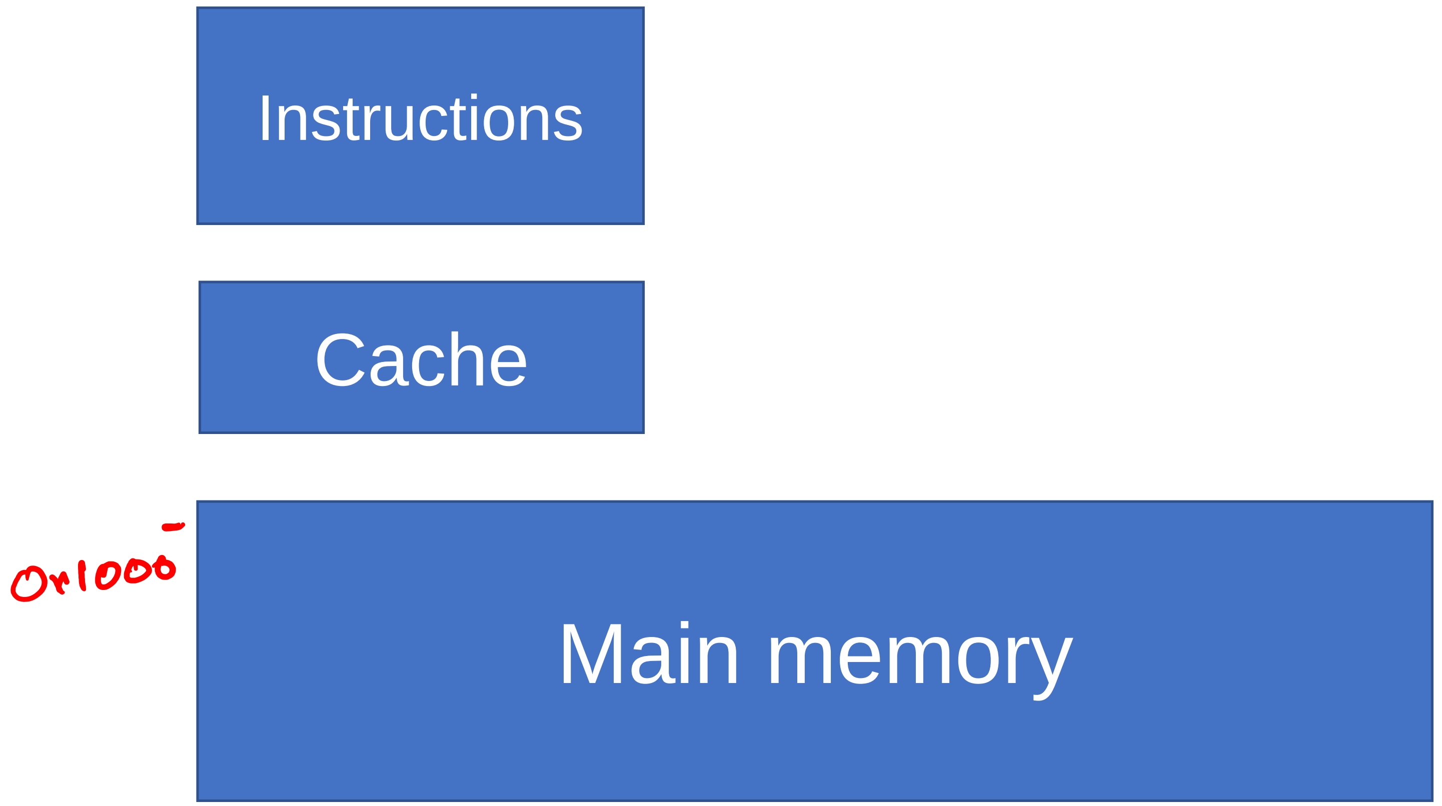
Cache

0x1000
↓
mov x, %eax



Cache

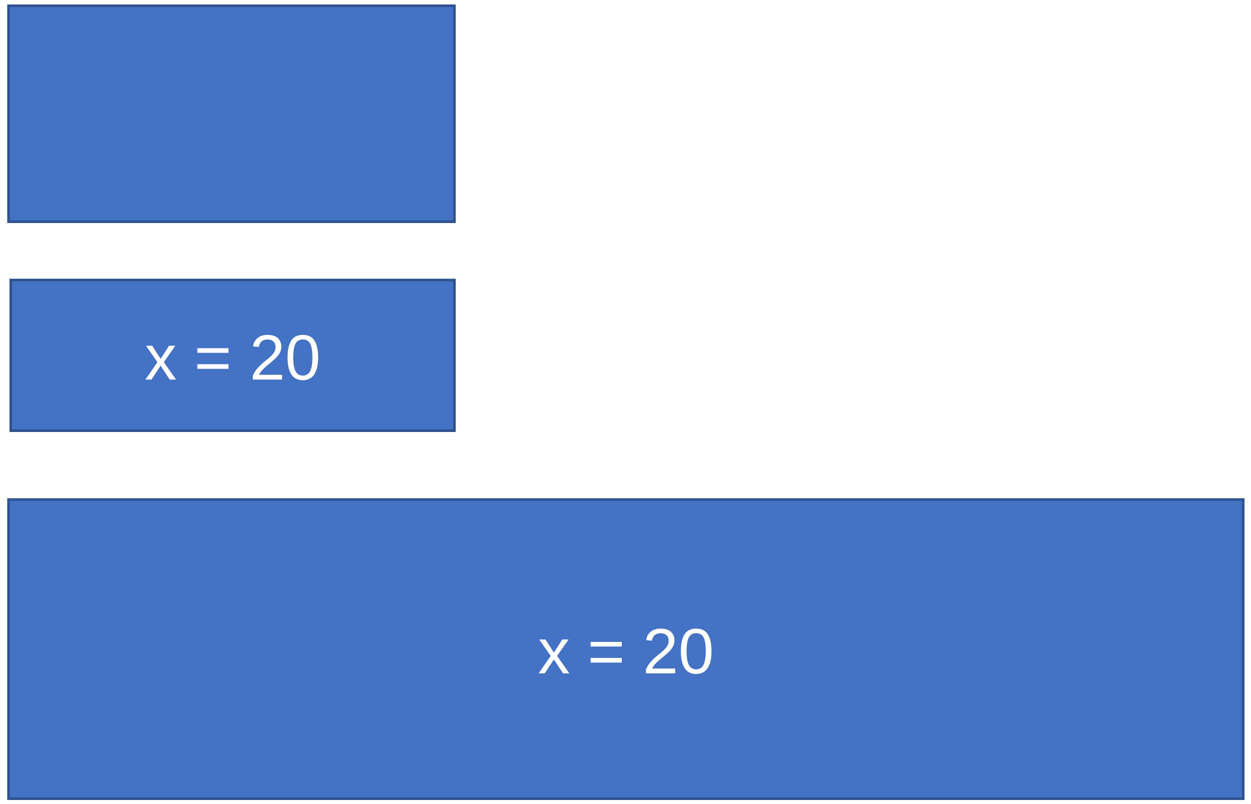
- Cache is a fast storage media between RAM and CPU
  - Cache is ten times faster than RAM

- data is brought to cache before it can be read/written by the instructions

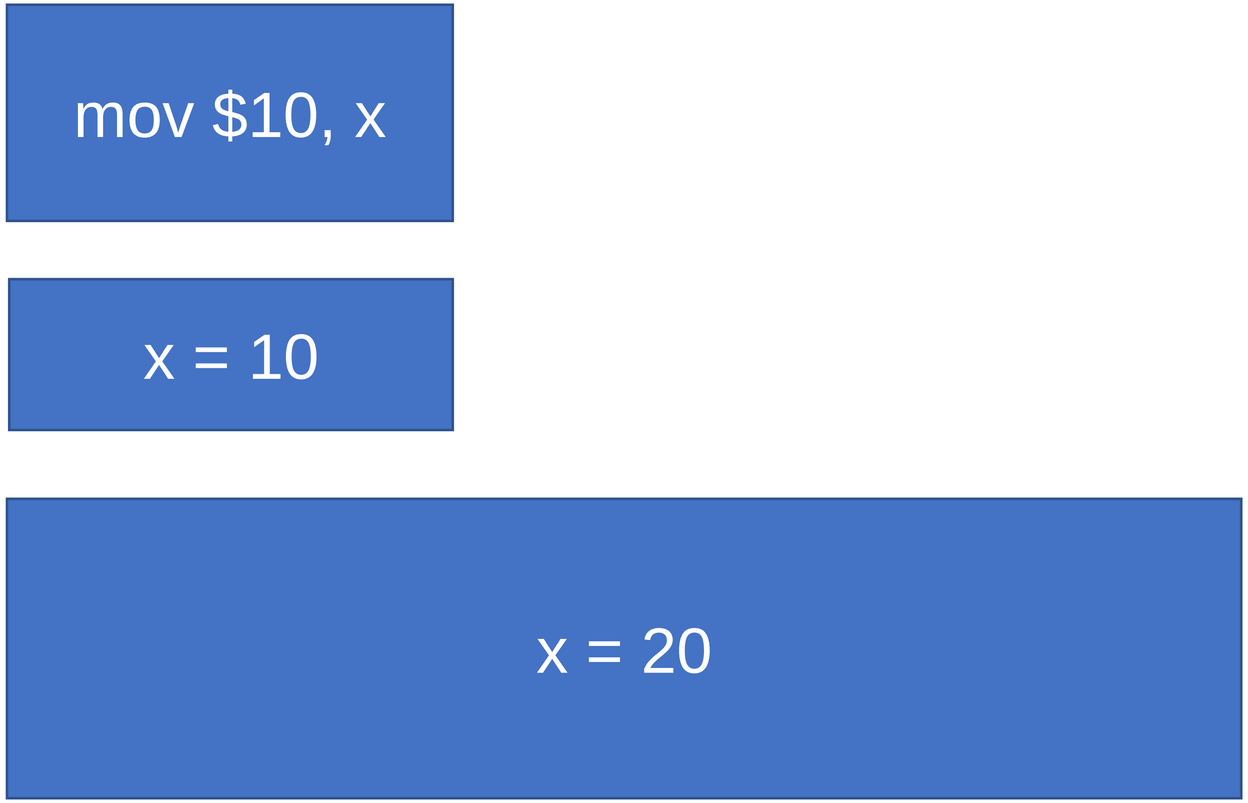
- typical cache line size is 64 bytes

- an entire cache line is brought to the cache when a memory byte is accessed

Write-back cache



Write-back cache



When a cache line is written to main memory?

- Asynchronous
    - At the time of replacement

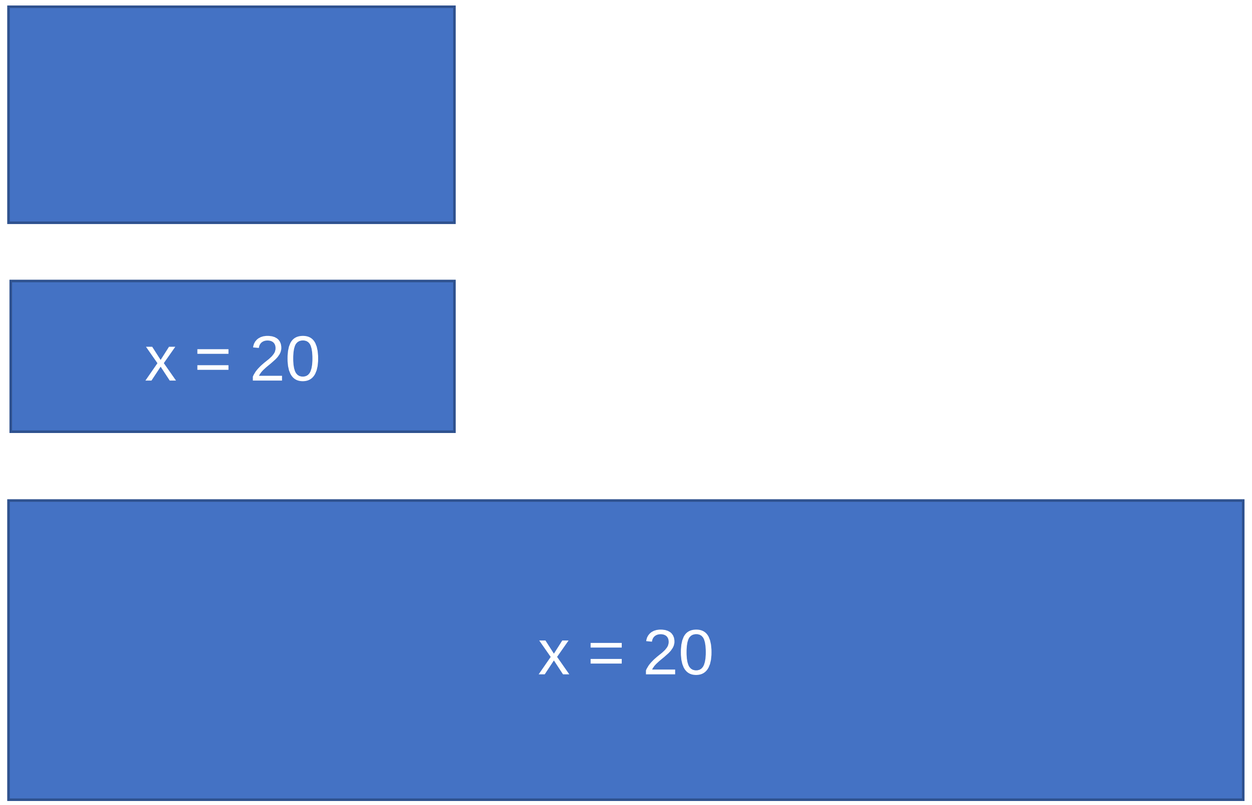
How a memory cache write-back is different from buffer cache write-back?

buffer cache

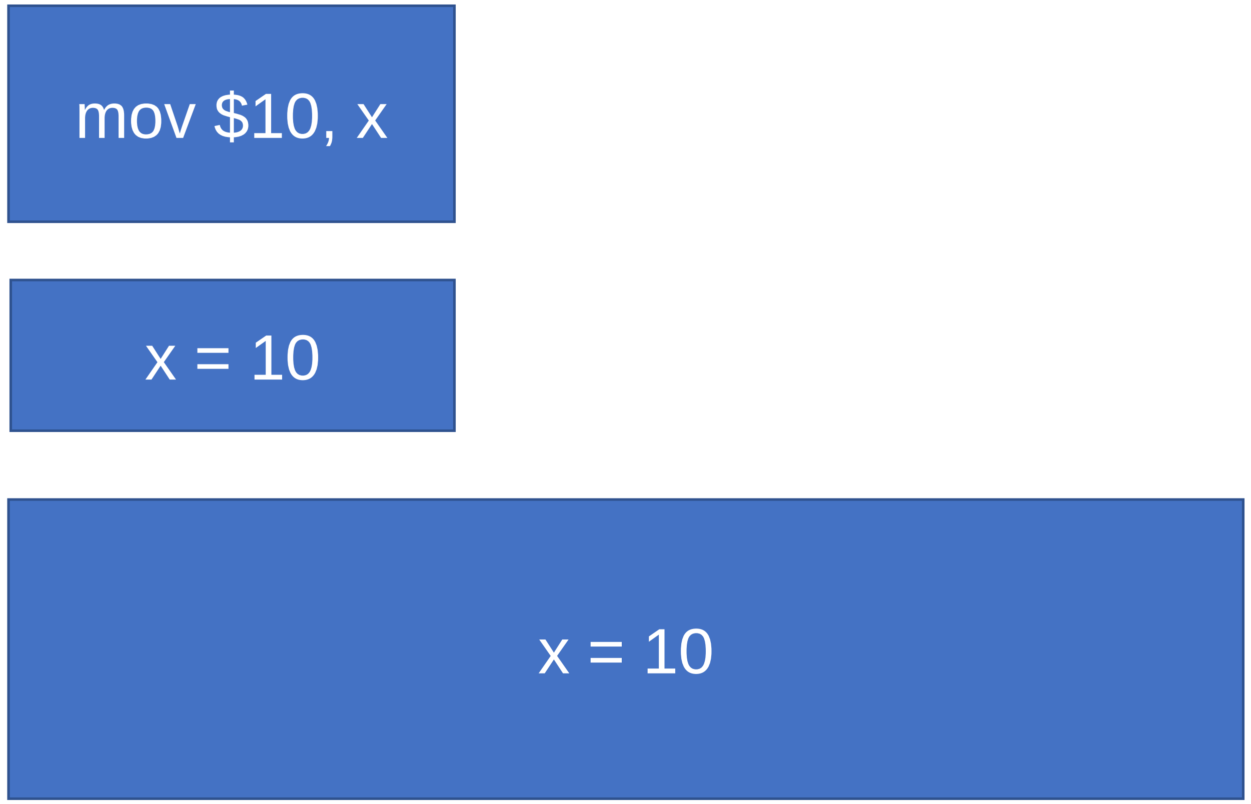
- write-back in buffer cache does not necessarily mean replacement
    - write-back happen during commit

- The commit policy also depends upon how an application interacts with the user

Write-through cache



Write-through cache



Disadvantage of write-through

- Every write goes to RAM
    - slow

Advantages of write-through

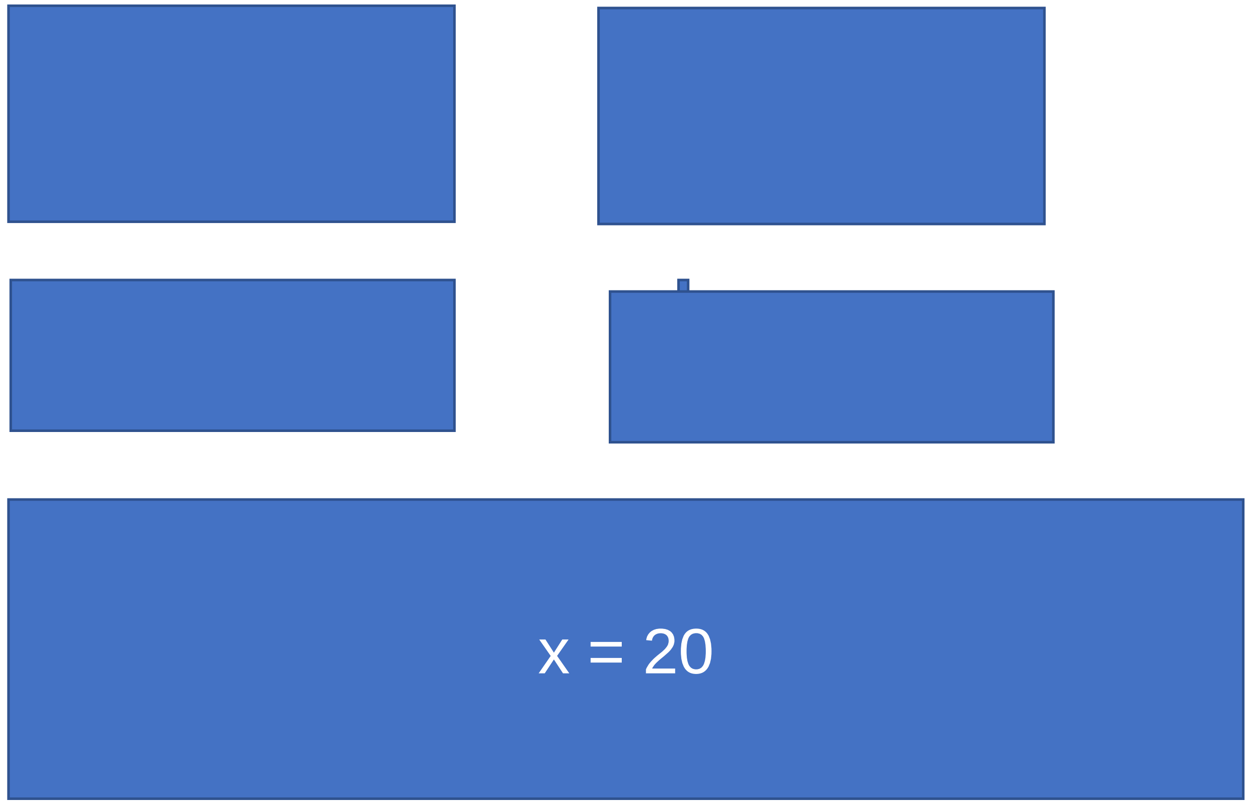
Advantages of write-through

- Cache coherence hardware is simple

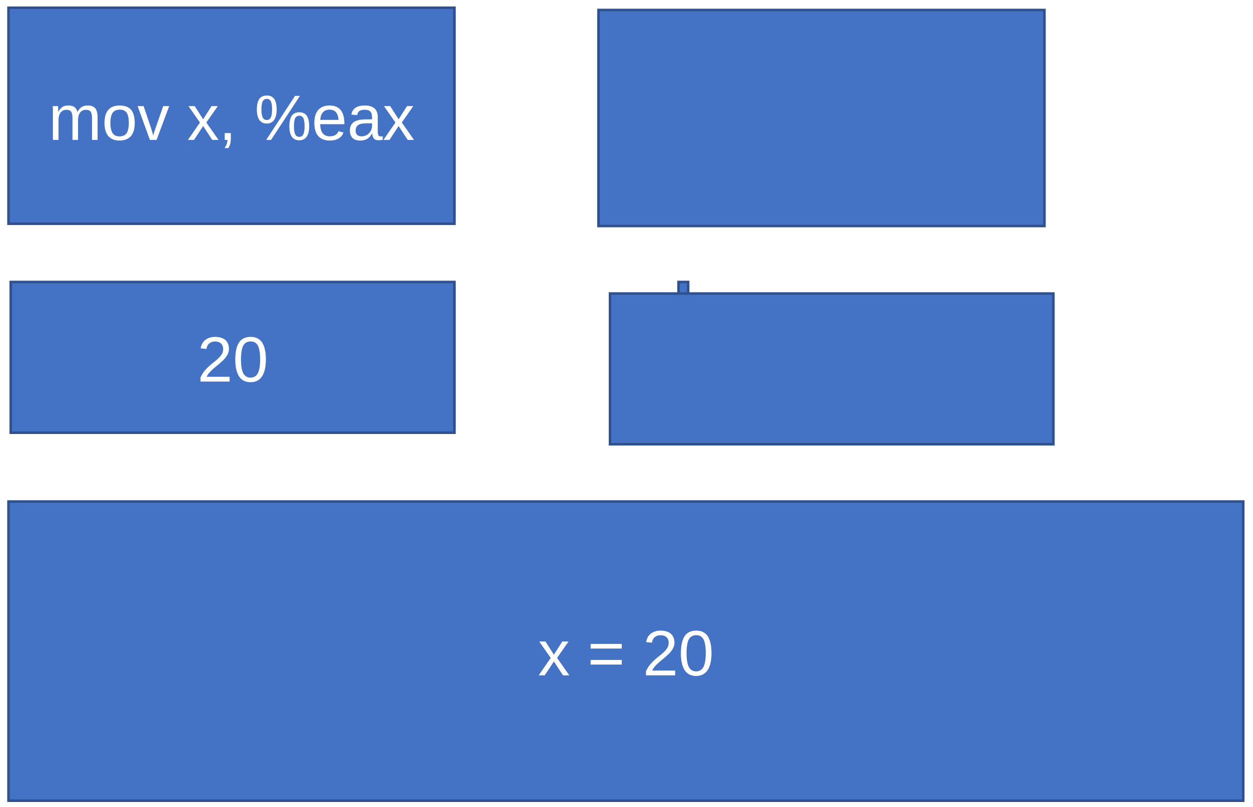
Can directly read write with memory

- Memory mapped devices require write-through memory
  - Page table entry contains a write-through flag

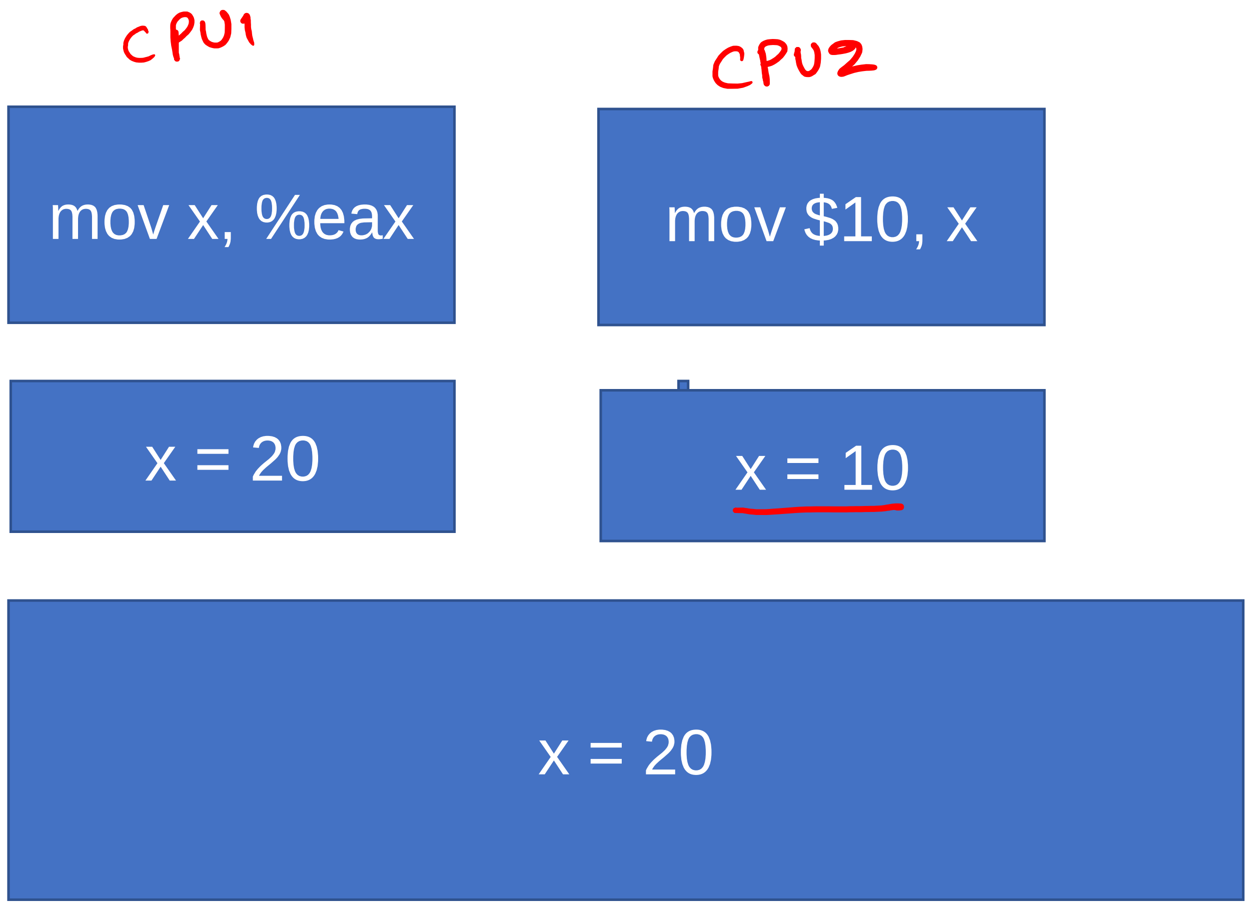
Write-back cache



Write-back cache



Write-back cache



Cache coherence

- A cache coherence protocol ensures the consistency of shared data among multiple per-processor caches

MESI

- A cache line can be in any of these states

- Modified (M): doesn’t match main memory and only present in current cache
- Exclusive (E): matches main memory and only present in current cache
- Shared (S): matches main memory and may present in multiple private caches
- Invalid (I): this cache line is invalid

MESI

- For a pair of caches, the permitted states of a give cache line as follows:

	M	E	S	I
M	x	x	x	y
E	x	x	x	y
S	x	x	y	y
I	y	y	y	y

MESI

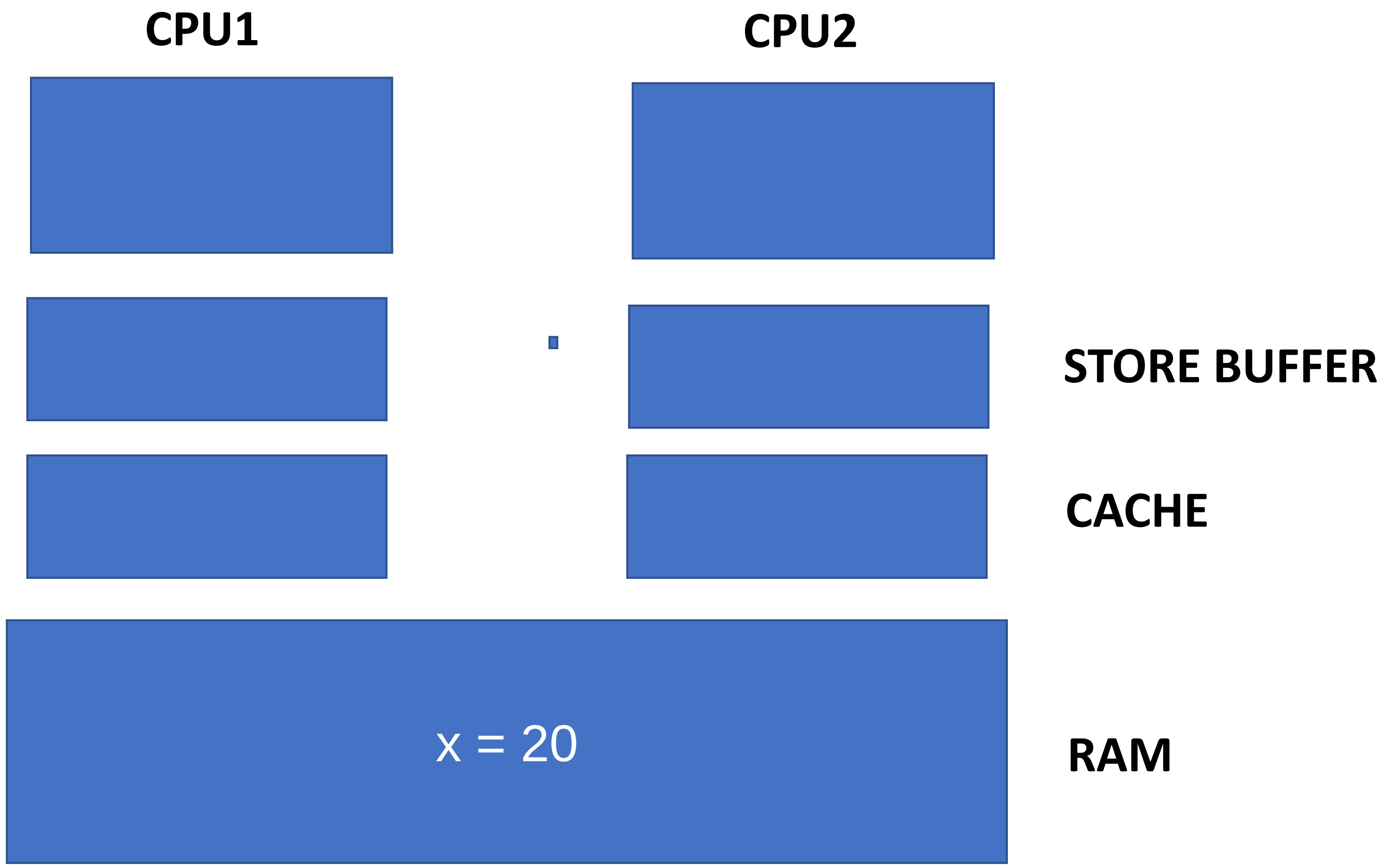
REQ	P1	P2	P3
Initially	NP	NP	NP
R1	E	NP	NP
W1	M	NP	NP
R3	S	NP	S
W3	I	NP	M
R1	S	NP	S
R3	S	NP	S
R2	S	S	S

write back

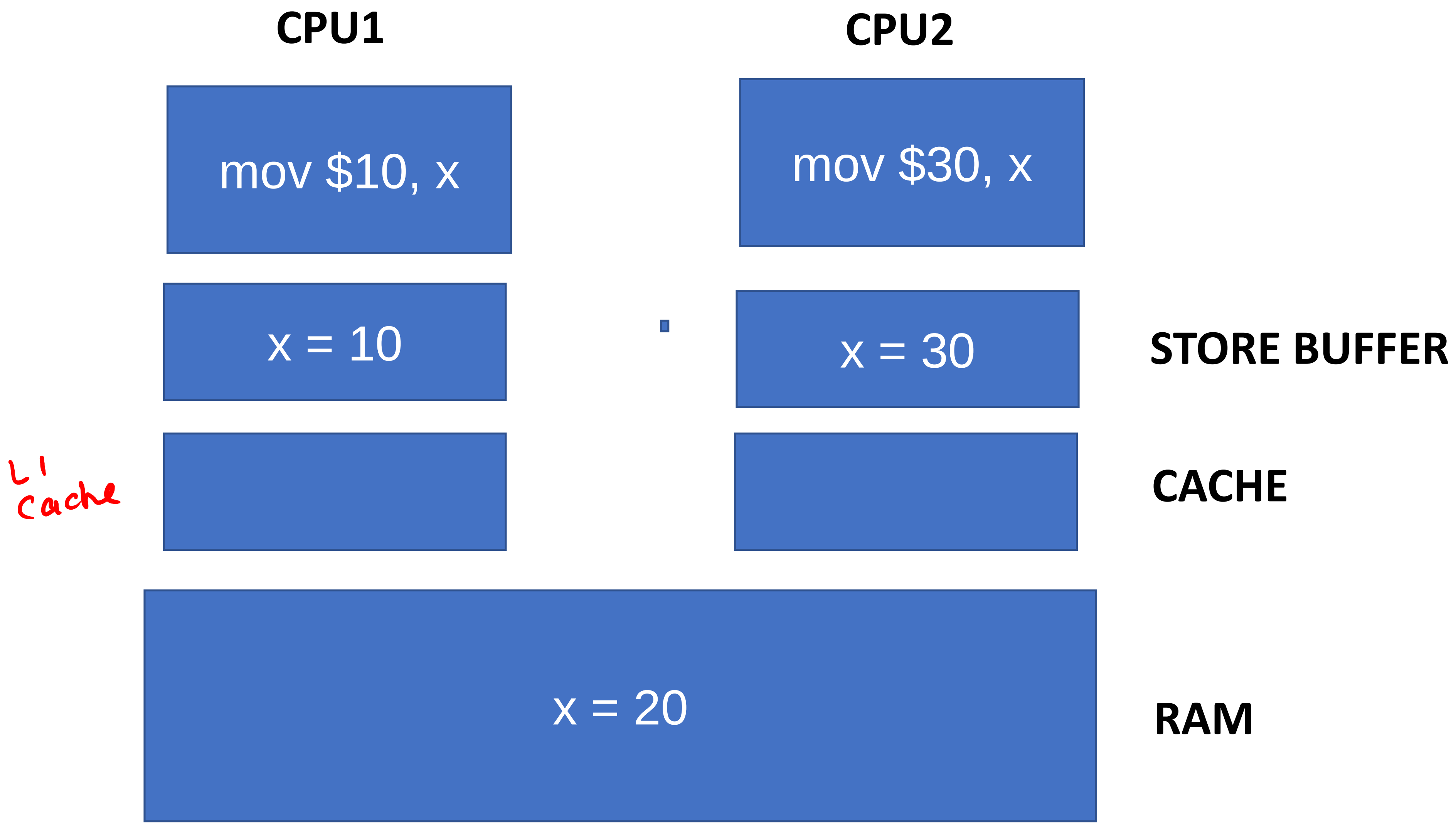
write back

M	Modified
E	Exclusive
S	Shared
I	Invalid

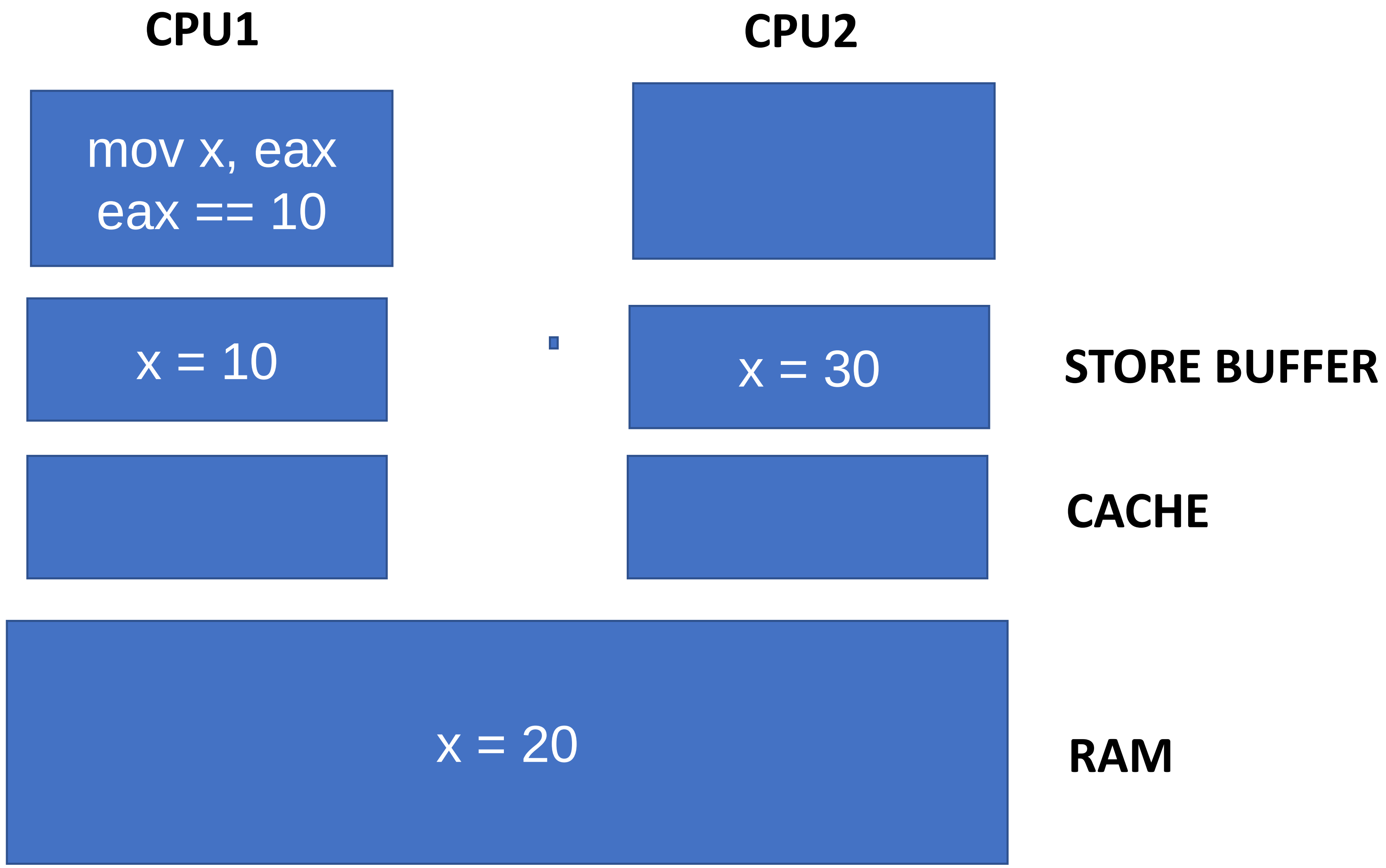
Store buffer



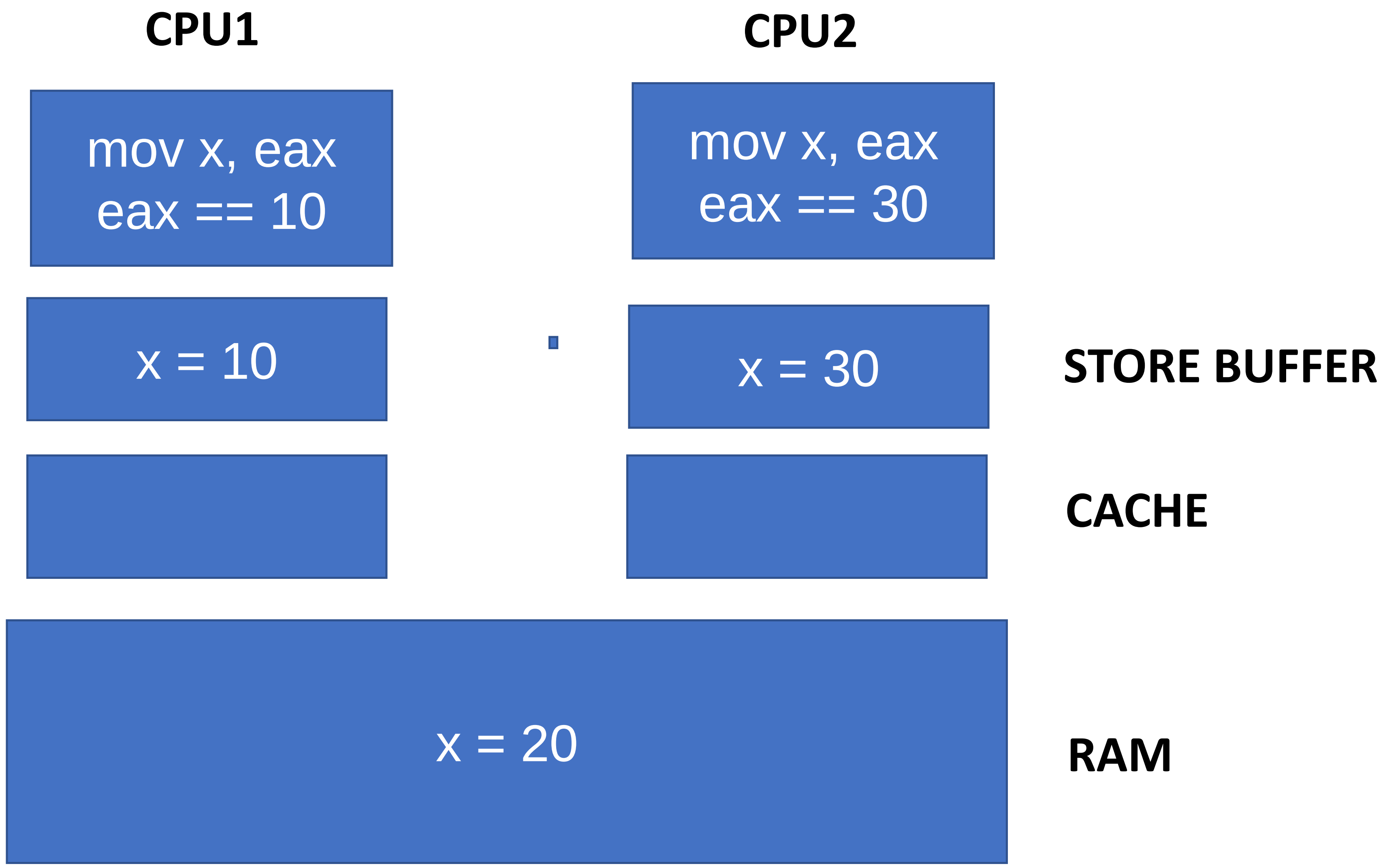
Store buffer



Store buffer



Store buffer



Store buffer

- No cache coherency at the store buffer granularity

- reads from the same core, which are present in the store buffer, are served from the store buffer

Memory ordering

- load and store are atomic operations on x86

- Memory ordering refers to the order in which the processor issues read and write request to the system memory

$x = 1$ $x = 2$

$x = 1 \mid 2$

Memory ordering

P0	P1
✓ x = 1	y = 1 ✓
✓ r1 = y	r2 = x ✓

Initially, x and y are 0.

$$\boxed{\begin{array}{l} x = 1 \\ r1 = y \\ y = 1 \\ r2 = x \end{array}}$$

x = 1
y = 1
r1 = y
r2 = x

$$\boxed{\begin{array}{l} r1 = y \\ x = 1 \\ y = 1 \\ r2 = x \end{array}}$$

on x86, loads and stores are not reordered with like operations

P0	P1
x = 1	r1 = y
y = 1	r2 = x

r1 == 1
r2 == 0

Initially, x and y are 0.

Stores are not reordered with earlier loads

P0	P1
r1 = x	r2 = y
y = 1	x = 1

r2 == 1 && r1 == 1

Initially, x and y are 0.

Loads may be reordered with earlier stores to a different memory location

P0	P1
x = 1 mfence	y = 1
r1 = y	r2 = x

r1 == 0 && r2 == 0

Initially, x and y are 0.

Memory reordering

- On x86, a load may be reordered with an older stores to a different memory location
- a load may not be reordered with a locked instruction

For details you can refer to Chapter-11 of **Intel software developers manual vol-3**

How to prevent reordering?

- memory barrier (mfence) on x86
- a memory barrier pause execution until all the loads and stores prior to memory barrier are completed
- the store buffer is drained on memory barrier

Locking on multiprocessor

- lock_acquire
- lock_release

Peterson's solution

a load can be reordered with earlier stores on the different memory loc

```
volatile int turn;
volatile boolean flag[2];
```

acquire: flag[j] = False

```
flag[i] = TRUE;
turn = j;
while (flag[j] && turn == j);
```

mfence

release:

```
flag[i] = FALSE;
```

store flag[i]
load j
store turn
load flag[j]
load turn
load j

load j
load flag[j]
store flag[i]
store turn

load j
store flag[i]
store turn
load flag[j]

Peterson’s solution

- Does not work on multiprocessor because of reordering

```
volatile int turn;
volatile boolean flag[2];

acquire:
flag[i] = TRUE;
turn = j;
__sync_synchronize ();
while (flag[j] && turn == j);

release:
flag[i] = FALSE;
```

__sync_synchronize

- is a memory barrier (mfence on x86)
- no reordering across memory barrier
- drains the store buffer
- __sync_synchronize waits for all the outstanding read/write operations to finish

add $1, 0x1000

t = *0x1000
t = t+1
*0x1000 = t

lock prefix

- certain instructions can be prefixed with "lock"
- "lock" prefix ensures the atomicity of instruction
  - x86 instructions are complex, and a single instruction can do multiple reads and writes
  - you can think of "lock" prefix as: it inserts an acquire/release around the instruction
- "lock" prefix works directly on the cache bypassing store buffer
- locked instruction drains the store buffer on current CPU

lock prefix

lock add $1, 0x1000

add $1, 0x1000

0x1000 = 0;

Thread 1:

for 1000 times acquire()

    add $1, 0x1000

    release()

Thread: 2

for 1000 times acquire()

    add $1, 0x1000

    release()

lock prefix

```
add $1, 0x1000
```

```
0x1000 = 0;
```

```
Thread 1:
for 1000 times
   lock add $1, 0x1000
```

```
Thread: 2
for 1000 times
   lock add $1, 0x1000
```