
01001101	10
01001110	9A
01001111	BF
01010000	A4
01010001	5A
01010010	79
01010011	F3
	...
11001100	FF
11001101	B3
11001110	19
11001111	CD
	...

	Way A V	Way A tag	3	2	1	0	Way B V	Way B tag	3	2	1	0
0	0	xxxx	XX	XX	XX	XX	1	1100	1A	BB	33	21
1	1	0010	32	56	BC	10	1	0011	90	81	12	A3
2	1	0011	23	11	65	94	1	1101	31	EE	FB	C2
3	1	1100	CD	19	B3	FF	1	0100	BF	9A	10	A3

Replacement strategies

When a miss occurs, which block will be replaced?

Two different scenarios

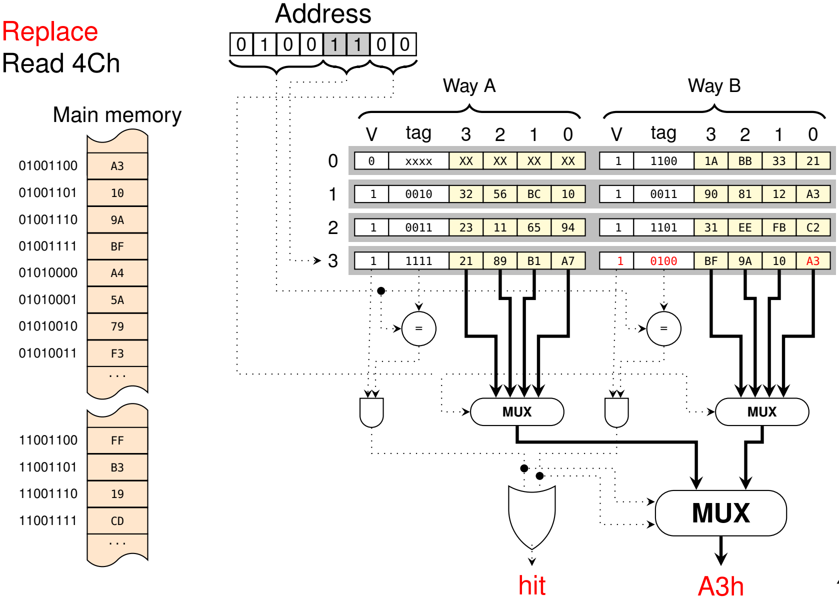
1. several allocations available for the block to be copied
2. all allocations 'are occupied'

Strategies

- *Least Recently Used* (LRU) $\rightarrow$ The block replaced is the one that has been unused for the longest time
- Random $\rightarrow$ Candidate blocks are randomly selected to spread allocation uniformly

Example

Replace
Read 4Ch



Example

Replace

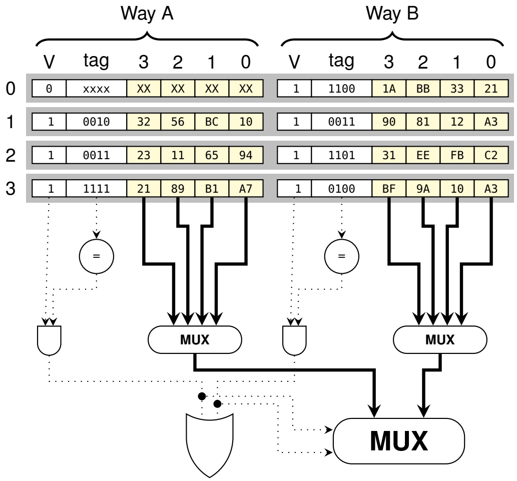
Read CEh

Address

1	1	0	0	1	1	1	0

Main memory

01001100	A3
01001101	10
01001110	9A
01001111	BF
01010000	A4
01010001	5A
01010010	79
01010011	F3
	...
11001100	FF
11001101	B3
11001110	19
11001111	CD
	...



	Way A V	Way A tag	3	2	1	0	Way B V	Way B tag	3	2	1	0
0	0	xxxx	XX	XX	XX	XX	1	1100	1A	BB	33	21
1	1	0010	32	56	BC	10	1	0011	90	81	12	A3
2	1	0011	23	11	65	94	1	1101	31	EE	FB	C2
3	1	1111	21	89	B1	A7	1	0100	BF	9A	10	A3

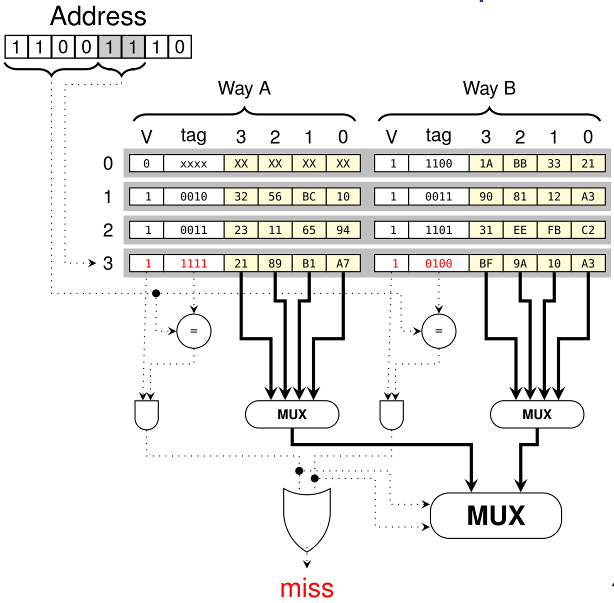
Example

Replace

Read CEh

Main memory

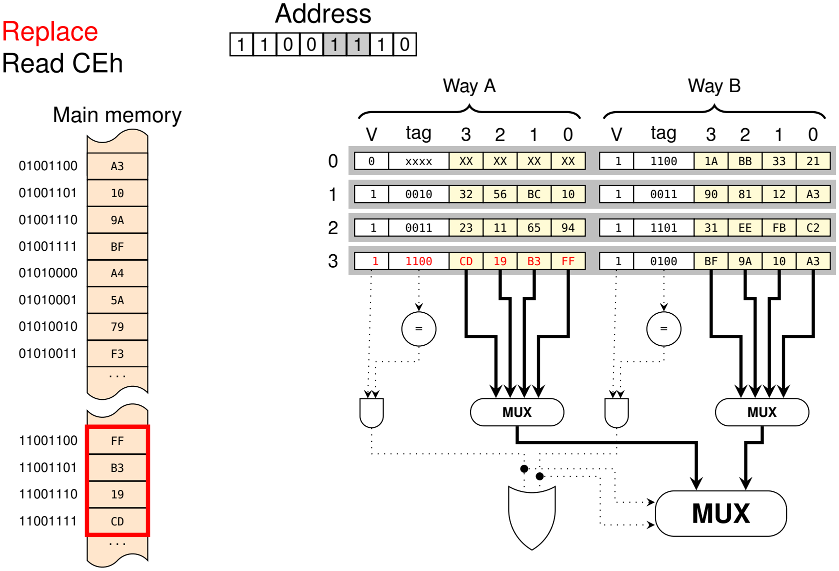
01001100	A3
01001101	10
01001110	9A
01001111	BF
01010000	A4
01010001	5A
01010010	79
01010011	F3
	...
11001100	FF
11001101	B3
11001110	19
11001111	CD
	...



	Way A						Way B					
	V	tag	3	2	1	0	V	tag	3	2	1	0
0	0	xxxx	XX	XX	XX	XX	1	1100	1A	BB	33	21
1	1	0010	32	56	BC	10	1	0011	90	81	12	A3
2	1	0011	23	11	65	94	1	1101	31	EE	FB	C2
3	1	1111	21	89	B1	A7	1	0100	BF	9A	10	A3

Example

Replace
Read CEh



Writing strategies

What happens when the CPU writes a data item?

- The information in the cache is a copy of the lower levels
- Writing operations can or cannot be cached

Strategies

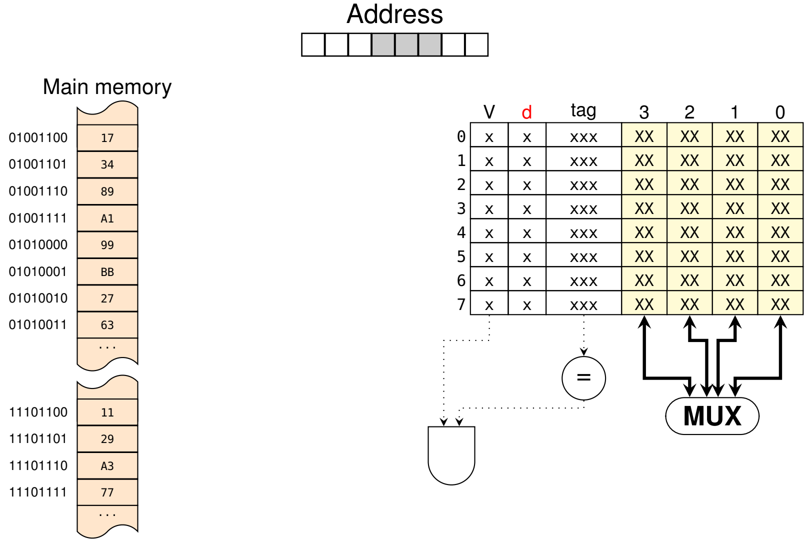
- *Write-trough*
  - write simultaneously to several levels of the hierarchy
- *Write-back*
  - the block is written to the lower level when it is replaced

Cache miss

- *Write allocate*
  - the block is copied into the cache and then it is written
- *No write allocate*
  - the writing operation is performed only to main memory (it is not cached)

Write-back

Higher hardware complexity ⇒ *dirty* bit



	V	d	tag	3	2	1	0
0	x	x	xxx	XX	XX	XX	XX
1	x	x	xxx	XX	XX	XX	XX
2	x	x	xxx	XX	XX	XX	XX
3	x	x	xxx	XX	XX	XX	XX
4	x	x	xxx	XX	XX	XX	XX
5	x	x	xxx	XX	XX	XX	XX
6	x	x	xxx	XX	XX	XX	XX
7	x	x	xxx	XX	XX	XX	XX

Write-back

Higher hardware complexity $\Rightarrow$ *dirty* bit

Write FFh in 4Dh

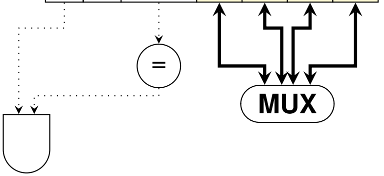
Address

0	1	0	0	1	1	0	1

Main memory

01001100	17
01001101	34
01001110	89
01001111	A1
01010000	99
01010001	BB
01010010	27
01010011	63
	...
11101100	11
11101101	29
11101110	A3
11101111	77
	...

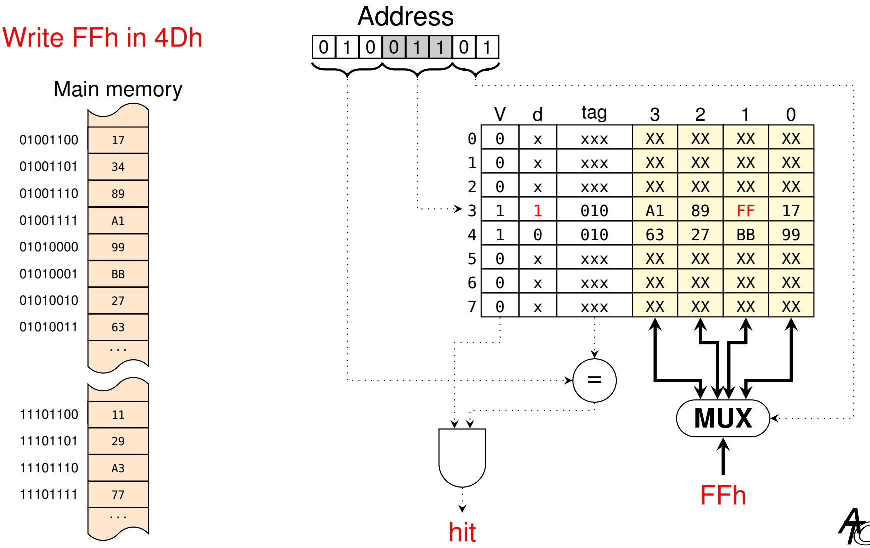
	V	d	tag	3	2	1	0
0	0	x	xxx	XX	XX	XX	XX
1	0	x	xxx	XX	XX	XX	XX
2	0	x	xxx	XX	XX	XX	XX
3	1	0	010	A1	89	34	17
4	1	0	010	63	27	BB	99
5	0	x	xxx	XX	XX	XX	XX
6	0	x	xxx	XX	XX	XX	XX
7	0	x	xxx	XX	XX	XX	XX



Write-back

Higher hardware complexity ⇒ *dirty* bit

Write FFh in 4Dh



Write-back

Higher hardware complexity $\Rightarrow$ *dirty* bit

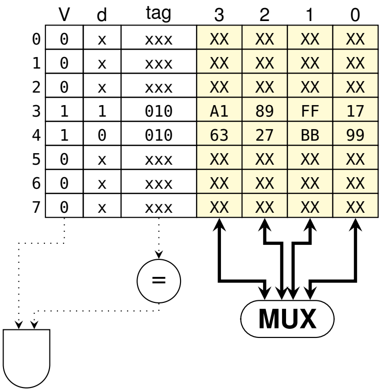
Read EEh

Address

1	1	1	0	1	1	1	0

Main memory

01001100	17
01001101	34
01001110	89
01001111	A1
01010000	99
01010001	BB
01010010	27
01010011	63
	...
11101100	11
11101101	29
11101110	A3
11101111	77
	...

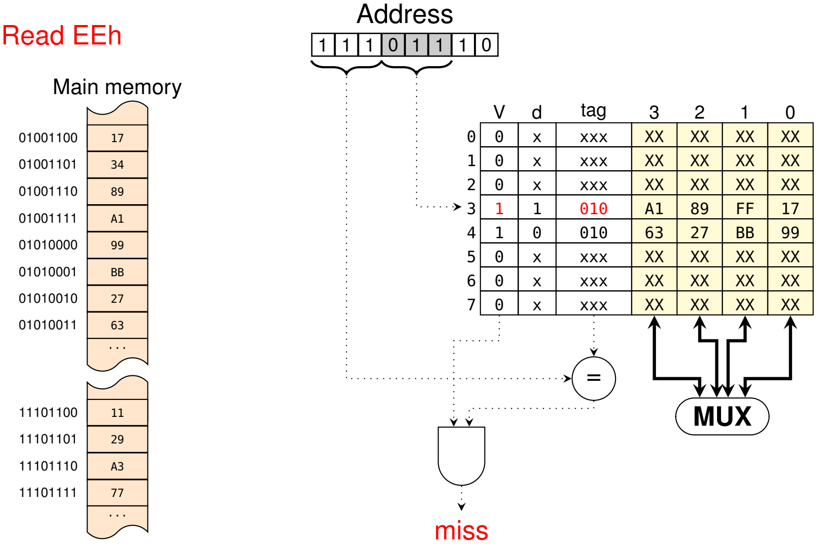


	V	d	tag	3	2	1	0
0	0	x	xxx	XX	XX	XX	XX
1	0	x	xxx	XX	XX	XX	XX
2	0	x	xxx	XX	XX	XX	XX
3	1	1	010	A1	89	FF	17
4	1	0	010	63	27	BB	99
5	0	x	xxx	XX	XX	XX	XX
6	0	x	xxx	XX	XX	XX	XX
7	0	x	xxx	XX	XX	XX	XX

Write-back

Higher hardware complexity ⇒ *dirty* bit

Read EEh



Write-back

Higher hardware complexity ⇒ *dirty* bit

Read EEh

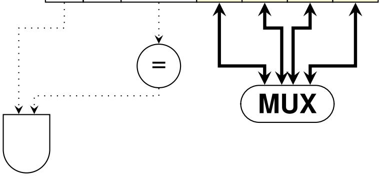
Address

1	1	1	0	1	1	1	0

Main memory

01001100	17
01001101	FF
01001110	89
01001111	A1
01010000	99
01010001	BB
01010010	27
01010011	63
	...
11101100	11
11101101	29
11101110	A3
11101111	77
	...

	V	d	tag	3	2	1	0
0	0	x	xxx	XX	XX	XX	XX
1	0	x	xxx	XX	XX	XX	XX
2	0	x	xxx	XX	XX	XX	XX
3	1	1	010	A1	89	FF	17
4	1	0	010	63	27	BB	99
5	0	x	xxx	XX	XX	XX	XX
6	0	x	xxx	XX	XX	XX	XX
7	0	x	xxx	XX	XX	XX	XX



Write-back

Higher hardware complexity $\Rightarrow$ *dirty* bit

Read EEh

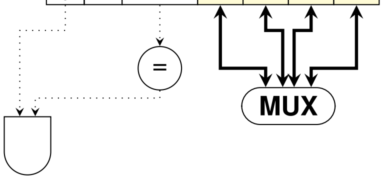
Address

1	1	1	0	1	1	1	0

Main memory

01001100	17
01001101	FF
01001110	89
01001111	A1
01010000	99
01010001	BB
01010010	27
01010011	63
	...
11101100	11
11101101	29
11101110	A3
11101111	77
	...

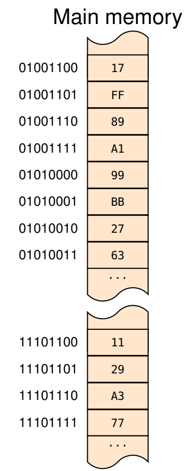
	V	d	tag	3	2	1	0
0	0	x	xxx	XX	XX	XX	XX
1	0	x	xxx	XX	XX	XX	XX
2	0	x	xxx	XX	XX	XX	XX
3	1	0	111	77	A3	29	11
4	1	0	010	63	27	BB	99
5	0	x	xxx	XX	XX	XX	XX
6	0	x	xxx	XX	XX	XX	XX
7	0	x	xxx	XX	XX	XX	XX

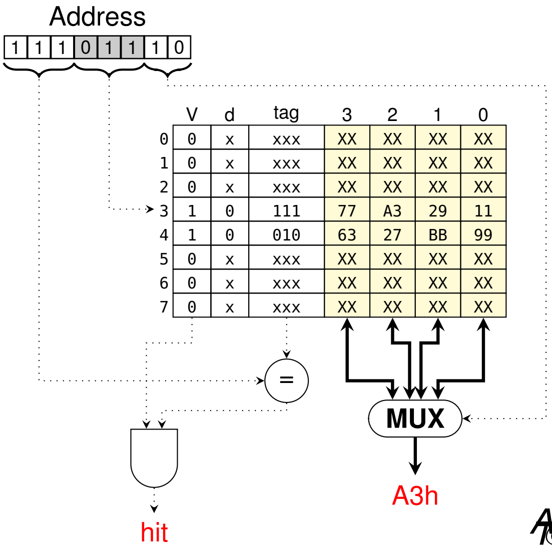


Write-back

Higher hardware complexity ⇒ *dirty* bit

Read EEh





	V	d	tag	3	2	1	0
0	0	x	xxx	XX	XX	XX	XX
1	0	x	xxx	XX	XX	XX	XX
2	0	x	xxx	XX	XX	XX	XX
3	1	0	111	77	A3	29	11
4	1	0	010	63	27	BB	99
5	0	x	xxx	XX	XX	XX	XX
6	0	x	xxx	XX	XX	XX	XX
7	0	x	xxx	XX	XX	XX	XX

Coherence problems

$$\left.\begin{array}{l}\bullet \text{ The CPU reads and writes from the cache memory} \\ \bullet \text{ The cache memory stores copies from the main memory}\end{array}\right\}\ \text{Memory location with different values}$$

Any other device accessing the main memory?

Coherence problems may appear

1. Main memory is modified $\Rightarrow$ The CPU accesses an obsolete data item
2. CPU writes to cache $\Rightarrow$ Main memory obsolete

What devices can access the main memory?

1. I/O interfaces mapped in the address space
2. I/O interfaces via Direct Memory Access (DMA)

Solutions

1. Set areas as non-cacheable
    - it is not optimal for DMA interfaces
2. *Snooping*
    - observes control and address lines
    - stops the interface to undo incoherences

Coherence problems

I/O interface reads from the main memory

- ✔ *Write-through* $\Rightarrow$ no problem
- ✘ *Write-back* $\Rightarrow$ problems with dirty blocks

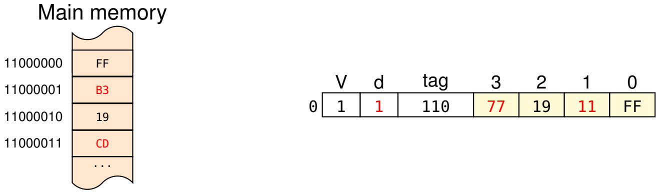
I/O interface writes in the main memory

- ✘ *Write-through* $\Rightarrow$ problem if the block is cached
- ✘ *Write-back* $\Rightarrow$ problem if the block is cached (furthermore, it may be a dirty block)

I/O interface reading + *write-back*

The block to be read is a dirty block

1. dirty block in cache ⇒ incoherence with the main memory
2. the peripheral device requests reading the block (11000010)
3. the cache controller stops the reading operation and updates the block in the main memory
4. the reading operation is now allowed

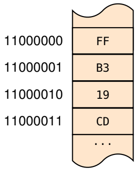


I/O interface reading + *write-back*

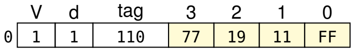
The block to be read is a dirty block

1. dirty block in cache $\Rightarrow$ incoherence with the main memory
2. the peripheral device requests reading the block (11000010)
3. the cache controller stops the reading operation and updates the block in the main memory
4. the reading operation is now allowed

Main memory



11000000	FF
11000001	B3
11000010	19
11000011	CD
	...

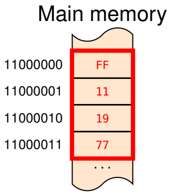


	V	d	tag	3	2	1	0
0	1	1	110	77	19	11	FF

I/O interface reading + *write-back*

The block to be read is a dirty block

1. dirty block in cache ⇒ incoherence with the main memory
2. the peripheral device requests reading the block (11000010)
3. the cache controller stops the reading operation and updates the block in the main memory
4. the reading operation is now allowed

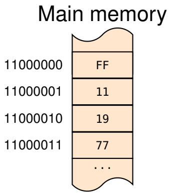


	V	d	tag	3	2	1	0
0	1	0	110	77	19	11	FF

I/O interface reading + *write-back*

The block to be read is a dirty block

1. dirty block in cache ⇒ incoherence with the main memory
2. the peripheral device requests reading the block (11000010)
3. the cache controller stops the reading operation and updates the block in the main memory
4. the reading operation is now allowed

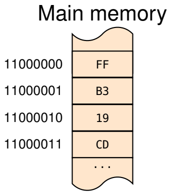


	V	d	tag	3	2	1	0
0	1	0	110	77	19	11	FF

I/O interface writing + *write-through*

The block is cached

1. block cached and coherent with the main memory
2. the peripheral device requests writing the block (11000011)
3. the cache controller allows the writing operation
4. the block in the cache is invalidated

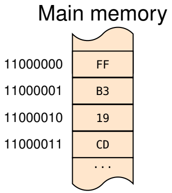


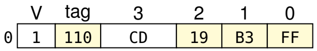
	V	tag	3	2	1	0
0	1	110	CD	19	B3	FF

I/O interface writing + *write-through*

The block is cached

1. block cached and coherent with the main memory
2. the peripheral device requests writing the block (11000011)
3. the cache controller allows the writing operation
4. the block in the cache is invalidated

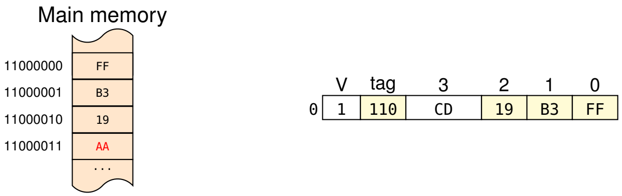




I/O interface writing + *write-through*

The block is cached

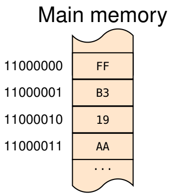
1. block cached and coherent with the main memory
2. the peripheral device requests writing the block (11000011)
3. the cache controller allows the writing operation
4. the block in the cache is invalidated

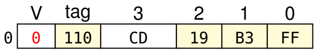


I/O interface writing + *write-through*

The block is cached

1. block cached and coherent with the main memory
2. the peripheral device requests writing the block (11000011)
3. the cache controller allows the writing operation
4. the block in the cache is invalidated



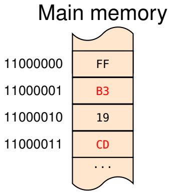


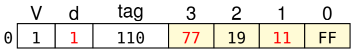
	V	tag	3	2	1	0
0	0	110	CD	19	B3	FF

I/O interface writing + *write-back*

The block is cached (and dirty)

1. the block is cached and dirty $\Rightarrow$ incoherent with the main memory
2. the peripheral device requests writing the block (11000011)
3. the cache controller stops the writing operation and updates the block in the main memory
4. the writing operation is allowed and the block in the cache is invalidated

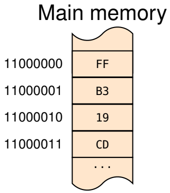




I/O interface writing + *write-back*

The block is cached (and dirty)

1. the block is cached and dirty ⇒ incoherent with the main memory
2. the peripheral device requests writing the block (11000011)
3. the cache controller stops the writing operation and updates the block in the main memory
4. the writing operation is allowed and the block in the cache is invalidated

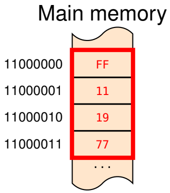


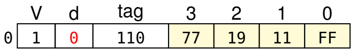
	V	d	tag	3	2	1	0
0	1	1	110	77	19	11	FF

I/O interface writing + *write-back*

The block is cached (and dirty)

1. the block is cached and dirty ⇒ incoherent with the main memory
2. the peripheral device requests writing the block (11000011)
3. the cache controller stops the writing operation and updates the block in the main memory
4. the writing operation is allowed and the block in the cache is invalidated



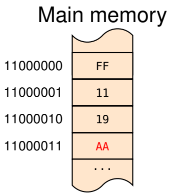


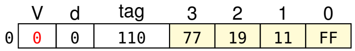
	V	d	tag	3	2	1	0
0	1	0	110	77	19	11	FF

I/O interface writing + *write-back*

The block is cached (and dirty)

1. the block is cached and dirty ⇒ incoherent with the main memory
2. the peripheral device requests writing the block (11000011)
3. the cache controller stops the writing operation and updates the block in the main memory
4. the writing operation is allowed and the block in the cache is invalidated





Cache organization

- Number of cache levels
- Type of information stored

Cache levels

Problem
Large performance gap between the cache and the main memory

Solution
Tradeoff between cache latency and hit rate
Several levels of cache are used $\Rightarrow$ cut down the performance penalty

- called L1, L2, L3, etc.
- intermediate speed and capacity
- usually, three levels

Type of information

Unified cache

A single cache stores all types of information

Separated caches

Data cache and instruction cache

Which one provides better performance?

Type of information

Unified cache

A single cache stores all types of information

- ✔ simple, a single piece of hardware
- ✔ leverages the blocks
- ✔ higher hit rate

Separated caches

Data cache and instruction cache

- ✘ replicated hardware
- ✘ fixed amount of cache blocks per type
- ✘ lower hit rate
- ✔ concurrent accesses ⇒ L1 is usually separated

Example: Intel Core i7 2700K

L1 separated cache: data (4x) and code (4x)

- $4 \times 32\,\text{KB}$
- 64 words per block (64 bytes)
- 8 ways

Unified L2 cache (4x)

- $4 \times 256\,\text{KB}$
- 64 words per block (64 bytes)
- 8 ways

Unified L3 cache

- 8 MB
- 64 words per block (64 bytes)
- 16 ways

Example: Intel Core i7 2700K

L1 separated cache: data (4x) and code (4x)

- 4×32 KB
- 64 words per block (64 bytes)
- 8 ways

64 sets

Unified L2 cache (4x)

- 4×256 KB
- 64 words per block (64 bytes)
- 8 ways

512 sets

Unified L3 cache

- 8 MB
- 64 words per block (64 bytes)
- 16 ways

$2^{13} = 8192$ sets