Computer Architecture

Week 12: Cache

Fenerbahçe Üniversitesi

Professor & TAs

Prof: Dr. Vecdi Emre Levent

Office: 311

Email: emre.levent@fbu.edu.tr

TA: Arş. Gör. Uğur Özbalkan

Office: 311

Email: ugur.ozbalkan@fbu.edu.tr

Course Plan

- Cache

Programs 101

Load/Store Architectures:

- Read data from memory (put in registers)
- Manipulate it
- Store it back to memory

C Code

```
int main (int argc, char* argv[ ]) {
   int i;
   int m = n;
   int sum = 0;
   for (i = 1; i <= m; i++) {
      sum += i;
   }
   printf (“...”, n, sum);
}
```

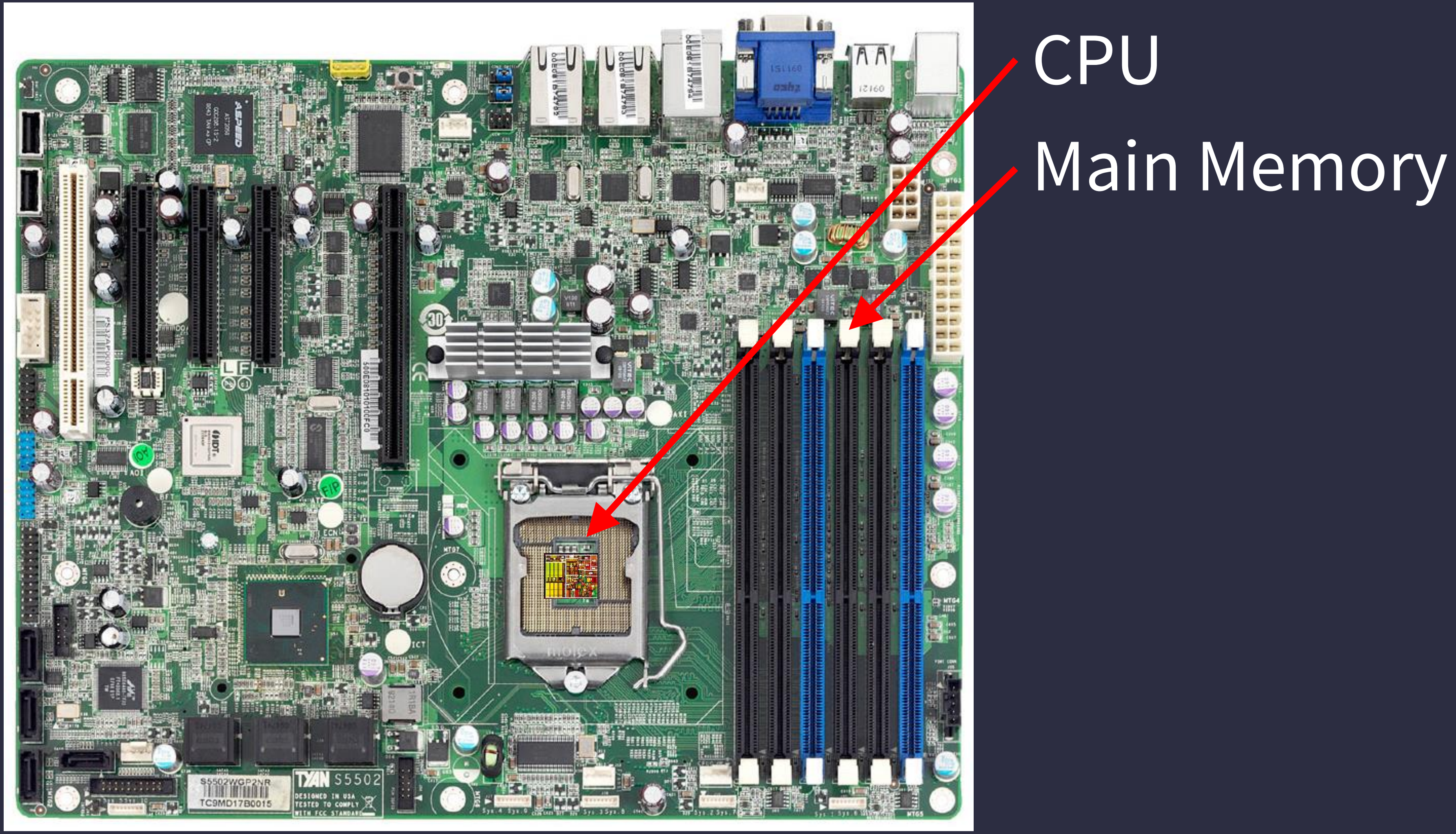
RISC-V Assembly

```
main:   addi    sp,sp,-48
        sw      x1,44(sp)
        sw      fp,40(sp)
        move    fp,sp
        sw      x10,-36(fp)
        sw      x11,-40(fp)
        la      x15,n
        lw      x15,0(x15)
        sw      x15,-28(fp)
        sw      x0,-24(fp)
        li      x15,1
        sw      x15,-20(fp)
 L2:    lw      x14,-20(fp)
        lw      x15,-28(fp)
        blt     x15,x14,L3
        . . .
```

■ **Instructions that read from or write to memory…**

What’s the problem?

- + big
- – slow
- – far away



The Need for Speed

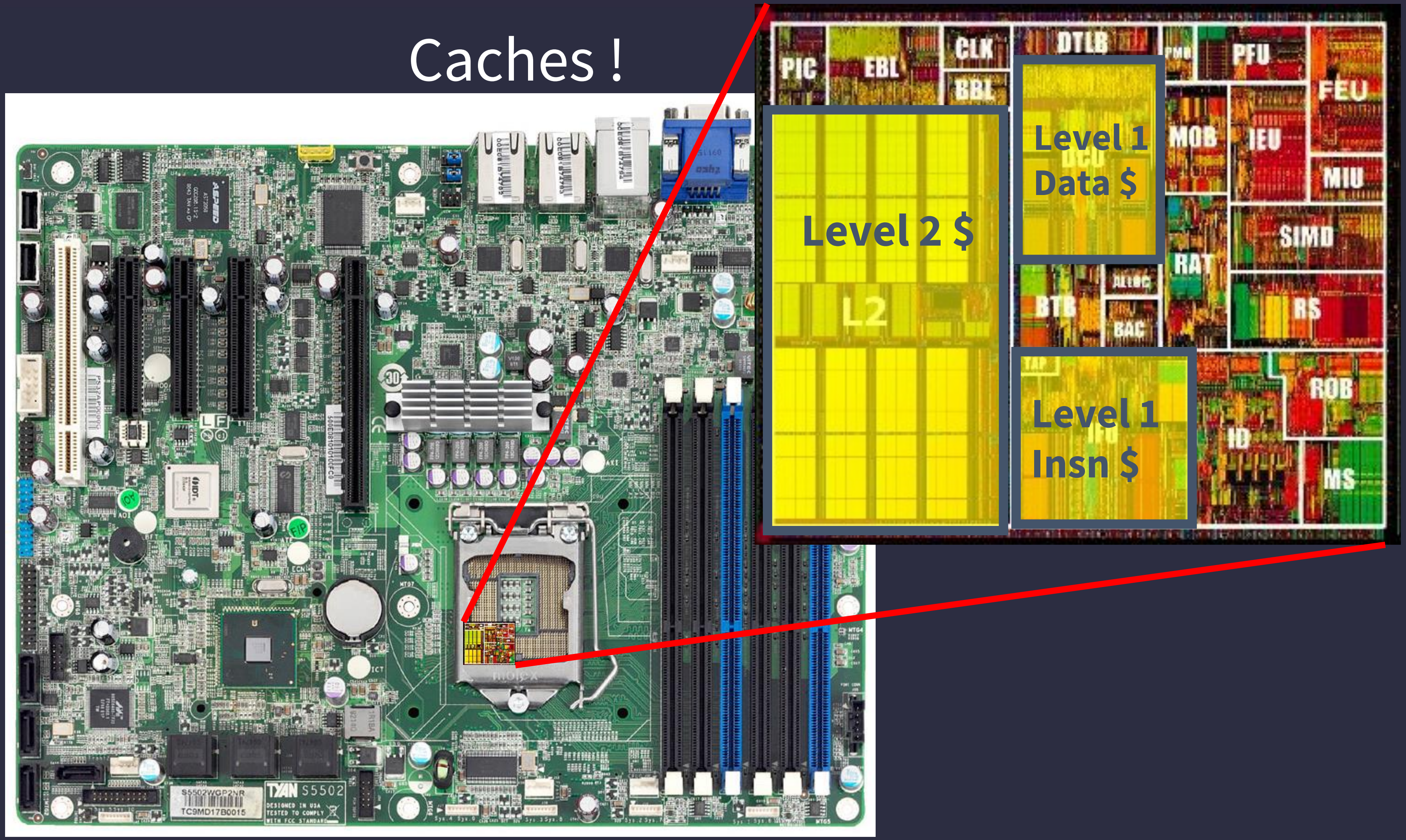
Instruction speeds:

- `add,sub,shift`: 1 cycle
- `mult`: 3 cycles
- `load/store`: **100 cycles**

  (2 GHz processor → 0.5 ns clock, off-chip 50 ns)

What’s the solution?

Caches !



Locality Locality Locality

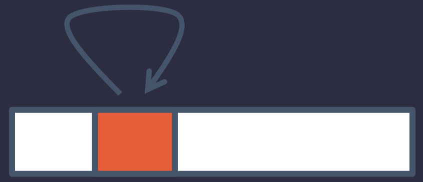
- the same thing again soon

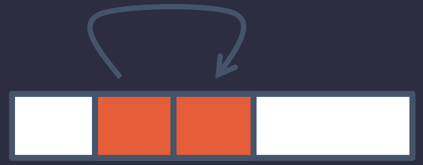
  → Temporal Locality

- something near that thing, soon

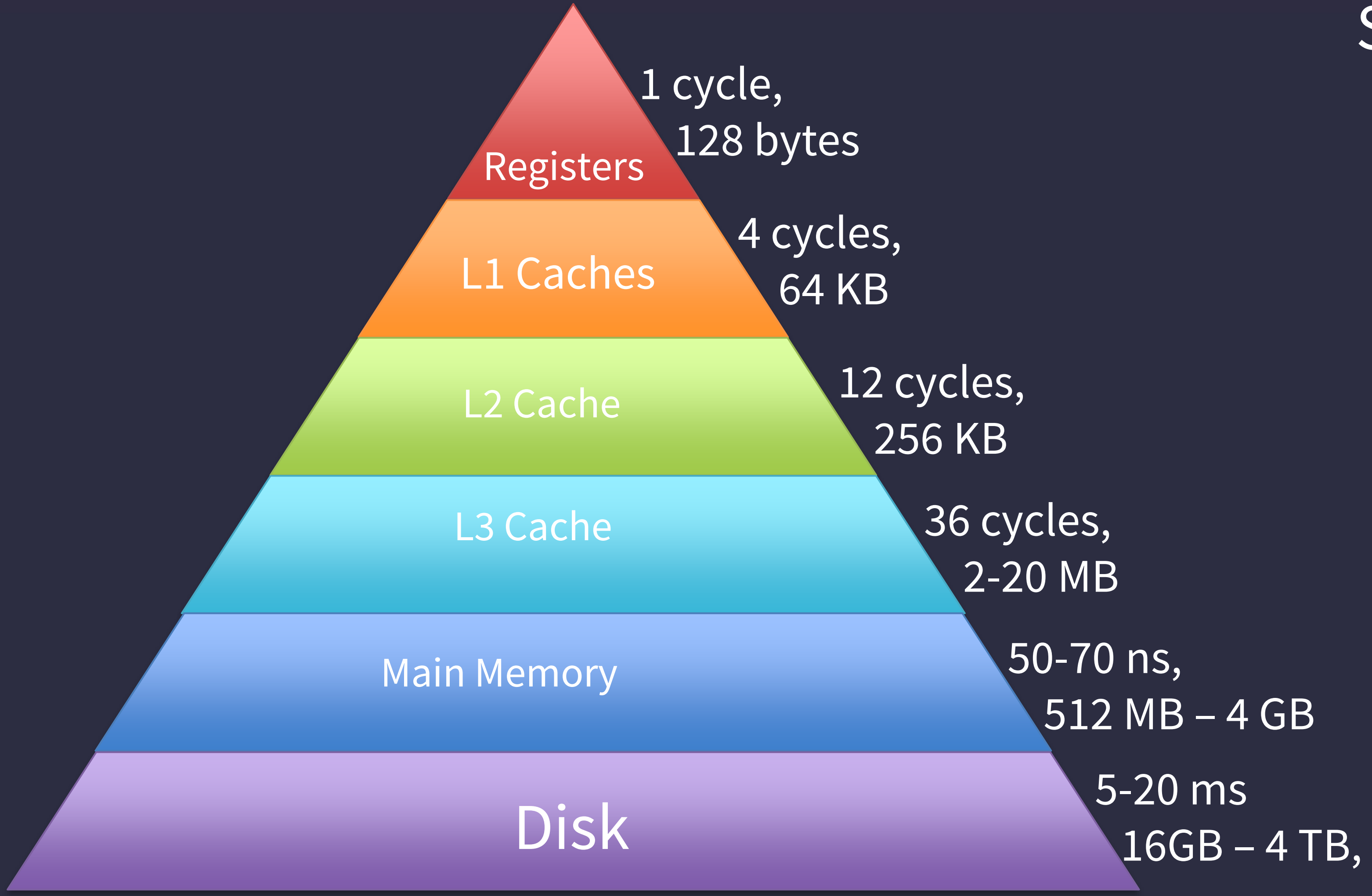
  → Spatial Locality

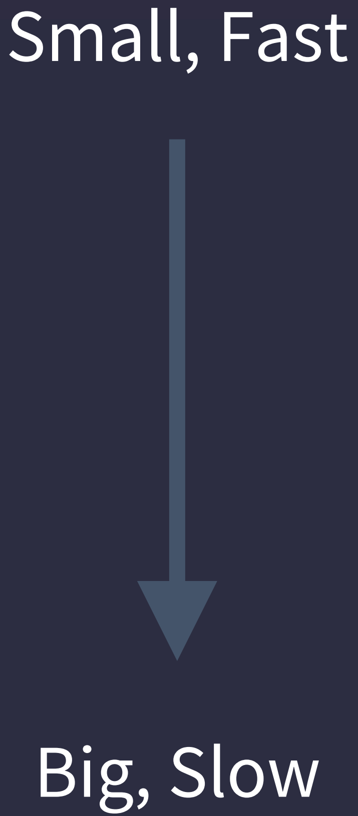
```
total = 0;
for (i = 0; i < n; i++)
    total += a[i];
return total;
```





The Memory Hierarchy





Some Terminology

Cache hit

- data is in the Cache
- t_{hit} : time it takes to access the cache
- Hit rate (%hit): # cache hits / # cache accesses

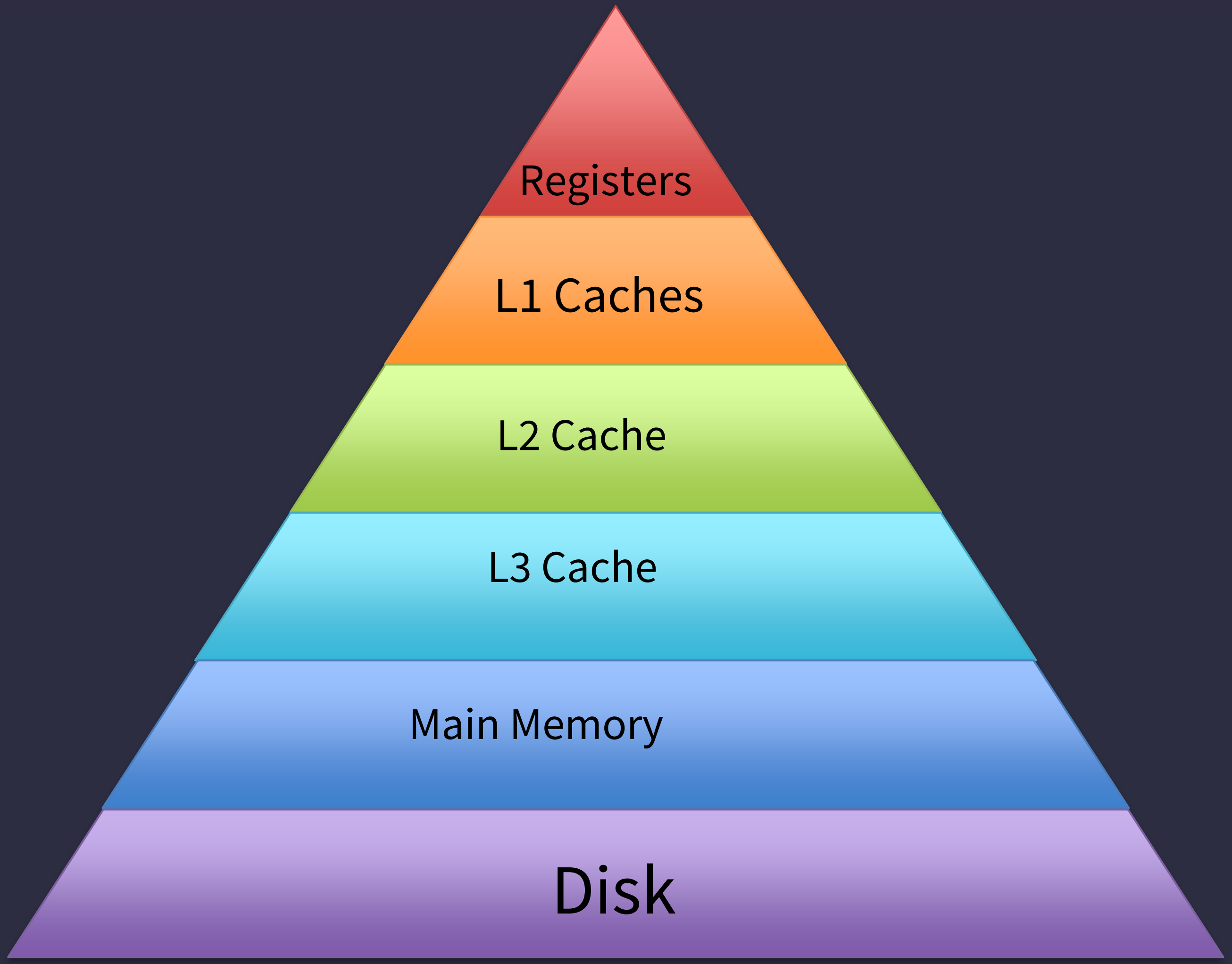
Cache miss

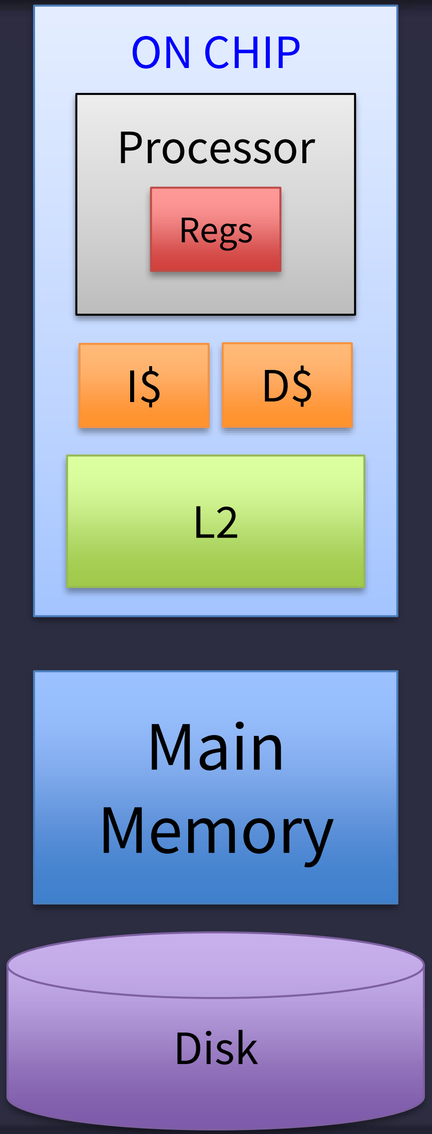
- data is **not** in the Cache
- t_{miss} : time it takes to get the data from below the $
- Miss rate (%miss): # cache misses / # cache accesses

Cacheline or cacheblock or simply line or block

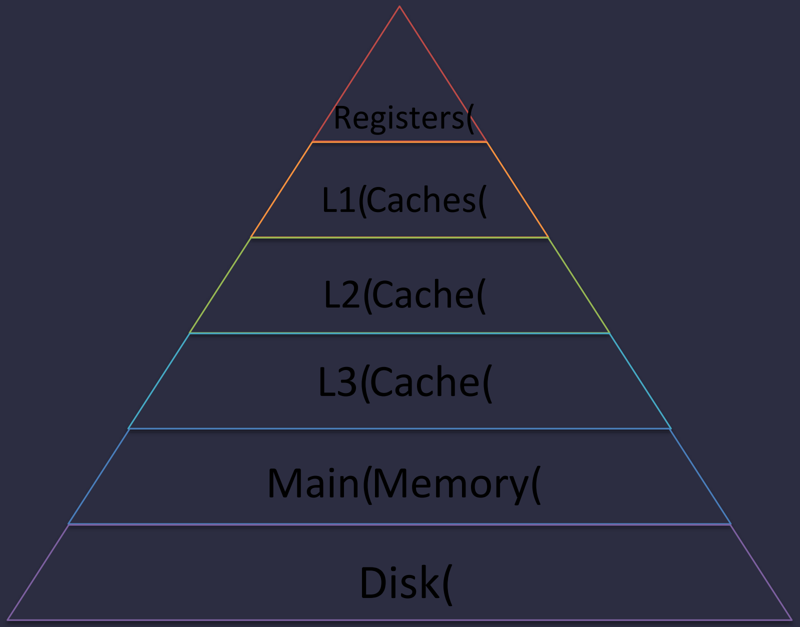
- Minimum unit of info that is present/or not in the cache

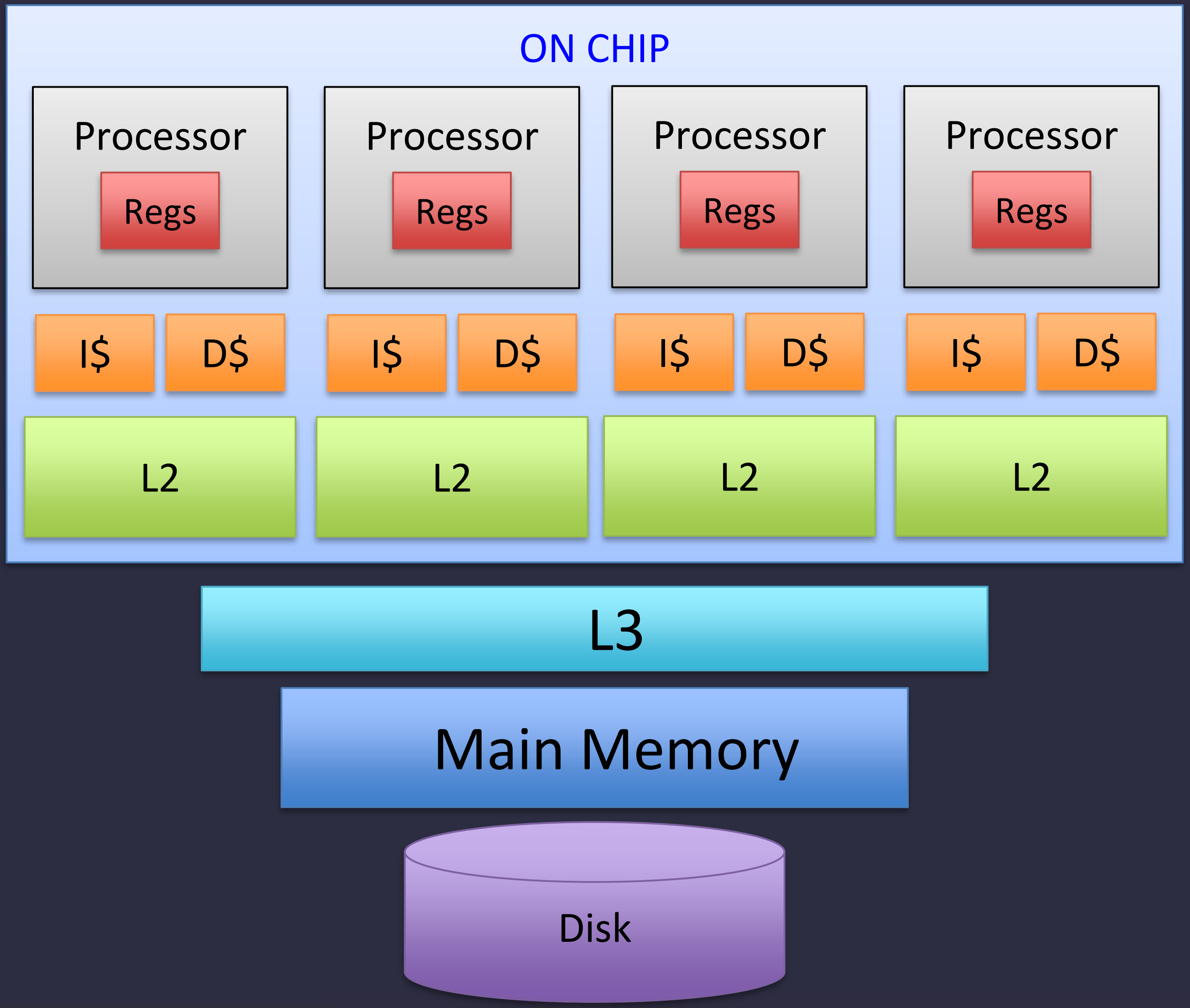
Single Core Memory Hierarchy





Multi-Core Memory Hierarchy





Memory Hierarchy by the Numbers

CPU clock rates ~0.33ns – 2ns (3GHz-500MHz)

Memory technology	Transistor count	Access time	Access time in cycles	$ per GB in 2021	Capacity
SRAM (on chip)	6-8 transistors	0.5-2.5 ns	1-3 cycles	$4k	256 KB
SRAM (off chip)		1.5-30 ns	5-15 cycles	$4k	32 MB
DRAM	1 transistor (needs refresh)	50-70 ns	150-200 cycles	$10-$20	8 GB
SSD (Flash)		5k-50k ns	Tens of thousands	$0.75-$1	512 GB
Disk		5M-20M ns	Millions	$0.05-$0.1	4 TB

Basic Cache Design

Direct Mapped Caches

16 Byte Memory

```
load   1100 → r1
```

- Byte-addressable memory
- 4 address bits → 16 bytes total
- b addr bits → 2^b bytes in memory

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

4-Byte, Direct Mapped Cache

CACHE

index
xxxx

index	**data**
00	A
01	B
10	C
11	D

← Cache entry
= row
= **(cache) line**
= **(cache) block**
Block Size: 1 byte

MEMORY

addr	**data**
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Direct mapped:

- Each address maps to 1 cache block
- 4 entries → 2 index bits (2^n → n bits)

4-Byte, Direct Mapped Cache

```
tag|index
  xxxx
```

CACHE

index	tag	data
00	00	A
01	00	B
10	00	C
11	00	D

Tag: minimalist label/address

`address = tag + index`

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

4-Byte, Direct Mapped Cache

CACHE

index	V	tag	data
00	0	00	X
01	0	00	X
10	0	00	X
11	0	00	X

One last tweak: **valid bit**

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #1 of a 4-byte, DM Cache

tag | index
xxxx

CACHE

index	V	tag	data
00	1	11	N
01	0	xx	X
10	0	xx	X
11	0	xx	X

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

`load 1100 Miss`

Lookup:

- Index into $
- Check tag
- Check valid bit

Simulation #1 of a 4-byte, DM Cache

tag|index
XXXX

CACHE

index	V	tag	data
00	1	11	N
01	0	xx	X
10	0	xx	X
11	0	xx	X

```
load    1100    Miss
. . .
load    1100    Hit!
```

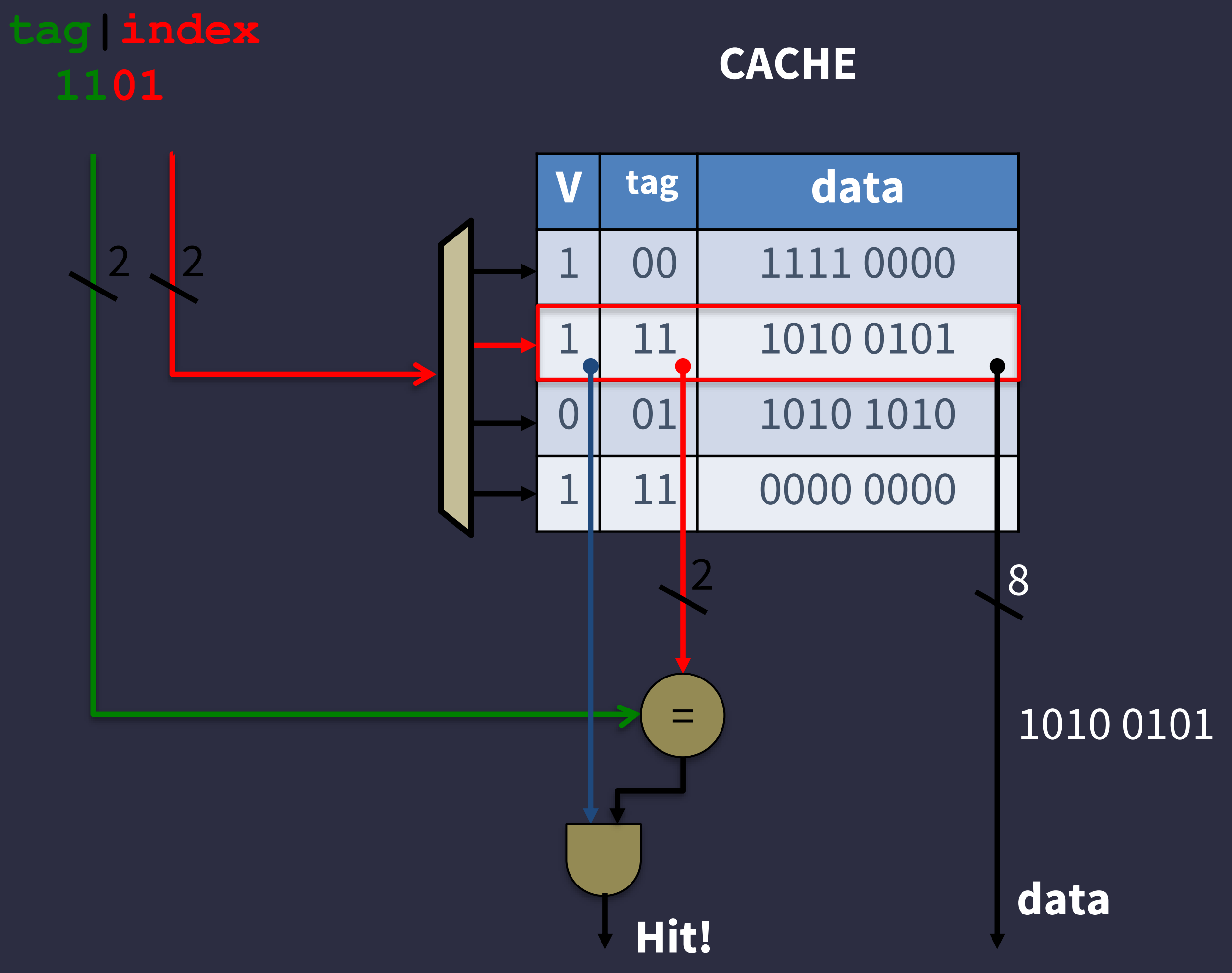
Awesome!

Lookup:
- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Block Diagram 4-entry, direct mapped Cache



V	tag	data
1	00	1111 0000
1	11	1010 0101
0	01	1010 1010
1	11	0000 0000

Great!
Are we done?

Simulation #2: 4-byte, DM Cache

CACHE

index	V	tag	data
00	0	11	X
01	0	11	X
10	0	11	X
11	0	11	X

```
load    1100    Miss
load    1101
load    0100
load    1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #2: 4-byte, DM Cache

CACHE

index	V	tag	data
00	1	11	N
01	0	11	X
10	0	11	X
11	0	11	X

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

```
load    1100    Miss
load    1101
load    0100
load    1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

Simulation #2: 4-byte, DM Cache

CACHE

index	V	tag	data
00	1	11	N
01	0	11	X
10	0	11	X
11	0	11	X

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

```
  load    1100    Miss
➡ load    1101
  load    0100
  load    1100
```

Lookup:

➡ Index into $

➡ Check tag

➡ Check valid bit

Simulation #2: 4-byte, DM Cache

tag | index
XXXX

CACHE

index	V	tag	data
00	1	11	N
01	1	11	O
10	0	11	X
11	0	11	X

```
load   1100   Miss
load   1101   Miss
load   0100
load   1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #2: 4-byte, DM Cache

CACHE

index	V	tag	data
00	1	11	N
01	1	11	O
10	0	11	X
11	0	11	X

```
    load    1100    Miss
    load    1101    Miss
➡   load    0100    Miss
    load    1100
```

Lookup:

- ➡ Index into $
- ➡ Check tag
- ➡ Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #2: 4-byte, DM Cache

tag | index
XXXX

CACHE

index	V	tag	data
00	1	01	E
01	1	11	O
10	0	11	X
11	0	11	X

```
load    1100    Miss
load    1101    Miss
load    0100    Miss
load    1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #2: 4-byte, DM Cache

CACHE

index	V	tag	data
00	1	01	E
01	1	11	O
10	0	11	X
11	0	11	X

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

```
  load   1100   Miss
  load   1101   Miss
  load   0100   Miss
➡ load   1100   Miss
```

Lookup:

➡ Index into $

➡ Check tag

➡ Check valid bit

Simulation #2: 4-byte, DM Cache

tag | index
xxxx

CACHE

index	V	tag	data
00	1	11	N
01	1	11	O
10	0	11	X
11	0	11	X

```
load    1100   Miss
load    1101   Miss
load    0100   Miss
load    1100   Miss
```

Disappointed!

☹

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Reducing Misses by Increasing Block Size

- Leveraging Spatial Locality

Increasing Block Size

CACHE

offset

xxxx

index	V	tag	data
00	0	x	A | B
01	0	x	C | D
10	0	x	E | F
11	0	x	G | H

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

- **Block Size:** 2 bytes
- **Block Offset:** least significant bits indicate where you live in the block
- Which bits are the index? tag?

Simulation #3: 8-byte, DM Cache

tag | index | offset

xxxx

CACHE

index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	0	x	X | X
11	0	x	X | X

```
load    1100    Miss
load    1101
load    0100
load    1100
```

Lookup:

- ⇨ Index into $
- ⇨ Check tag
- ⇨ Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #3: 8-byte, DM Cache

tag | index | offset
xxxx

CACHE

index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	1	1	N | O
11	0	x	X | X

```
load   1100   Miss
load   1101
load   0100
load   1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #3: 8-byte, DM Cache

tag| index |offset
xxxx

CACHE

index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	1	1	N | O
11	0	x	X | X

```
load    1100  Miss
load    1101  Hit!
load    0100
load    1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #3: 8-byte, DM Cache

tag | index | offset
XXXX

CACHE

index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	1	1	N | O
11	0	x	X | X

```
load    1100    Miss
load    1101    Hit!
load    0100    Miss
load    1100
```

Lookup:

- ➡ Index into $
- ➡ Check tag
- ➡ Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #3: 8-byte, DM Cache

tag | index | offset

xxxx

CACHE

index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	1	0	E | F
11	0	x	X | X

```
load    1100   Miss
load    1101   Hit!
load    0100   Miss
load    1100
```

Lookup:

- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #3: 8-byte, DM Cache

tag| index |offset

xxxx

```
load    1100    Miss
load    1101    Hit!
load    0100    Miss
load    1100    Miss
```

CACHE

index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	1	0	E | F
11	0	x	X | X

Lookup:

- ⇨ Index into $
- ⇨ Check tag
- ⇨ Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #3: 8-byte, DM Cache

CACHE

index	V	tag	data
00	0	x	X | X
01	0	x	X | X
10	1	0	E | F
11	0	x	X | X

load	1100	Miss	cold
load	1101	Hit!	
load	0100	Miss	cold
load	1100	Miss	conflict

1 hit, 3 misses

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Removing Conflict Misses with Fully-Associative Caches

8 byte, fully-associative Cache

xxxx

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
0	xxx	X | X	0	xxx	X | X	0	xxx	X | X	0	xxx	X | X

What should the **offset** be?

What should the **index** be?

What should the **tag** be?

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #4: 8-byte, FA Cache

```
xxxx
tag|offset
```

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
0	xxx	X | X	0	xxx	X | X	0	xxx	X | X	0	xxx	X | X

```
load    1100        Miss
load    1101
load    0100
load    1100
```

Lookup:

- ~~Index into $~~
- Check tags
- Check valid bits

LRU Pointer

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #4: 8-byte, FA Cache

xxxx
tag|offset

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N | O	0	xxx	X | X	0	xxx	X | X	0	xxx	X | X

```
load    1100      Miss
load    1101      Hit!
load    0100
load    1100
```

Lookup:

- ~~Index into $~~

➡ Check tags

➡ Check valid bits

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #4: 8-byte, FA Cache

```
xxxx
tag|offset
```

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N | O	0	xxx	X | X	0	xxx	X | X	0	xxx	X | X

```
load   1100        Miss
load   1101        Hit!
load   0100        Miss
load   1100
```

Lookup:

- ~~Index into $~~
- ➡ Check tags
- ➡ Check valid bits

⇧ LRU Pointer

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Simulation #4: 8-byte, FA Cache

```
 xxxx
tag|offset
```

CACHE

V	tag	data	V	tag	data	V	tag	data	V	tag	data
1	110	N | O	1	010	E | F	0	xxx	X | X	0	xxx	X | X

```
load    1100        Miss
load    1101        Hit!
load    0100        Miss
load    1100        Hit!
```

Lookup:

- ~~Index into $~~
- ➡ Check tags
- ➡ Check valid bits

⇧ LRU Pointer

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

Pros and Cons of Full Associativity

+ No more conflicts!

+ Excellent utilization!

But either:

Parallel Reads

– lots of reading!

Serial Reads

– lots of waiting

Pros & Cons

	Direct Mapped	Fully Associative
Tag Size	Smaller	Larger
SRAM Overhead	Less	More
Controller Logic	Less	More
Speed	Faster	Slower
Price	Less	More
Scalability	Very	Not Very
# of conflict misses	Lots	Zero
Hit Rate	Low	High

Reducing Conflict Misses with Set-Associative Caches

Not too conflict. Not too slow.

… Just Right!

8 byte, 2-way set associative Cache

xxxx

tag | | offset

index

CACHE

index	V	tag	data	V	tag	data
0	0	xx	E | F	0	xx	N | O
1	0	xx	C | D	0	xx	P | Q

What should the **offset** be?

What should the **index** be?

What should the **tag** be?

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

8 byte, 2-way set associative Cache

```
xxxx
tag| |offset
index
```

CACHE

index	V	tag	data	V	tag	data
0	0	xx	X | X	0	xx	X | X
1	0	xx	X | X	0	xx	X | X

```
load    1100    Miss
load    1101
load    0100
load    1100
```

LRU Pointer

Lookup:

- ➡ Index into $
- ➡ Check tag
- ➡ Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

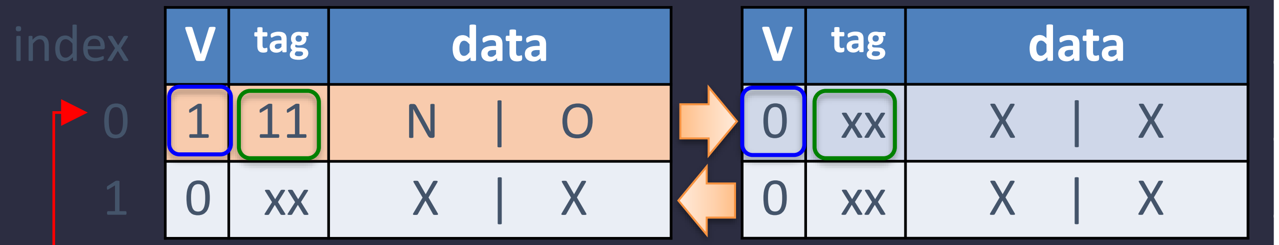
8 byte, 2-way set associative Cache

xxxx

tag | | offset

index

CACHE



index	V	tag	data
0	1	11	N | O
1	0	xx	X | X

V	tag	data
0	xx	X | X
0	xx	X | X

```
load    1100    Miss
load    1101    Hit!
load    0100
load    1100
```

LRU Pointer

Lookup:

- Index into $
- Check tag
- Check valid bit

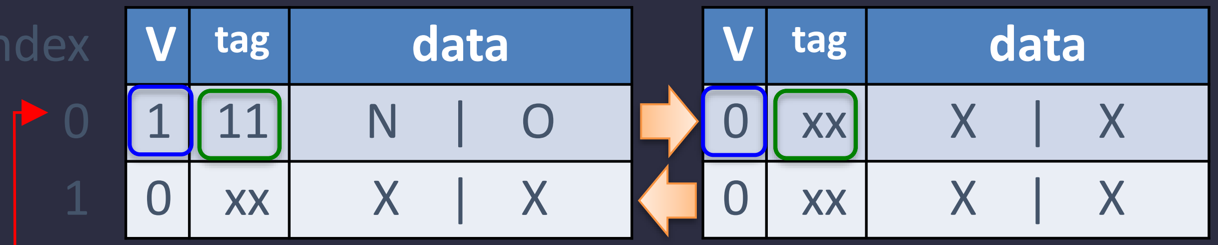
MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

8 byte, 2-way set associative Cache

xxxx
tag||offset
index

CACHE



index	V	tag	data
0	1	11	N | O
1	0	xx	X | X

V	tag	data
0	xx	X | X
0	xx	X | X

```
load    1100     Miss
load    1101     Hit!
load    0100     Miss
load    1100
```

LRU Pointer

Lookup:

- Index into $
- Check tag
- Check valid bit

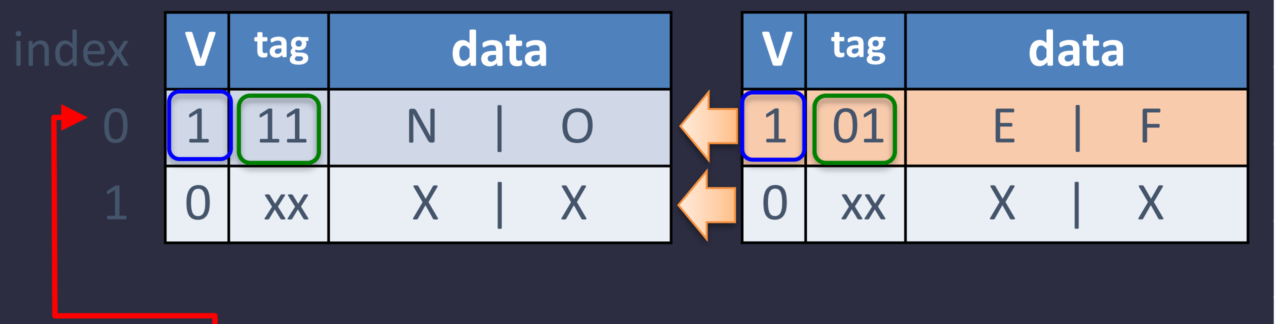
MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

8 byte, 2-way set associative Cache

```
xxxx
tag|index|offset
```

CACHE



index	V	tag	data	V	tag	data
0	1	11	N | O	1	01	E | F
1	0	xx	X | X	0	xx	X | X

```
load    1100    Miss
load    1101    Hit!
load    0100    Miss
load    1100    Hit!
```

LRU Pointer

Lookup:

- Index into $
- Check tag
- Check valid bit

MEMORY

addr	data
0000	A
0001	B
0010	C
0011	D
0100	E
0101	F
0110	G
0111	H
1000	J
1001	K
1010	L
1011	M
1100	N
1101	O
1110	P
1111	Q

24 byte, 3-way set associative Cache

5 bit address
2 byte block size
24 byte, 3-Way Set Associative CACHE

index	V	tag	data	
00	0	?	X	Y
01	0	?	X	Y
10	0	?	X	Y
11	0	?	X	Y

V	tag	data	
0	?	X’	Y’
0	?	X’	Y’
0	?	X’	Y’
0	?	X’	Y’

V	tag	data	
0	?	X’’	Y’’
0	?	X’’	Y’’
0	?	X’’	Y’’
0	?	X’’	Y’’

How many tag bits?

A) 0
B) 1
C) 2
D) 3
E) 4

24 byte, 3-way set associative Cache

5 bit address
2 byte block size
24 byte, 3-Way Set Associative CACHE

index	V	tag	data
00	0	?	X | Y
01	0	?	X | Y
10	0	?	X | Y
11	0	?	X | Y

V	tag	data
0	?	X’ | Y’
0	?	X’ | Y’
0	?	X’ | Y’
0	?	X’ | Y’

V	tag	data
0	?	X’’ | Y’’
0	?	X’’ | Y’’
0	?	X’’ | Y’’
0	?	X’’ | Y’’

How many tag bits? = 2

Eviction Policies

Which cache line should be evicted from the cache to make room for a new line?

- Direct-mapped: no choice, must evict line selected by index
- Associative caches
  - Random: select one of the lines at random
  - Round-Robin: similar to random
  - FIFO: replace oldest line
  - LRU: replace line that has not been used in the longest time

Misses: the Three C’s

- Cold (compulsory) Miss:

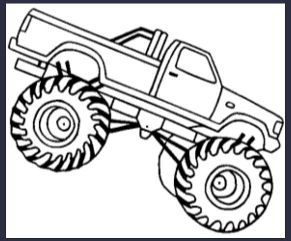
  never seen this address before

- Conflict Miss:

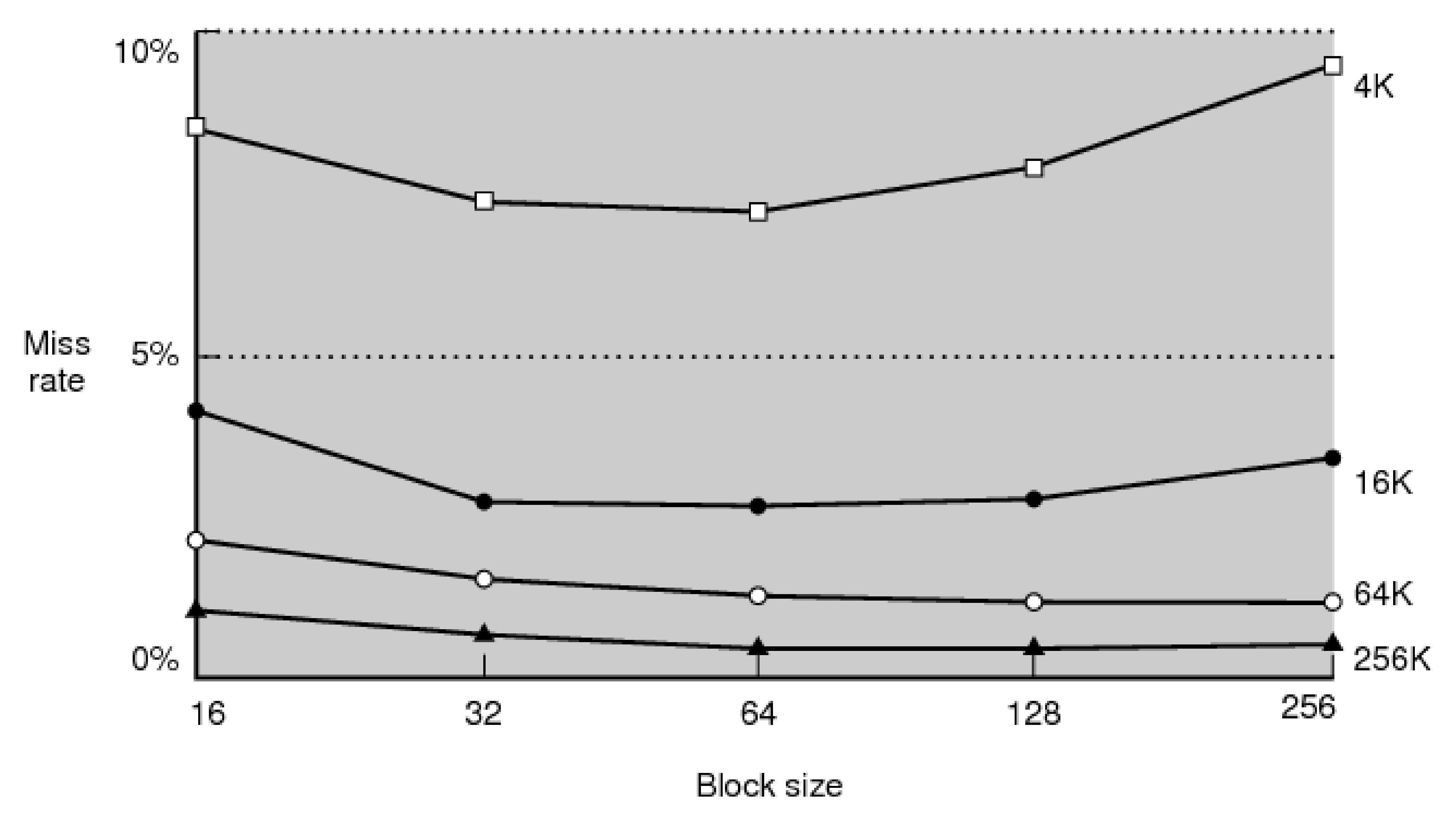
  cache associativity is too low

- Capacity Miss:

  cache is too small



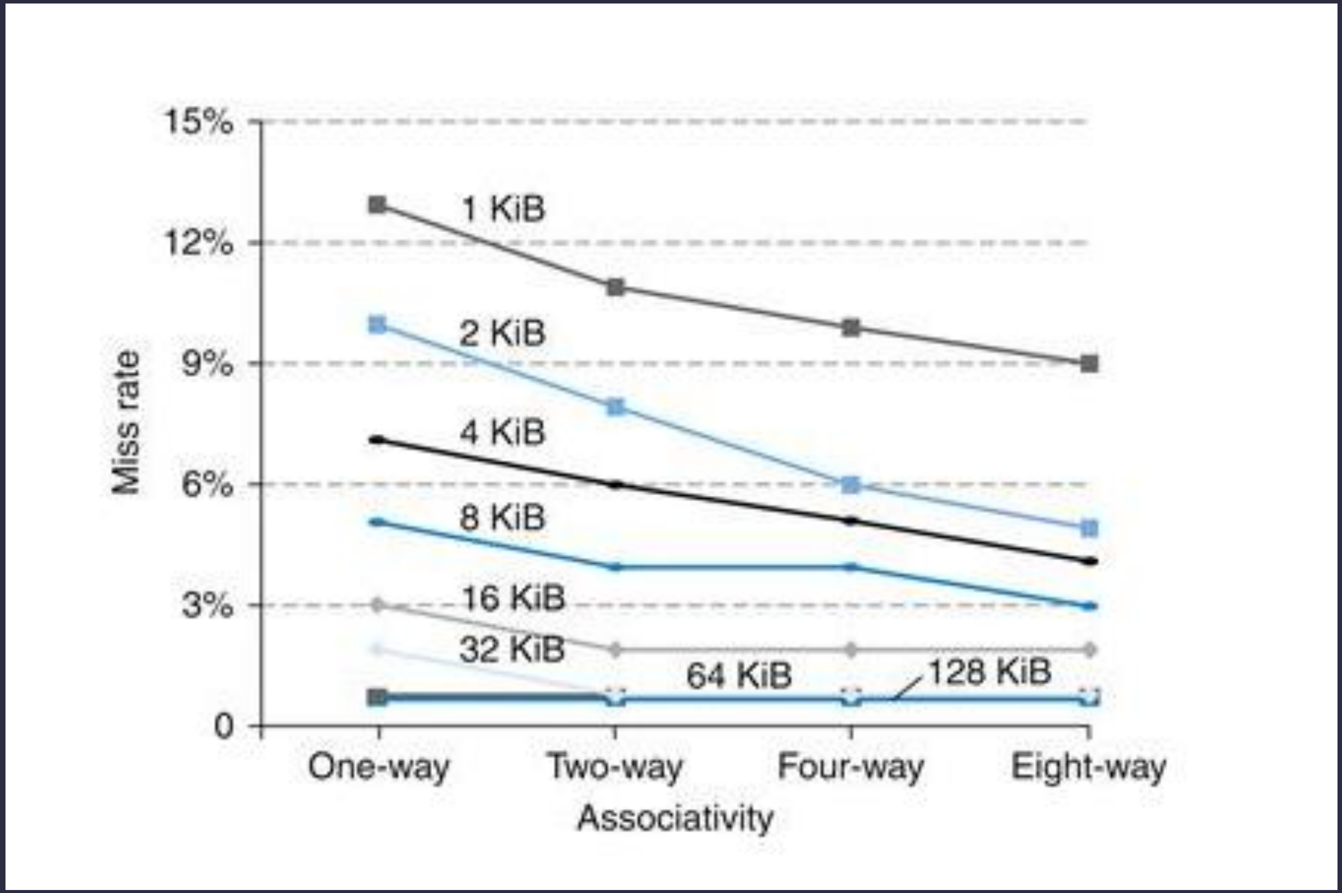
Miss Rate vs. Block Size



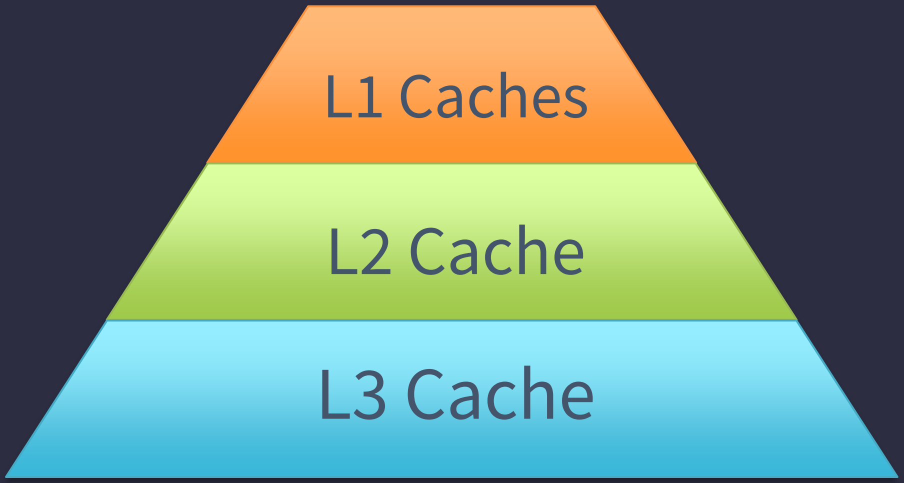
Block Size Tradeoffs

- For a given total cache size,
  Larger block sizes mean….
  - fewer lines
  - so fewer tags, less overhead
  - and fewer cold misses (within-block “prefetching”)
- But also…
  - fewer blocks available (for scattered accesses!)
  - so more conflicts
  - can decrease performance if **working set** can’t fit in $
  - and larger miss penalty (time to fetch block)

Miss Rate vs. Associativity



Which caches get what properties?



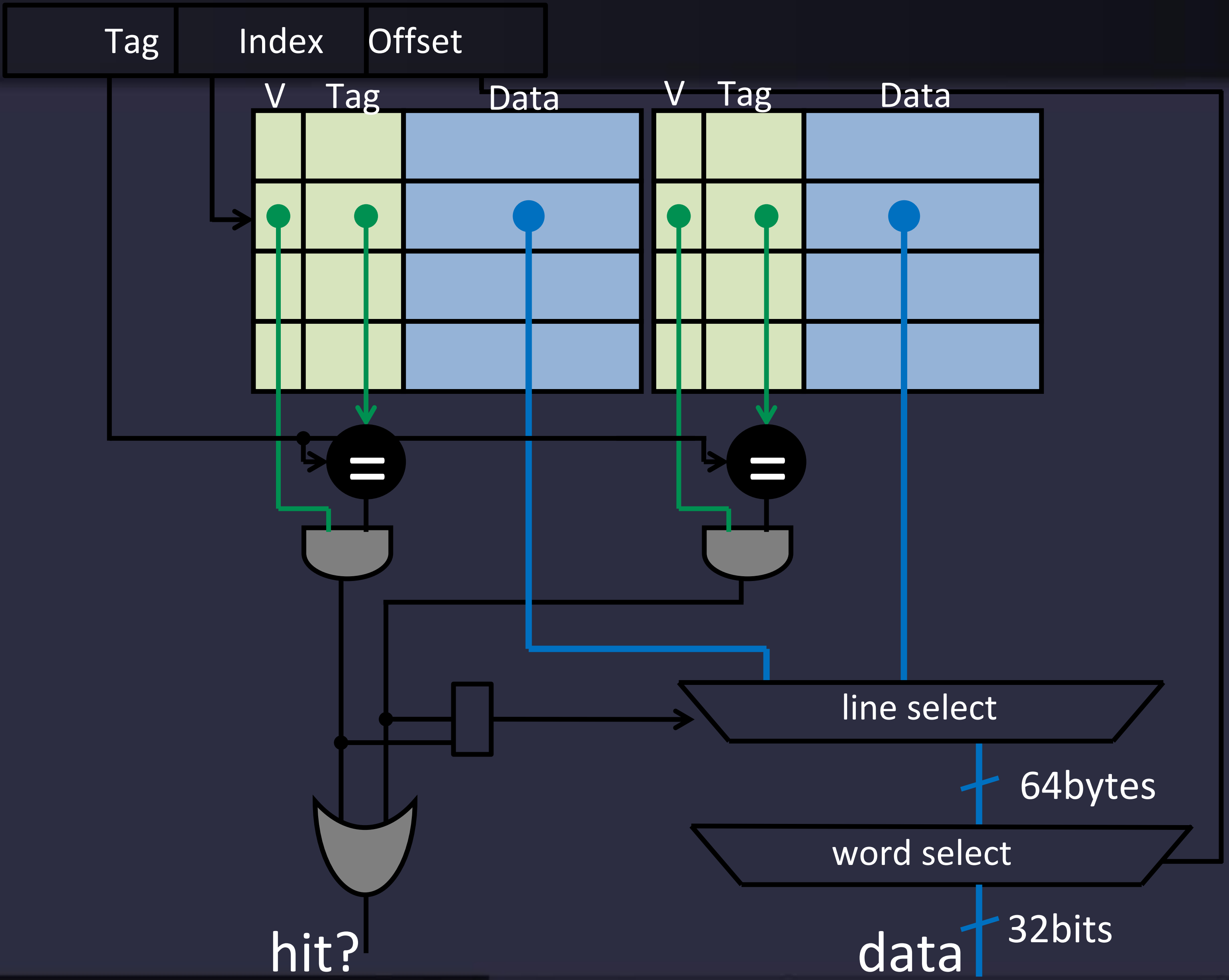
$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

Design with speed in mind

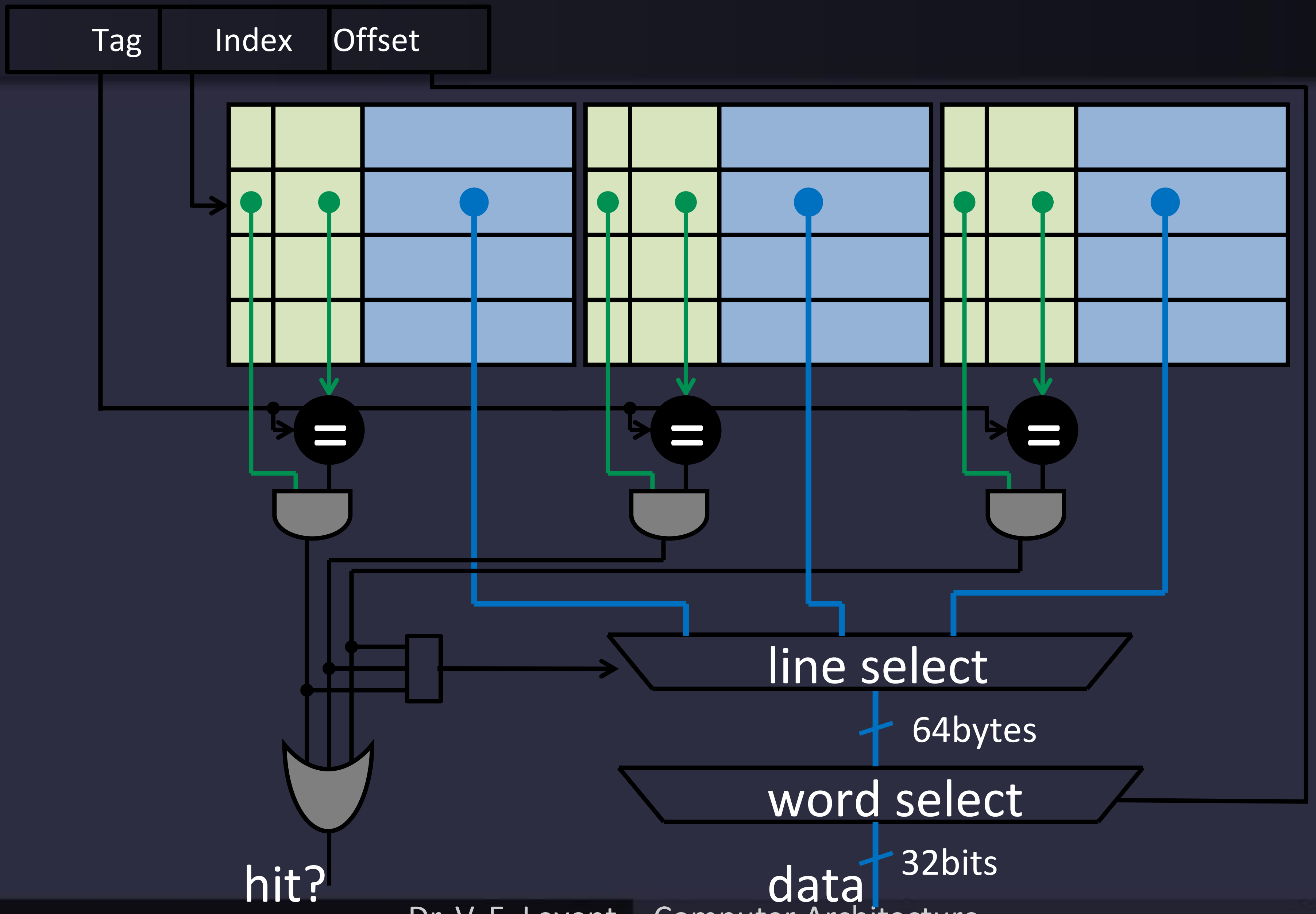
More Associative
Bigger Block Sizes
Larger Capacity

Design with miss rate in mind

2-Way Set Associative Cache (Reading)



3-Way Set Associative Cache (Reading)



Performance Calculation with $ Hierarchy

$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20%   (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$}$ =
  - $t_{avgD\$}$ =
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$}$ =
  - t_{avgL2} =
  - $t_{avgD\$}$ =

Performance Calculation with $ Hierarchy

$$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$$

- **Parameters**
  - Reference stream: all loads
  - D$: t_{hit} = 1ns, $\%_{miss}$ = 5%
  - L2: t_{hit} = 10ns, $\%_{miss}$ = 20% (local miss rate)
  - Main memory: t_{hit} = 50ns
- **What is $t_{avgD\$}$ without an L2?**
  - $t_{missD\$}$ = t_{hitM}
  - $t_{avgD\$}$ = $t_{hitD\$} + \%_{missD\$} {}^{*} t_{hitM}$ = 1ns+(0.05*50ns) = 3.5ns
- **What is $t_{avgD\$}$ with an L2?**
  - $t_{missD\$}$ = t_{avgL2}
  - t_{avgL2} = $t_{hitL2} + \%_{missL2} {}^{*} t_{hitM}$ = 10ns+(0.2*50ns) = 20ns
  - $t_{avgD\$}$ = $t_{hitD\$} + \%_{missD\$} {}^{*} t_{avgL2}$ = 1ns+(0.05*20ns) = 2ns

Performance Summary

Average memory access time (AMAT) depends on:

- cache architecture and size
- Hit and miss rates
- Access times and miss penalty

Cache design a very complex problem:

- Cache size, block size (aka line size)
- Number of ways of set-associativity (1, N, ∞)
- Eviction policy
- Number of levels of caching, parameters for each
- Separate I-cache from D-cache, or Unified cache
- Prefetching policies / instructions
- Write policy

Takeaway

Direct Mapped → fast, but low hit rate

Fully Associative → higher hit cost, higher hit rate

Set Associative → middleground

Cache performance is measured by the average memory access time (AMAT), which depends cache architecture and size, but also the access time for hit, miss penalty, hit rate.

What about Stores?

We want to write to the cache.

If the data is not in the cache?

Bring it in. (Write allocate policy)

Should we also update memory?

- Yes: write-through policy
- No: write-back policy

Write-Through Cache

16 byte, byte-addressed memory
4 byte, fully-associative cache:
2-byte blocks, write-allocate
4 bit addresses:
3 bit tag, 1 bit offset

Instructions:
LB x1 ← M[1]
LB x2 ← M[7]
SB x2 → M[0]
SB x1 → M[5]
LB x2 ← M[10]
SB x1 → M[5]
SB x1 → M[10]

lru	V	tag	data
1	0		
0	0		

Register File

x0	
x1	
x2	
x3	

Cache
Misses: 0
Hits: 0
Reads: 0
Writes: 0

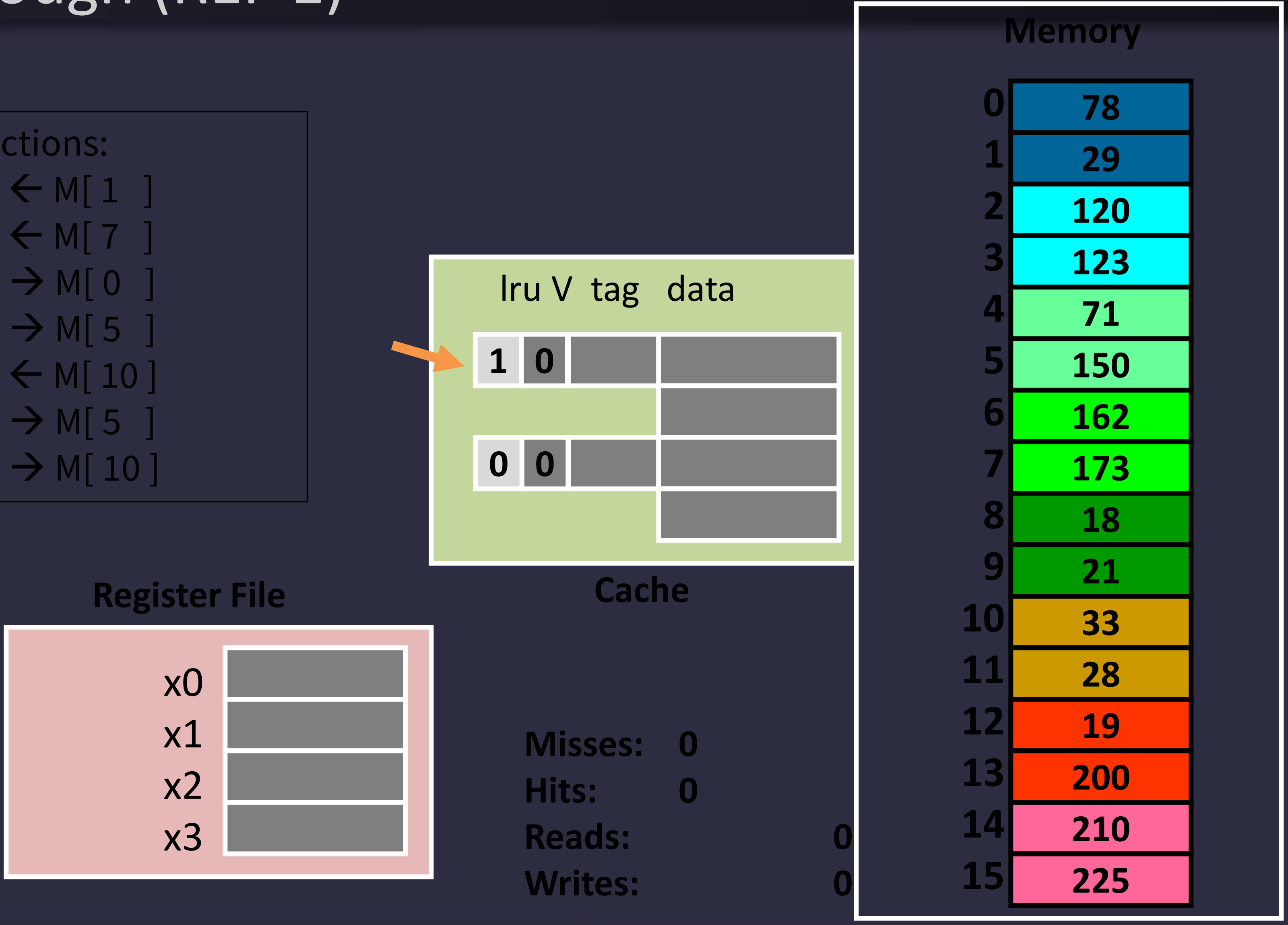
Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Through (REF 1)

Instructions:

- LB x1 ← M[1]
- LB x2 ← M[7]
- SB x2 → M[0]
- SB x1 → M[5]
- LB x2 ← M[10]
- SB x1 → M[5]
- SB x1 → M[10]



Cache

lru	V	tag	data
1	0		
0	0		

Register File

x0	
x1	
x2	
x3	

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses:	0
Hits:	0
Reads:	0
Writes:	0

Write-Through (REF 1)

Instructions:

LB x1 ← M[1] **M**

LB x2 ← M[7]

SB x2 → M[0]

SB x1 → M[5]

LB x2 ← M[10]

SB x1 → M[5]

SB x1 → M[10]

lru	V	tag	data
0	1	000	78
			29
1	0		

Cache

Register File

x0	
x1	29
x2	
x3	

Misses: 1

Hits: 0

Reads: 2

Writes: 0

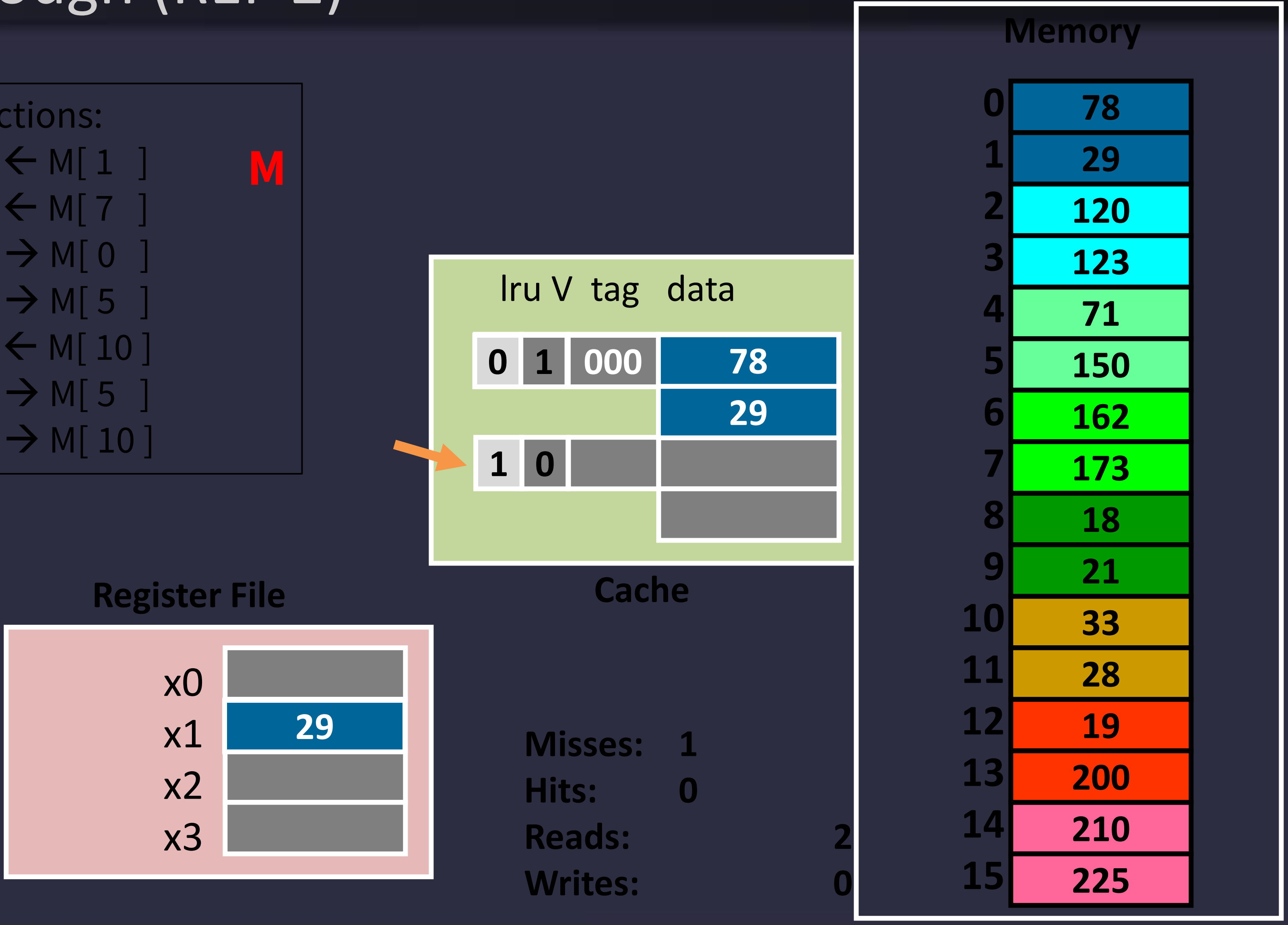
Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Through (REF 2)

Instructions:

- LB x1 ← M[1] **M**
- ➡ LB x2 ← M[7]
- SB x2 → M[0]
- SB x1 → M[5]
- LB x2 ← M[10]
- SB x1 → M[5]
- SB x1 → M[10]



Cache

lru	V	tag	data
0	1	000	78
			29
1	0		

Register File

x0	
x1	29
x2	
x3	

Memory

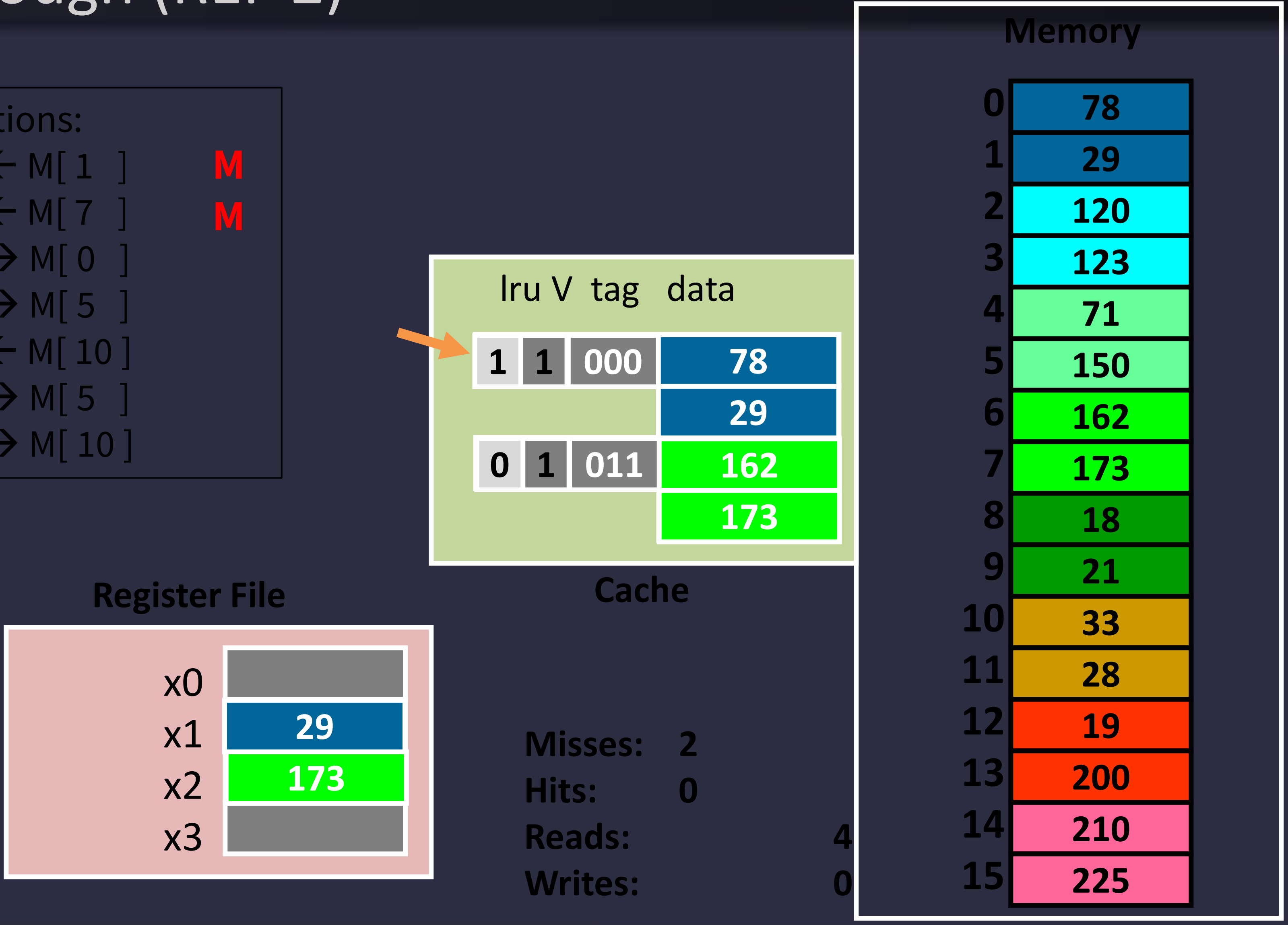
0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses: 1
Hits: 0
Reads: 2
Writes: 0

Write-Through (REF 2)

Instructions:

	LB x1 ← M[1]	M
➡	LB x2 ← M[7]	M
	SB x2 → M[0]	
	SB x1 → M[5]	
	LB x2 ← M[10]	
	SB x1 → M[5]	
	SB x1 → M[10]	



lru	V	tag	data
1	1	000	78
			29
0	1	011	162
			173

Cache

Register File

x0	
x1	29
x2	173
x3	

Misses: 2

Hits: 0

Reads: 4

Writes: 0

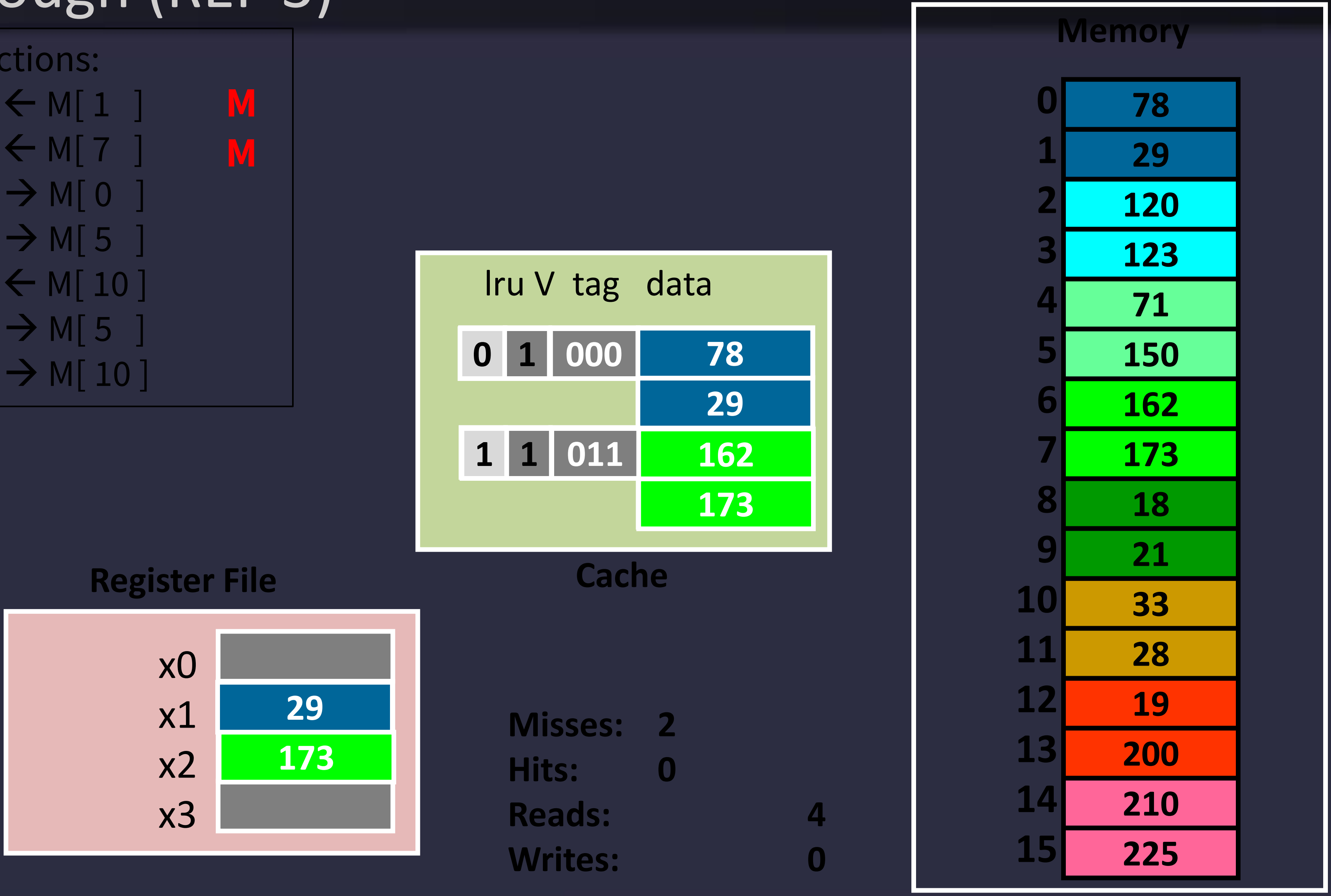
Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Through (REF 3)

Instructions:

LB x1 ← M[1] **M**
LB x2 ← M[7] **M**
➡ SB x2 → M[0]
SB x1 → M[5]
LB x2 ← M[10]
SB x1 → M[5]
SB x1 → M[10]



Cache

lru	V	tag	data
0	1	000	78
			29
1	1	011	162
			173

Register File

x0	
x1	29
x2	173
x3	

Memory

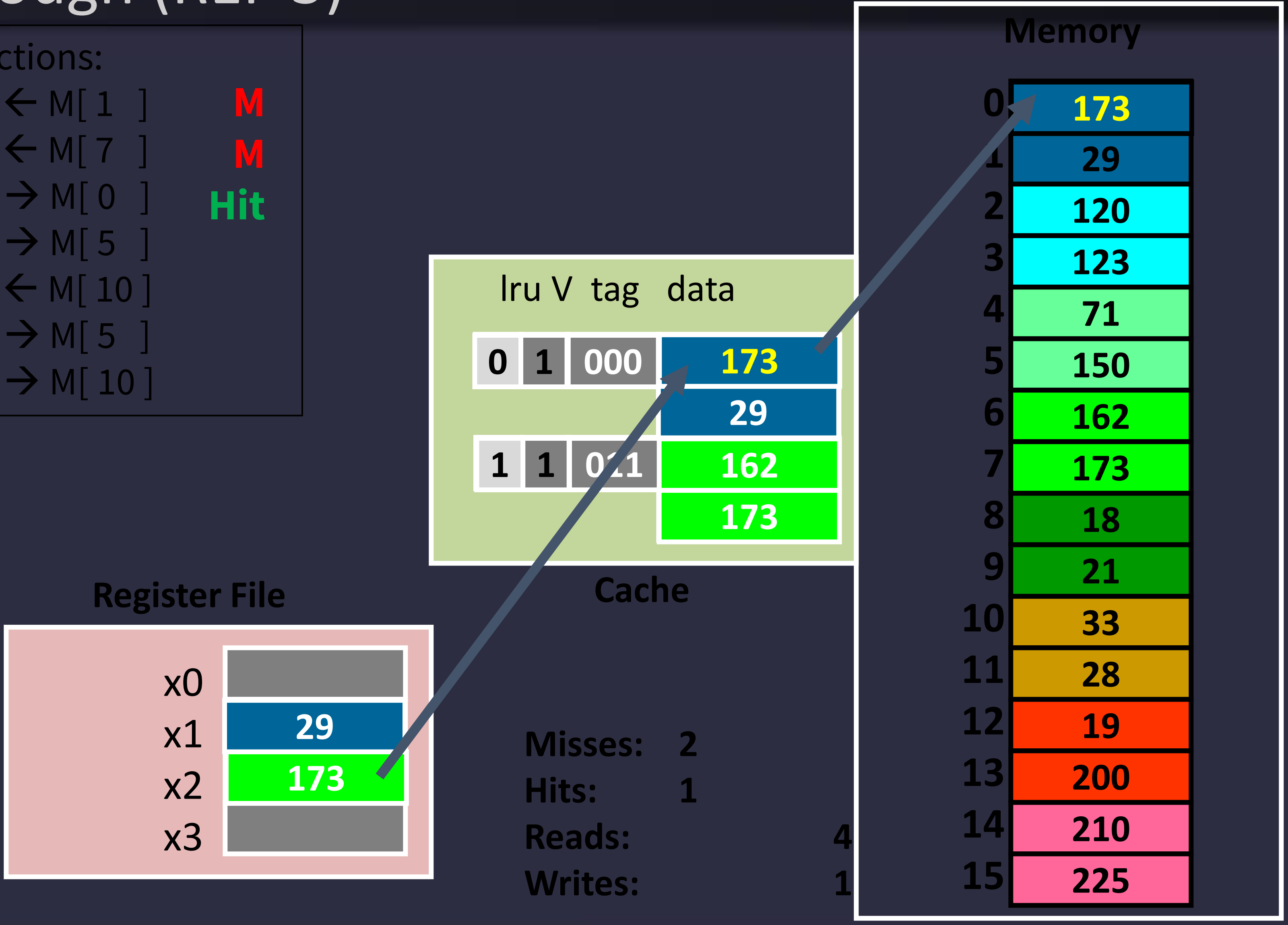
0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses:	2
Hits:	0
Reads:	4
Writes:	0

Write-Through (REF 3)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	



Register File

x0	
x1	29
x2	173
x3	

Cache

lru	V	tag	data
0	1	000	173
			29
1	1	011	162
			173

Memory

0	173
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses: 2

Hits: 1

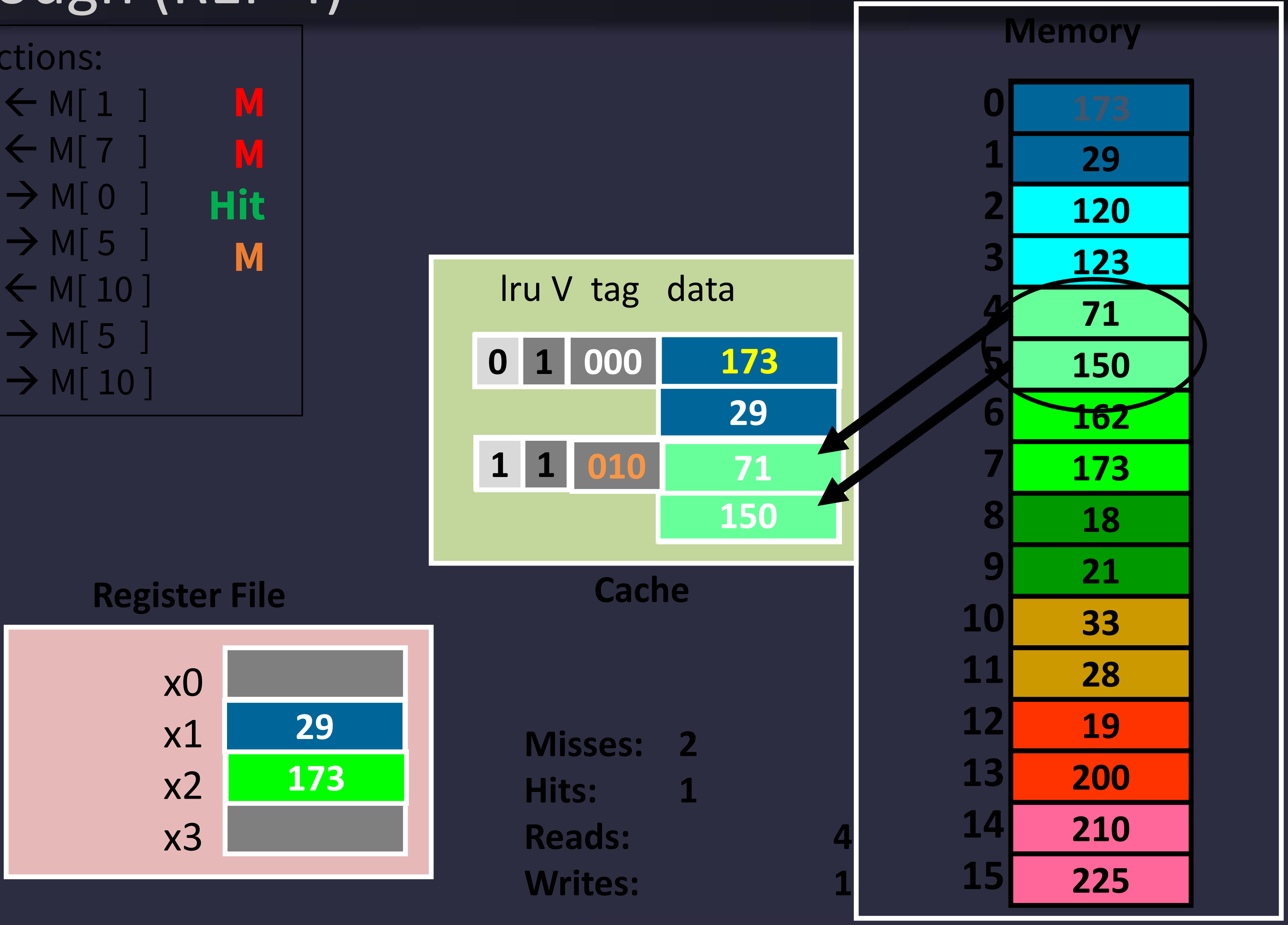
Reads: 4

Writes: 1

Write-Through (REF 4)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	



Cache

lru	V	tag	data
0	1	000	173
			29
1	1	010	71
			150

Memory

0	173
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Register File

x0	
x1	29
x2	173
x3	

Misses: 2

Hits: 1

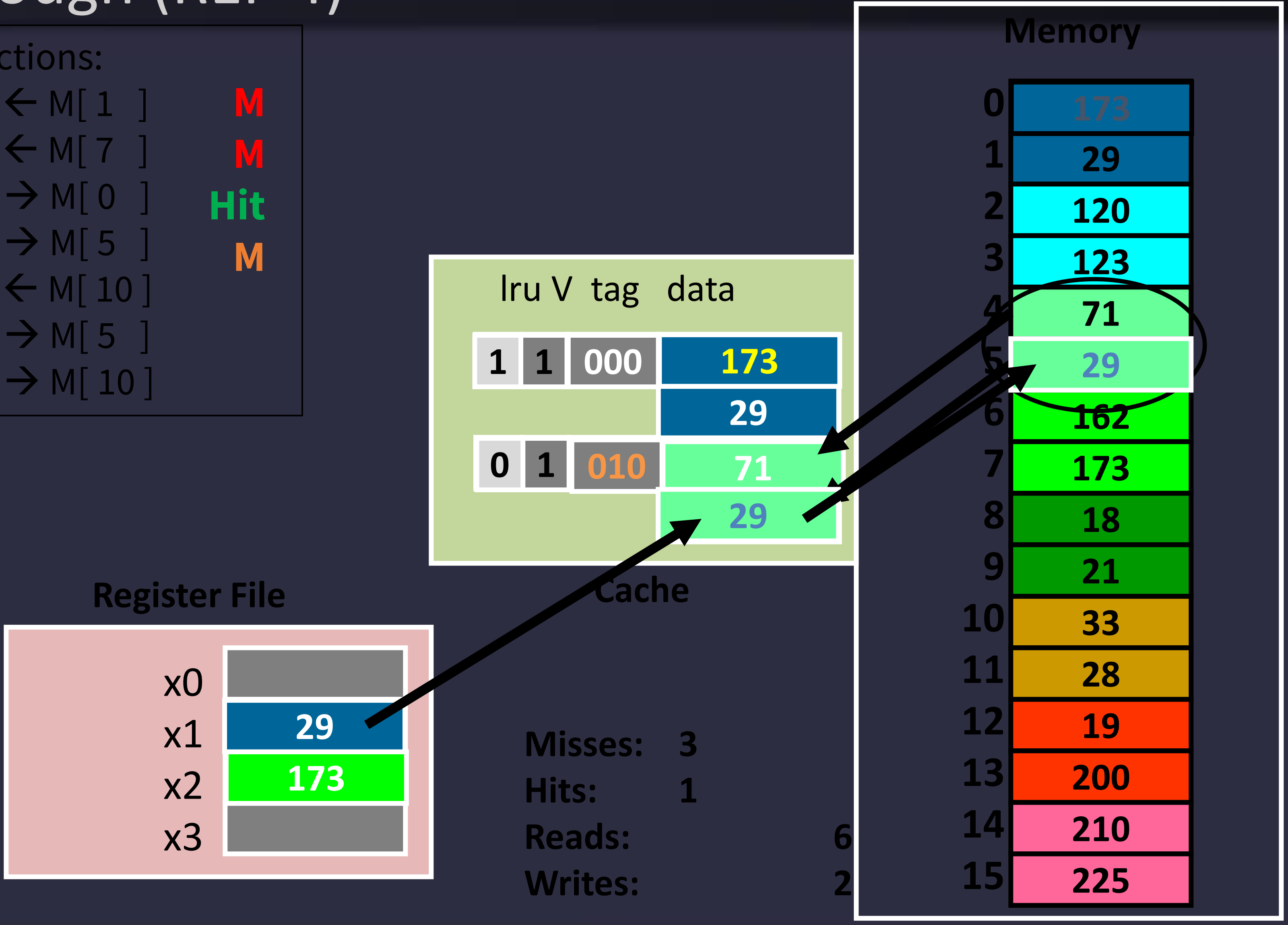
Reads: 4

Writes: 1

Write-Through (REF 4)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
➡ SB x1 → M[5]	M
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	



Cache

lru	V	tag	data
1	1	000	173
			29
0	1	010	71
			29

Memory

0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Register File

x0	
x1	29
x2	173
x3	

Misses:	3	
Hits:	1	
Reads:		6
Writes:		2

Write-Through (REF 4)

Instructions:

	LB x1 ← M[1]	M
	LB x2 ← M[7]	M
	SB x2 → M[0]	Hit
	SB x1 → M[5]	M
➡	LB x2 ← M[10]	
	SB x1 → M[5]	
	SB x1 → M[10]	

Cache

lru	V	tag	data
1	1	000	173
			29
0	1	010	71
			29

Register File

x0	
x1	29
x2	173
x3	

Misses:	3
Hits:	1
Reads:	6
Writes:	2

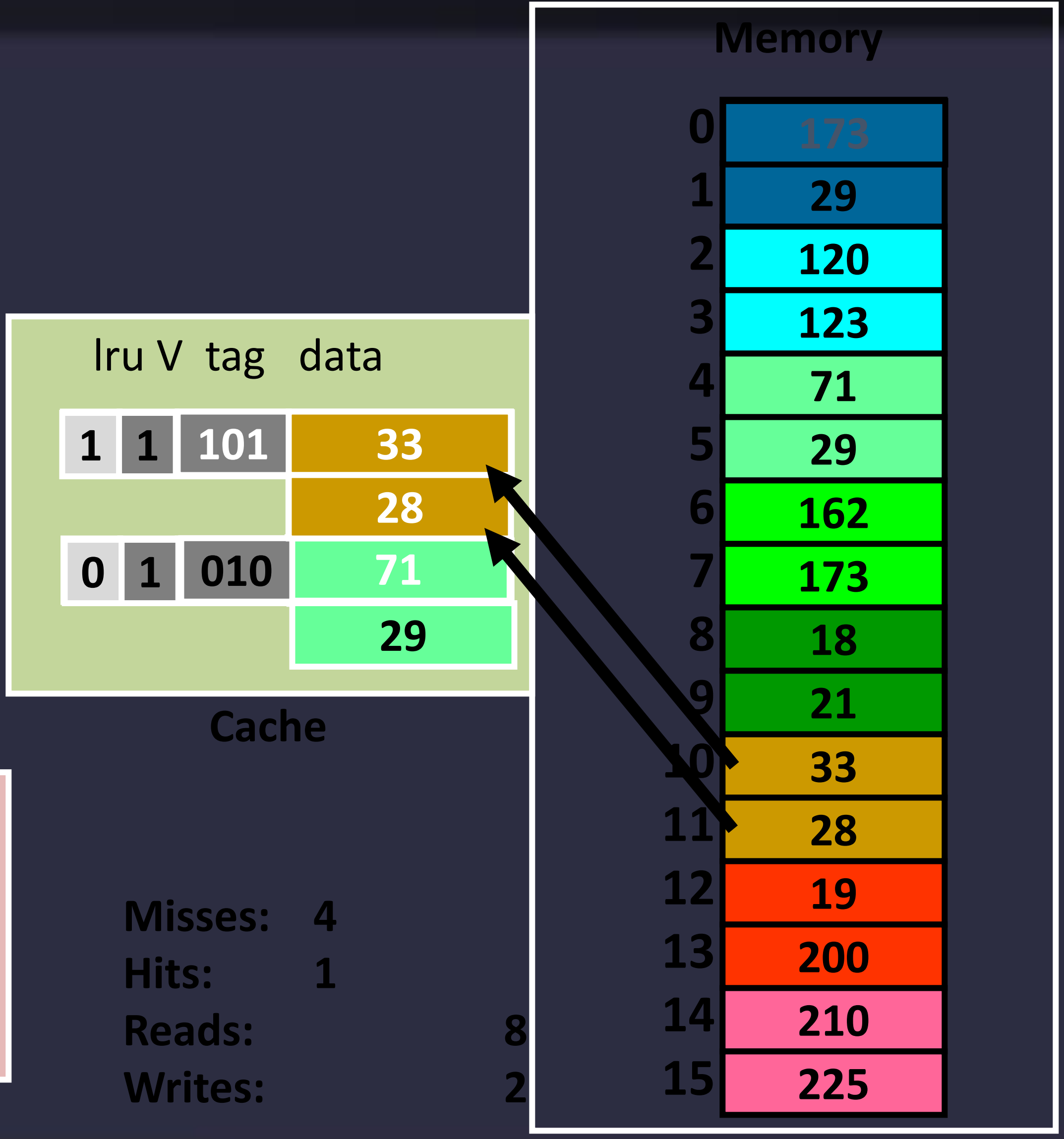
Memory

0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Through (REF 5)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
➡ LB x2 ← M[10]	M
SB x1 → M[5]	
SB x1 → M[10]	



Cache

lru	V	tag	data
1	1	101	33
			28
0	1	010	71
			29

Memory

0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Register File

x0	
x1	29
x2	33
x3	

Misses: 4
Hits: 1
Reads: 8
Writes: 2

Write-Through (REF 6)

Instructions:

	LB x1 ← M[1]	M
	LB x2 ← M[7]	M
	SB x2 → M[0]	Hit
	SB x1 → M[5]	M
	LB x2 ← M[10]	M
➡	SB x1 → M[5]	
	SB x1 → M[10]	

Cache

lru	V	tag	data
0	1	101	33
			28
1	1	010	71
			29

Register File

x0	
x1	29
x2	33
x3	

Misses: 4

Hits: 1

Reads: 8

Writes: 2

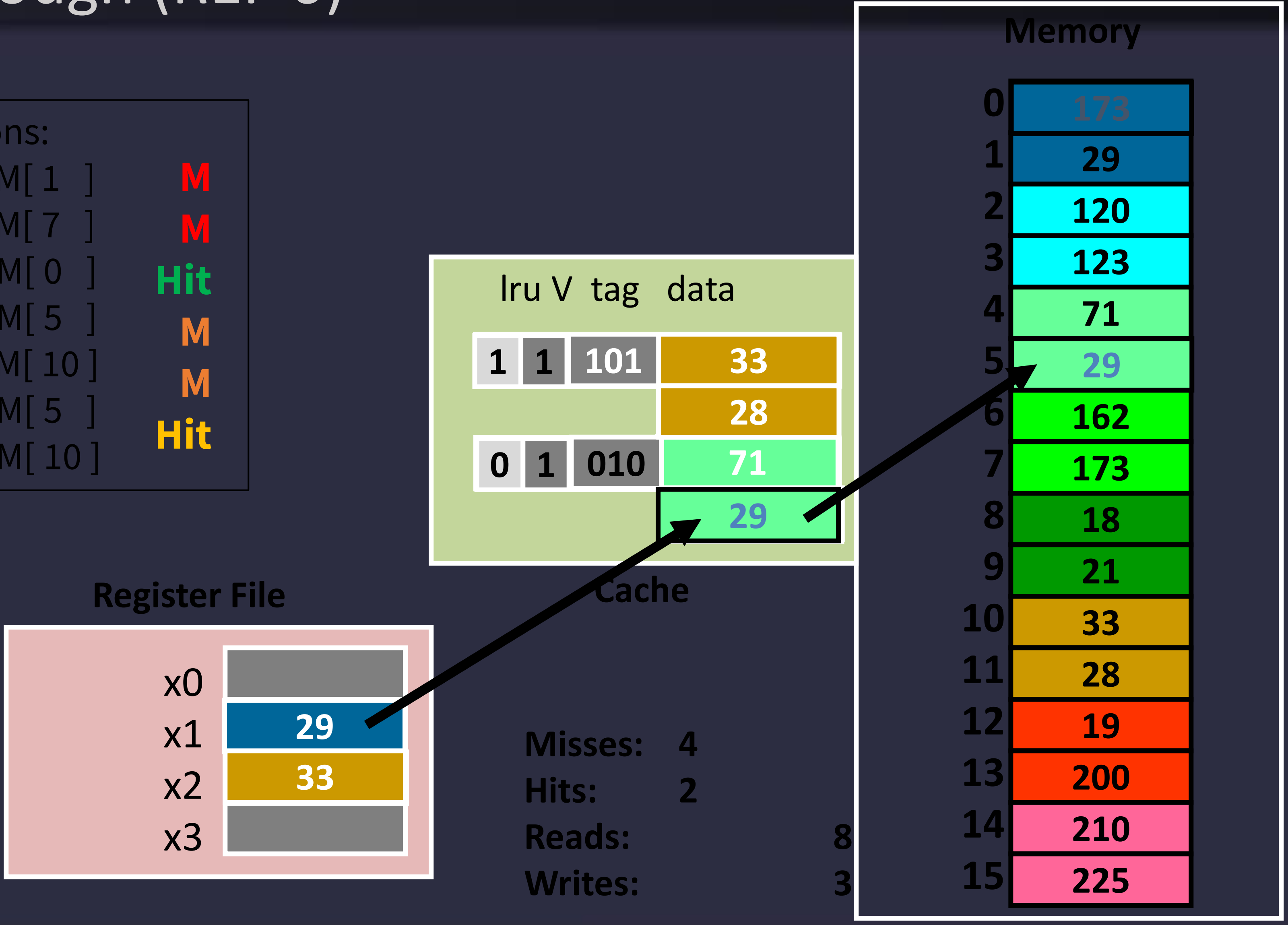
Memory

0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Through (REF 6)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	M
➡ SB x1 → M[5]	Hit
SB x1 → M[10]	



Cache

lru	V	tag	data
1	1	101	33
			28
0	1	010	71
			29

Register File

x0	
x1	29
x2	33
x3	

Memory

0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses:	4
Hits:	2
Reads:	8
Writes:	3

Write-Through (REF 7)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	M
SB x1 → M[5]	Hit
➡ SB x1 → M[10]	

Cache

lru	V	tag	data
1	1	101	33
			28
0	1	010	71
			29

Register File

x0	
x1	29
x2	33
x3	

Misses:	4
Hits:	2
Reads:	8
Writes:	3

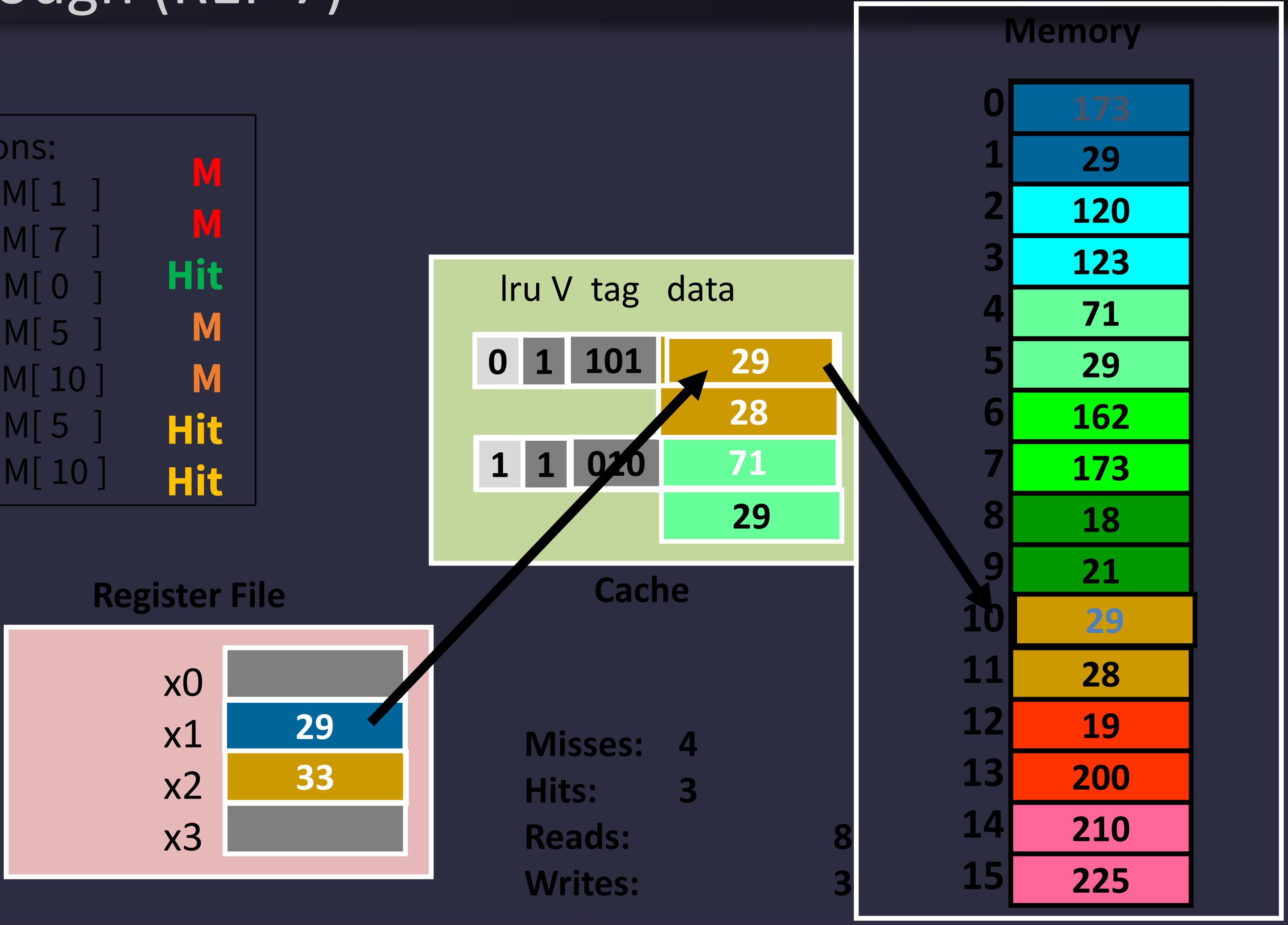
Memory

0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Through (REF 7)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	M
SB x1 → M[5]	Hit
SB x1 → M[10]	Hit



Register File

x0	
x1	29
x2	33
x3	

Cache

lru	V	tag	data
0	1	101	29
			28
1	1	010	71
			29

Memory

0	173
1	29
2	120
3	123
4	71
5	29
6	162
7	173
8	18
9	21
10	29
11	28
12	19
13	200
14	210
15	225

Misses: 4

Hits: 3

Reads: 8

Writes: 3

Summary: Write Through

Write-through policy with write allocate

- Cache miss: read entire block from memory
- Write: write only updated item to memory
- Eviction: no need to write to memory

Next Goal: Write-Through vs. Write-Back

What if we DON'T to write stores immediately to memory?

- Keep the current copy in cache, and update memory when data is **evicted** (write-back policy)
- Write-back all evicted lines?
  - No, only written-to blocks

Write-Back Meta-Data (Valid, Dirty Bits)

V	D	Tag	Byte 1	Byte 2	... Byte N

- V = 1 means the line has valid data
- D = 1 means the bytes are newer than main memory
- When allocating line:
  - Set V = 1, D = 0, fill in Tag and Data
- When writing line:
  - Set D = 1
- When evicting line:
  - If D = 0: just set V = 0
  - If D = 1: write-back Data, then set D = 0, V = 0

Write-back Example

- Example: How does a write-back cache work?
- Assume write-allocate

Handling Stores (Write-Back)

Instructions:

```
LB x1 ← M[ 1  ]
LB x2 ← M[ 7  ]
SB x2 → M[ 0  ]
SB x1 → M[ 5  ]
LB x2 ← M[ 10 ]
SB x1 → M[ 5  ]
SB x1 → M[ 10 ]
```

16 byte, byte-addressed memory
4 btye, fully-associative cache:
2-byte blocks, write-allocate
4 bit addresses:
3 bit tag, 1 bit offset

lru	V	d	tag	data
1	0			
0	0			

Cache

Register File

x0	
x1	
x2	
x3	

Misses: 0
Hits: 0
Reads: 0
Writes: 0

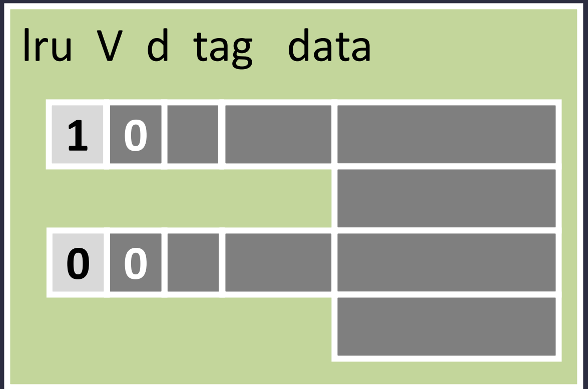
Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 1)

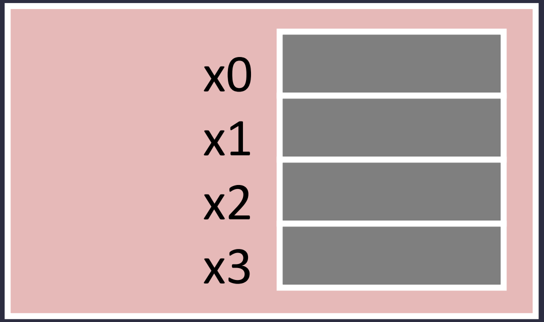
Instructions:

- LB x1 ← M[1]
- LB x2 ← M[7]
- SB x2 → M[0]
- SB x1 → M[5]
- LB x2 ← M[10]
- SB x1 → M[5]
- SB x1 → M[10]



Cache

Register File



Misses: 0
Hits: 0
Reads: 0
Writes: 0

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 1)

Instructions:

- LB x1 ← M[1] **M**
- LB x2 ← M[7]
- SB x2 → M[0]
- SB x1 → M[5]
- LB x2 ← M[10]
- SB x1 → M[5]
- SB x1 → M[10]

lru	V	d	tag	data
0	1	0	000	78
				29
1	0			

Cache

Register File

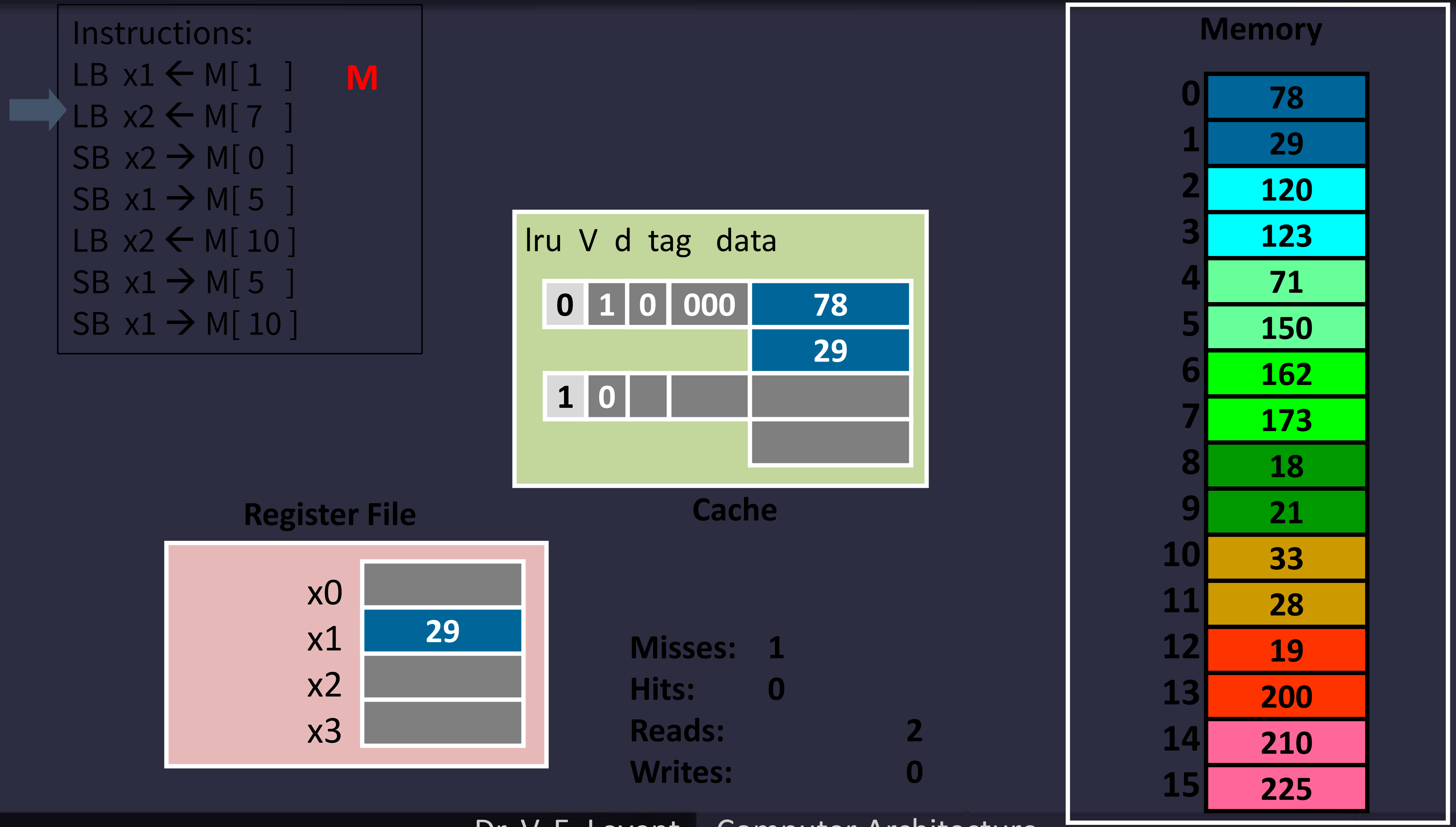
x0	
x1	29
x2	
x3	

Misses: 1
Hits: 0
Reads: 2
Writes: 0

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 2)



Instructions:

```
    LB  x1 ← M[ 1  ]      M
→   LB  x2 ← M[ 7  ]
    SB  x2 → M[ 0  ]
    SB  x1 → M[ 5  ]
    LB  x2 ← M[ 10 ]
    SB  x1 → M[ 5  ]
    SB  x1 → M[ 10 ]
```

lru	V	d	tag	data
0	1	0	000	78
				29
1	0			

Cache

Register File

x0	
x1	29
x2	
x3	

Misses: 1

Hits: 0

Reads: 2

Writes: 0

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 2)

Instructions:

LB x1 ← M[1] M
LB x2 ← M[7] M
SB x2 → M[0]
SB x1 → M[5]
LB x2 ← M[10]
SB x1 → M[5]
SB x1 → M[10]

lru	V	d	tag	data
1	1	0	000	78
				29
0	1	0	011	162
				173

Cache

Register File

x0	
x1	29
x2	173
x3	

Misses: 2
Hits: 0
Reads: 4
Writes: 0

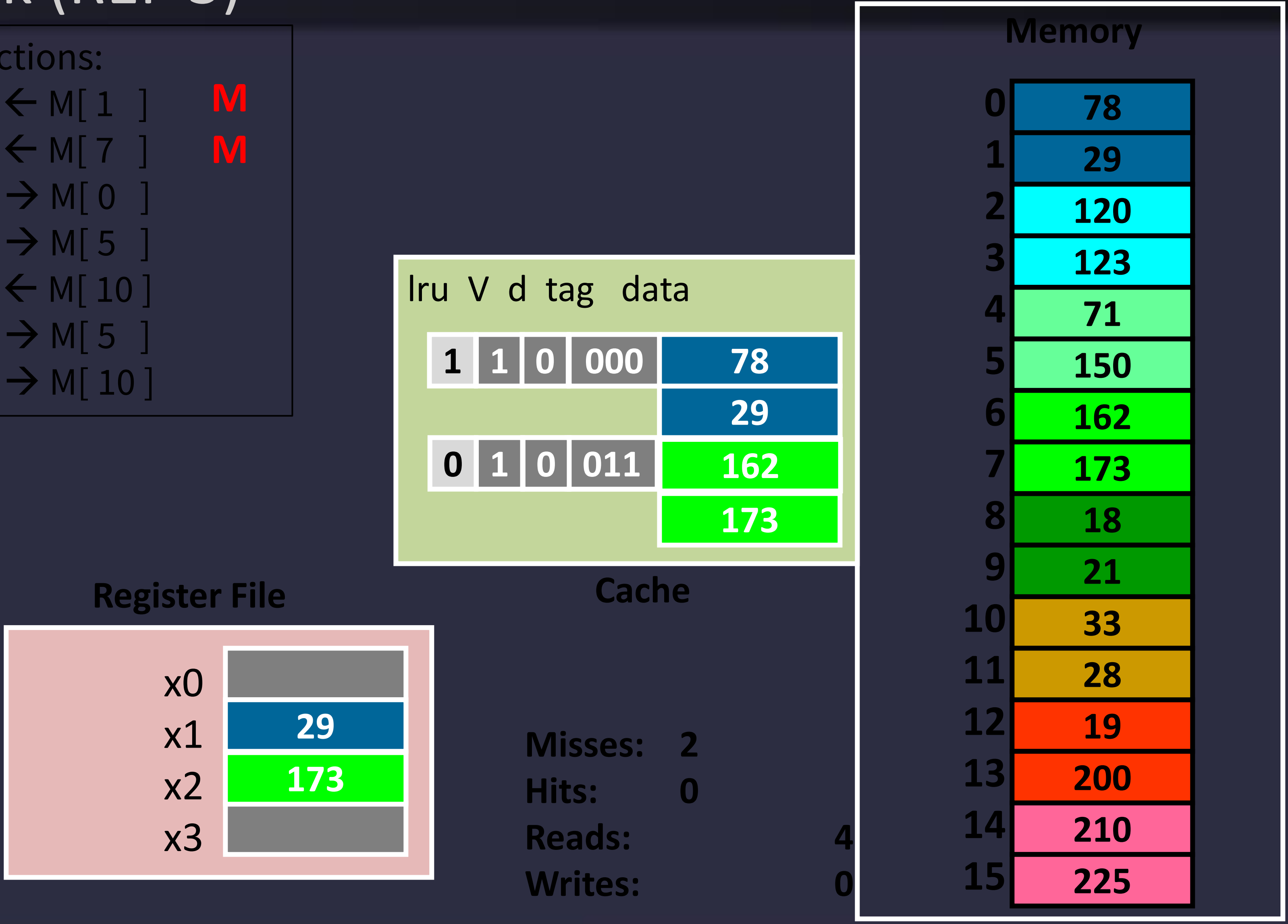
Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 3)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
➡ SB x2 → M[0]	
SB x1 → M[5]	
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	



Cache

lru	V	d	tag	data
1	1	0	000	78
				29
0	1	0	011	162
				173

Register File

x0	
x1	29
x2	173
x3	

Misses: 2

Hits: 0

Reads: 4

Writes: 0

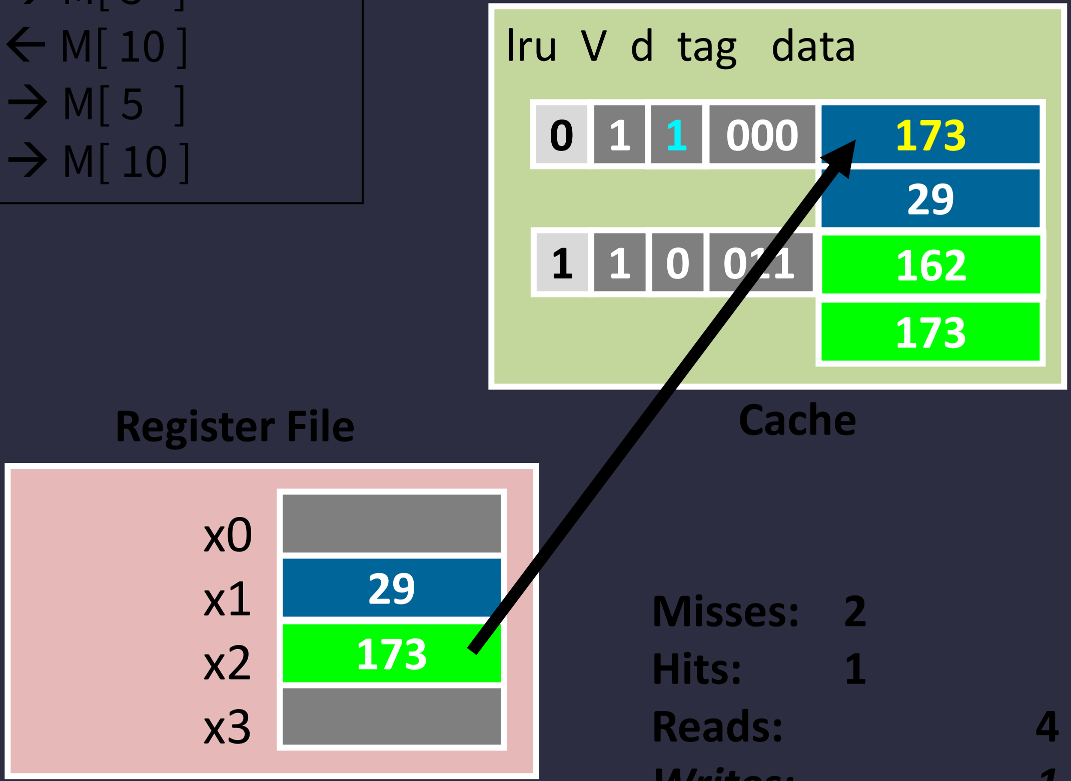
Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 3)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	



Cache

lru	V	d	tag	data
0	1	1	000	173
				29
1	1	0	011	162
				173

Register File

x0	
x1	29
x2	173
x3	

Misses:	2
Hits:	1
Reads:	4
Writes:	*1*

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 4)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	

lru	V	d	tag	data
0	1	1	000	173
				29
1	1	0	011	162
				173

Cache

Register File

x0	
x1	29
x2	173
x3	

Misses: 2
Hits: 1
Reads: 4
Writes: 0

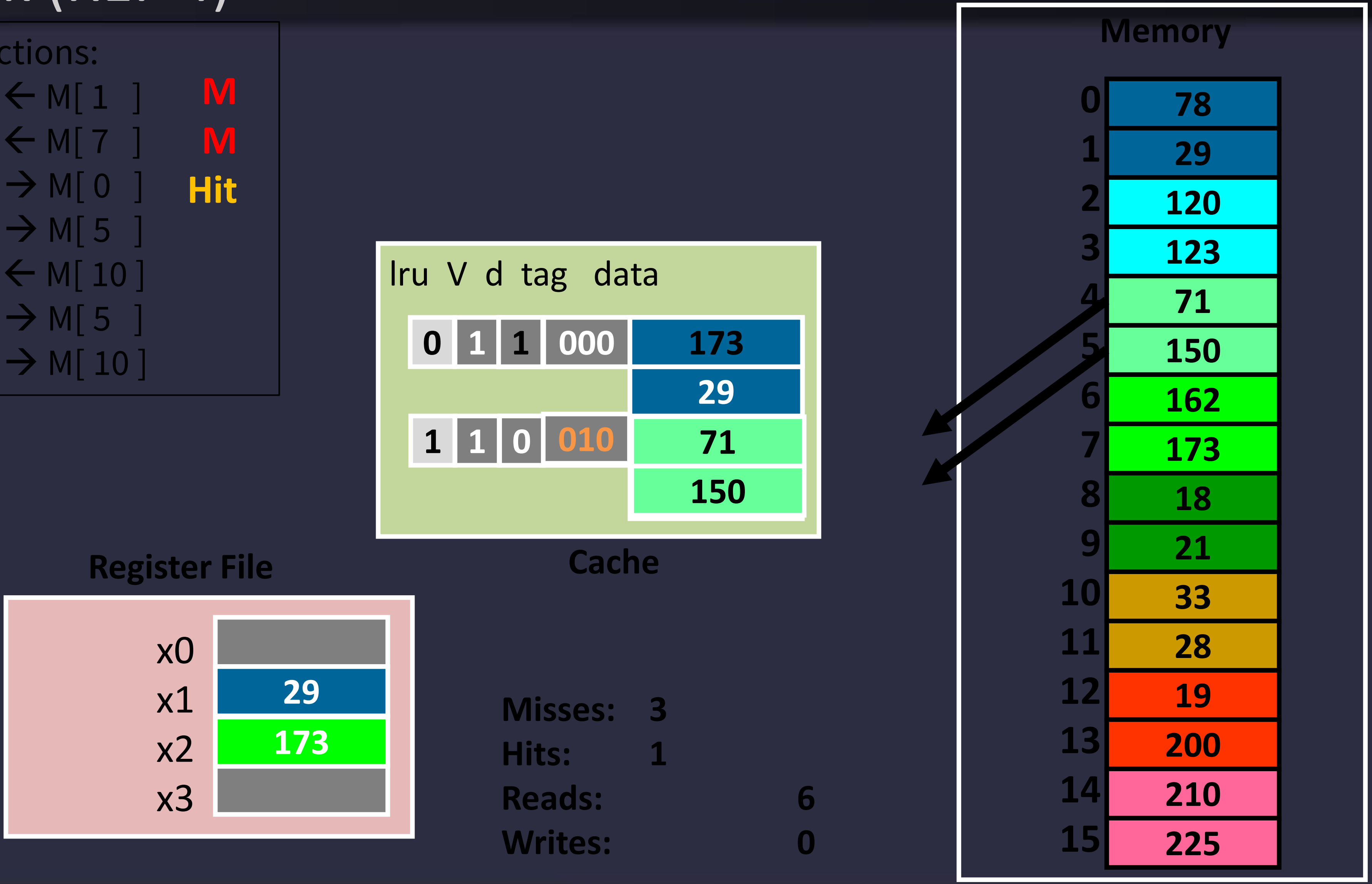
Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 4)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	



Cache

lru	V	d	tag	data
0	1	1	000	173 29
1	1	0	010	71 150

Register File

x0	
x1	29
x2	173
x3	

Memory

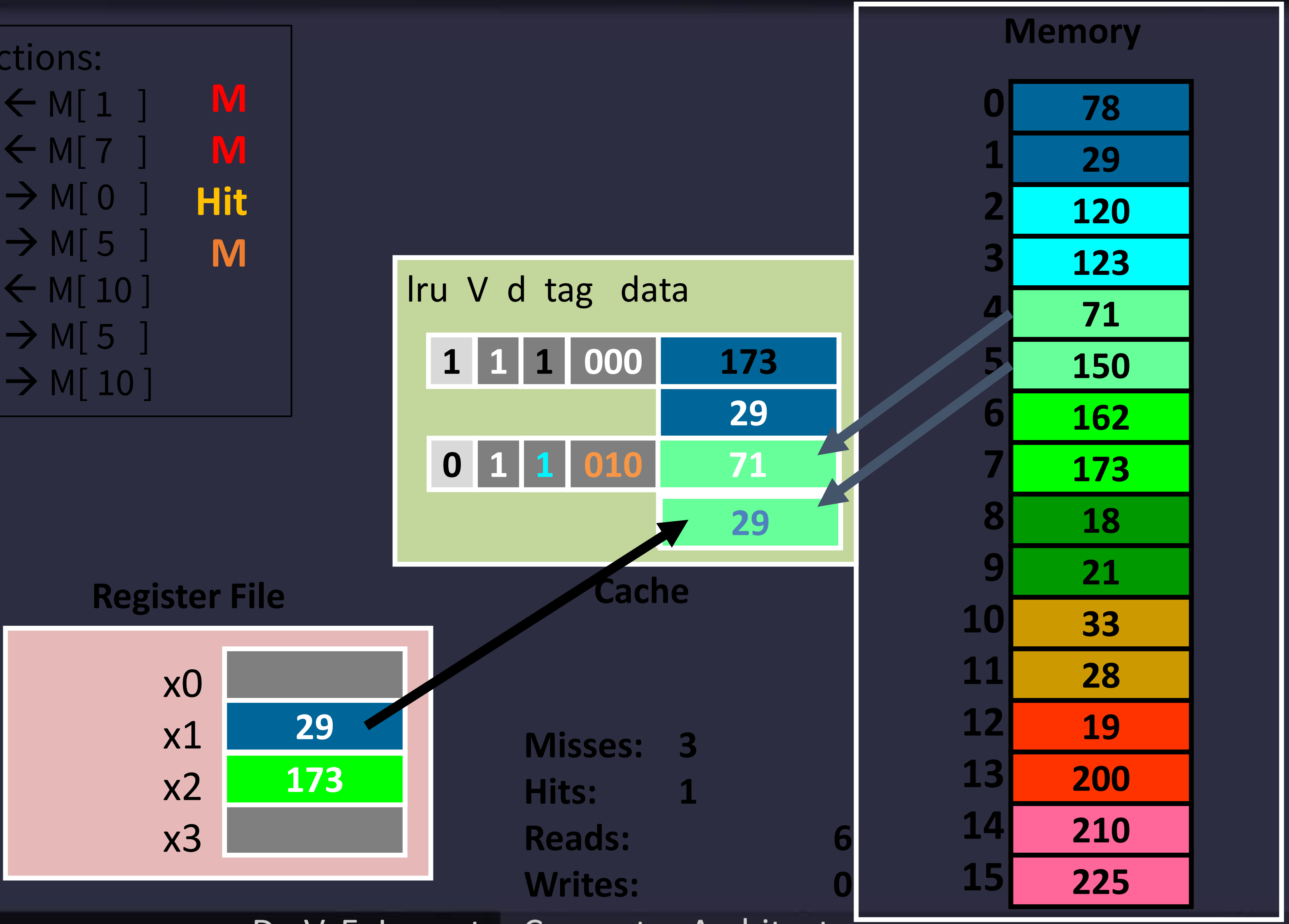
0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses:	3	
Hits:	1	
Reads:		6
Writes:		0

Write-Back (REF 4)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	



Cache

lru	V	d	tag	data
1	1	1	000	173 29
0	1	1	010	71 29

Register File

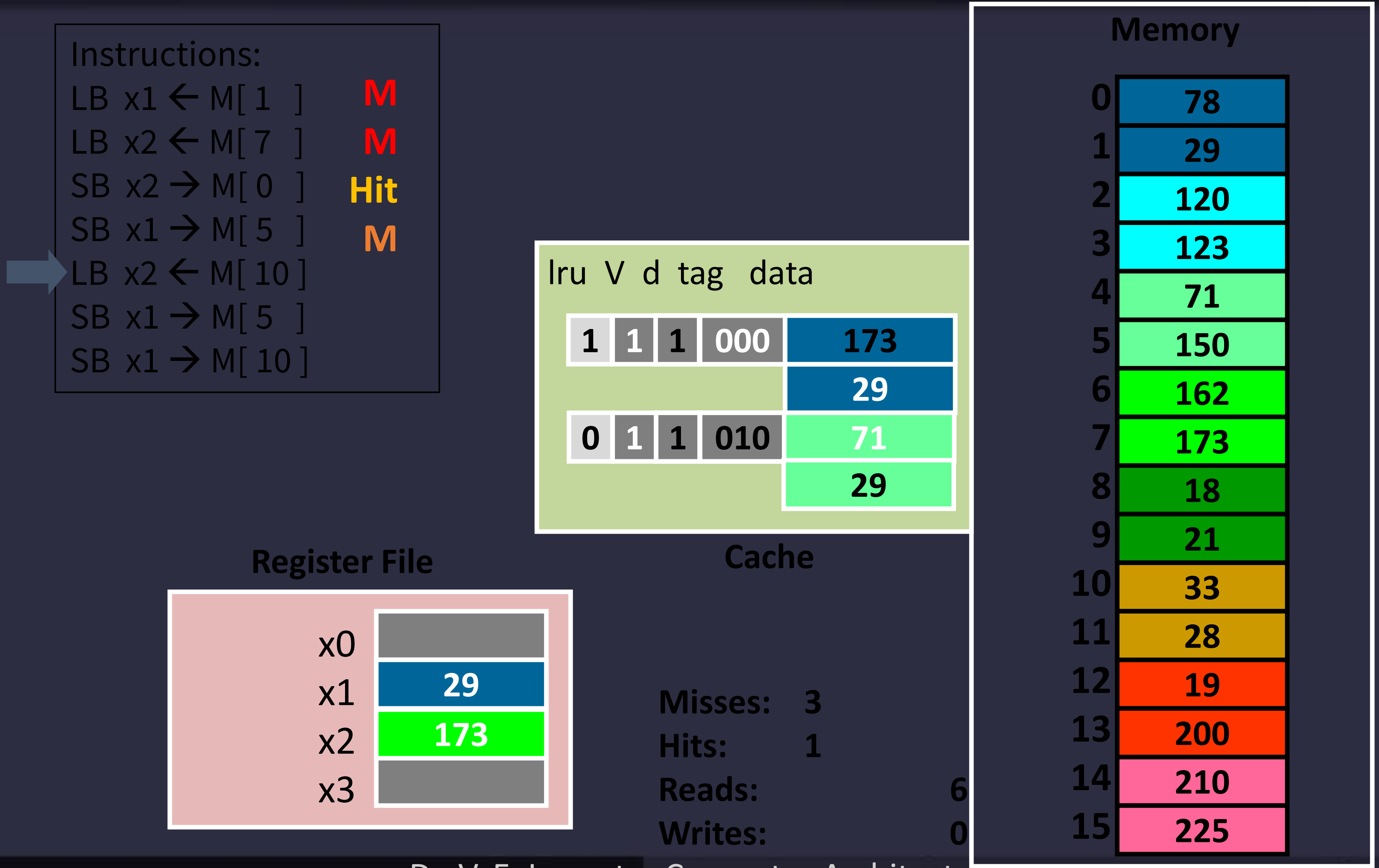
x0	
x1	29
x2	173
x3	

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses: 3
Hits: 1
Reads: 6
Writes: 0

Write-Back (REF 5)



Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	
SB x1 → M[5]	
SB x1 → M[10]	

Cache

lru	V	d	tag	data
1	1	1	000	173
				29
0	1	1	010	71
				29

Register File

x0	
x1	29
x2	173
x3	

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses: 3

Hits: 1

Reads: 6

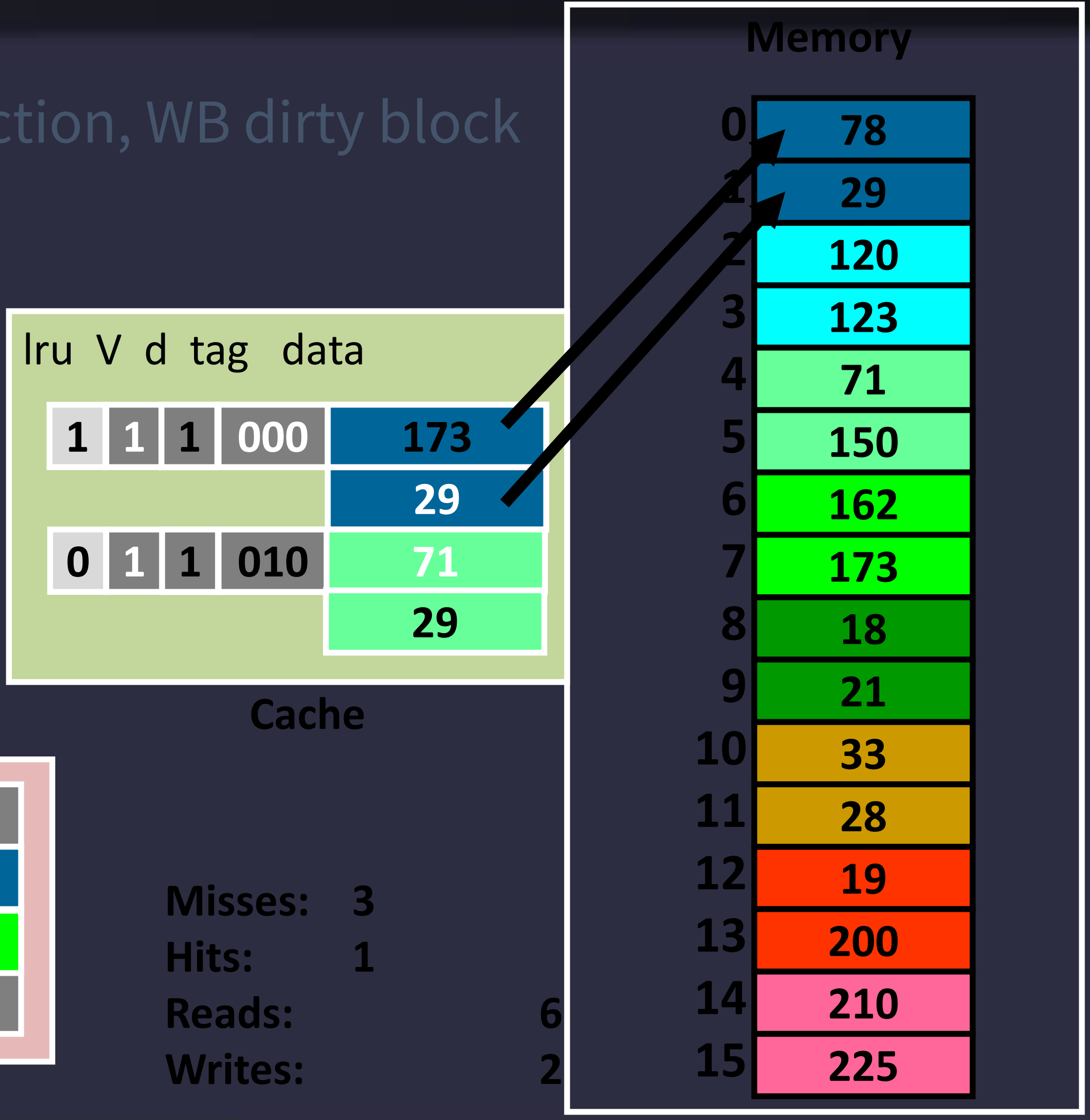
Writes: 0

Write-Back (REF 5)

Instructions:

	LB x1 ← M[1]	M
	LB x2 ← M[7]	M
	SB x2 → M[0]	Hit
	SB x1 → M[5]	M
➡	LB x2 ← M[10]	
	SB x1 → M[5]	
	SB x1 → M[10]	

Eviction, WB dirty block



Cache

lru	V	d	tag	data
1	1	1	000	173
				29
0	1	1	010	71
				29

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Register File

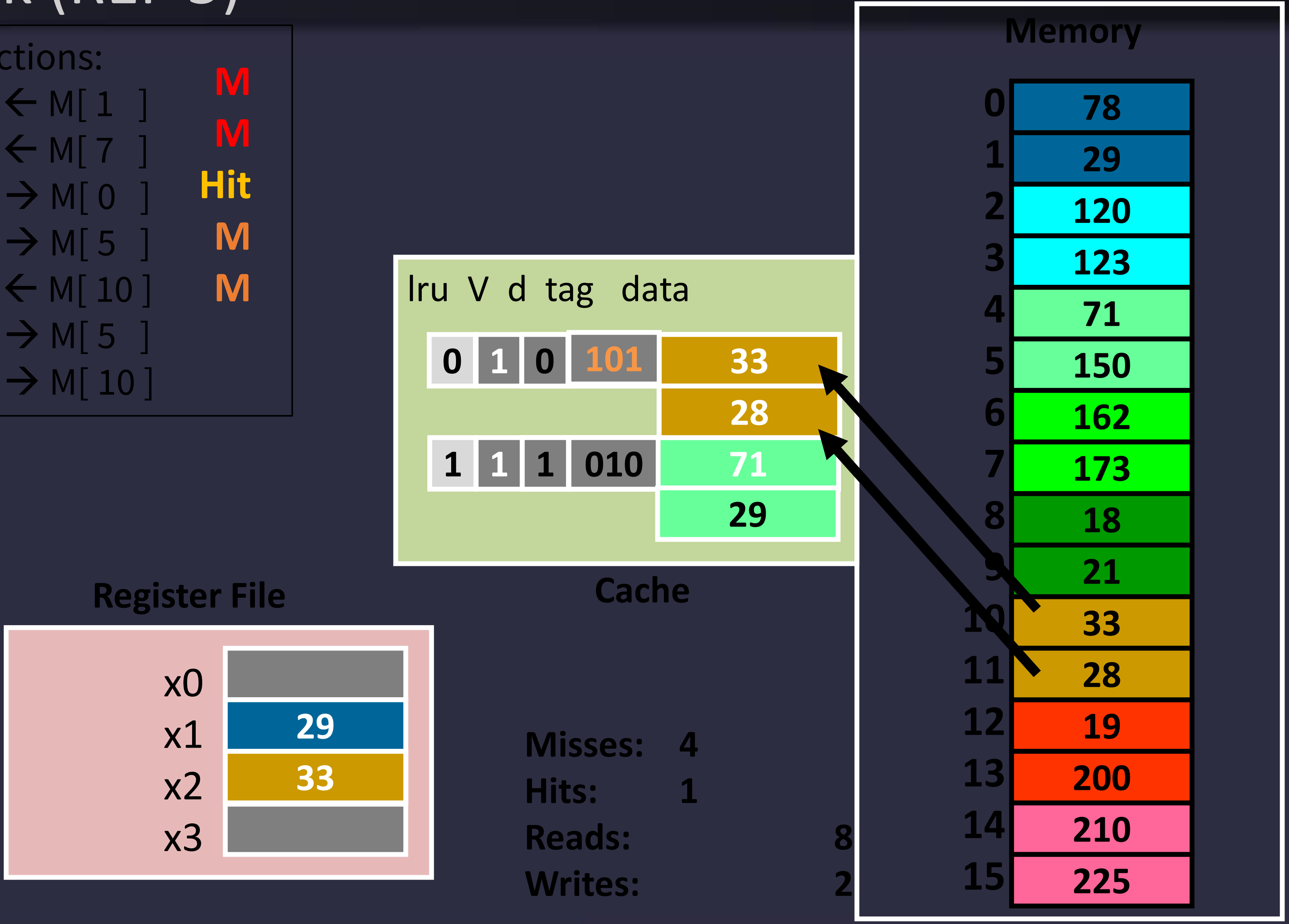
x0	
x1	29
x2	173
x3	

Misses:	3
Hits:	1
Reads:	6
Writes:	2

Write-Back (REF 5)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	M
SB x1 → M[5]	
SB x1 → M[10]	



Cache

lru	V	d	tag	data
0	1	0	101	33
				28
1	1	1	010	71
				29

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Register File

x0	
x1	29
x2	33
x3	

Misses: 4

Hits: 1

Reads: 8

Writes: 2

Write-Back (REF 6)

Instructions:	
LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	M
➡ SB x1 → M[5]	
SB x1 → M[10]	

lru	V	d	tag	data
0	1	0	101	33
				28
1	1	1	010	71
				29

Cache

Register File

x0	
x1	29
x2	33
x3	

Misses: 4

Hits: 1

Reads: 8

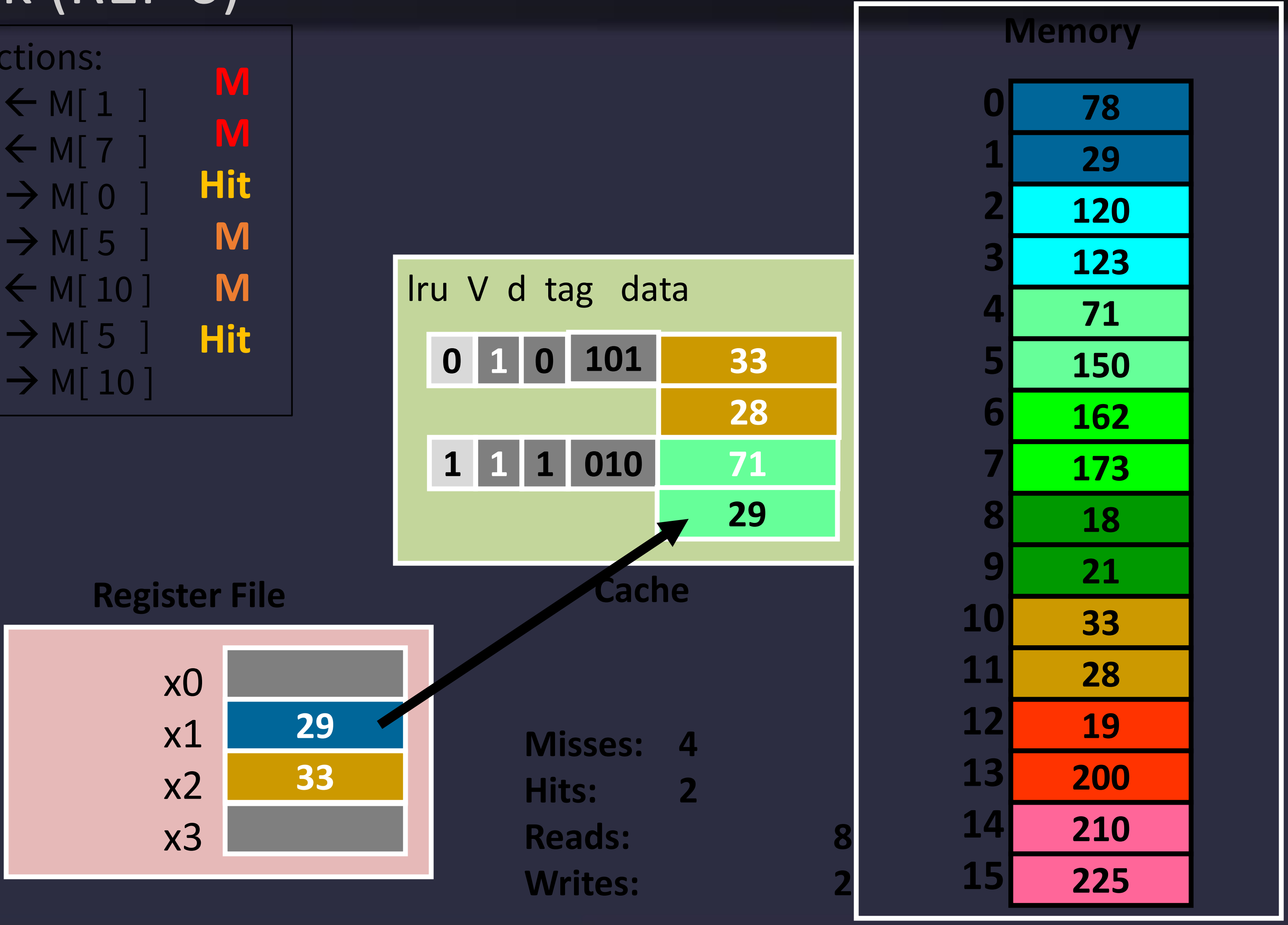
Writes: 2

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 6)

Instructions:	M
LB x1 ← M[1]	M
LB x2 ← M[7]	Hit
SB x2 → M[0]	M
SB x1 → M[5]	M
LB x2 ← M[10]	Hit
➡ SB x1 → M[5]	
SB x1 → M[10]	



lru	V	d	tag	data
0	1	0	101	33
				28
1	1	1	010	71
				29

Cache

Register File

x0	
x1	29
x2	33
x3	

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses:	4	
Hits:	2	
Reads:		8
Writes:		2

Write-Back (REF 7)

Instructions:

LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	M
SB x1 → M[5]	Hit
SB x1 → M[10]	

lru	V	d	tag	data
0	1	0	101	33
				28
1	1	1	010	71
				29

Cache

Register File

x0	
x1	29
x2	33
x3	

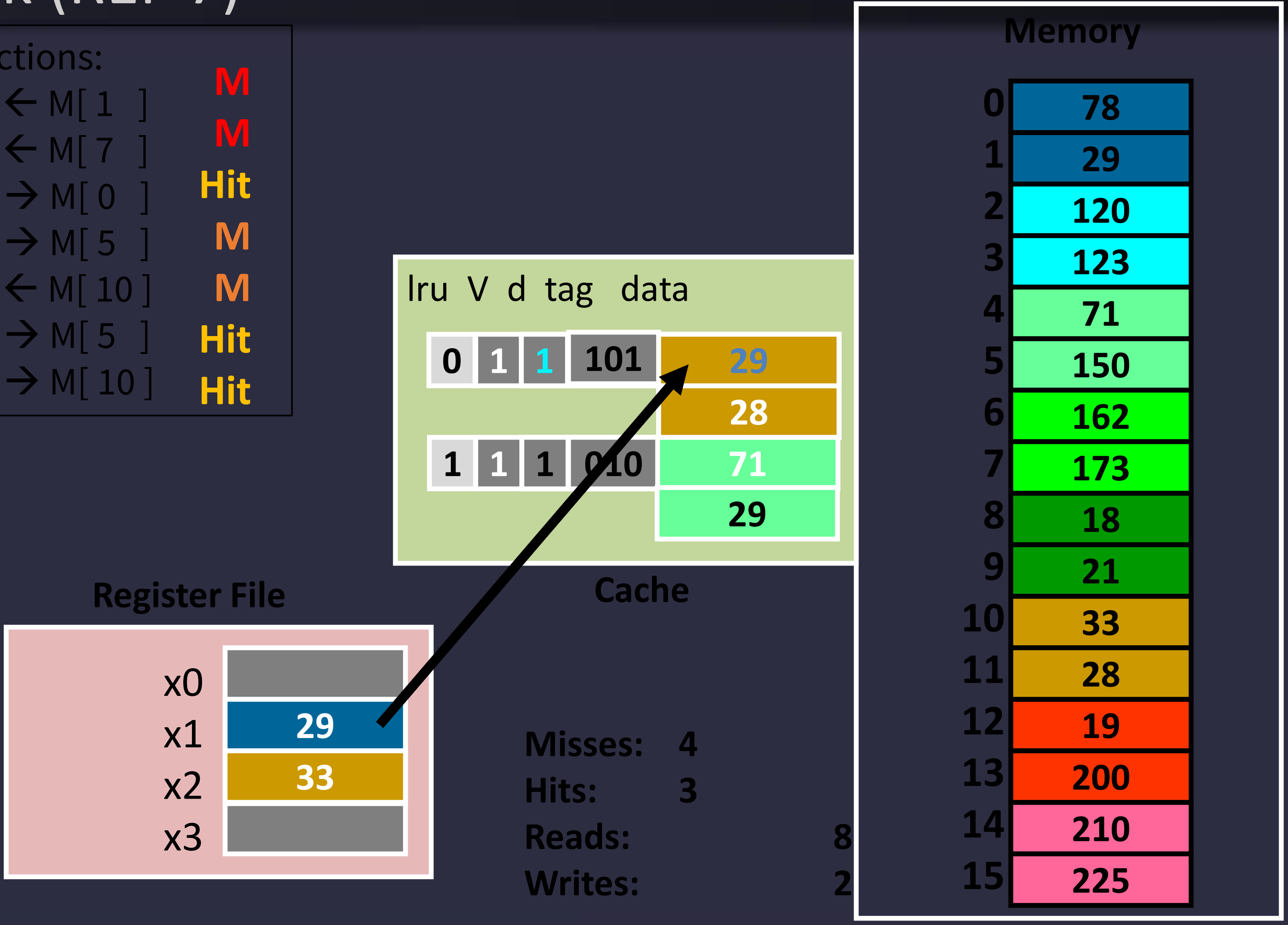
Misses: 4
Hits: 2
Reads: 8
Writes: 2

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 7)

Instructions:	
LB x1 ← M[1]	M
LB x2 ← M[7]	M
SB x2 → M[0]	Hit
SB x1 → M[5]	M
LB x2 ← M[10]	M
SB x1 → M[5]	Hit
SB x1 → M[10]	Hit



Cache

lru	V	d	tag	data
0	1	1	101	29 28
1	1	1	010	71 29

Register File

x0	
x1	29
x2	33
x3	

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Misses:	4	
Hits:	3	
Reads:		8
Writes:		2

Write-Back (REF 8,9)

Instructions:

...

SB $1 → M[5]

LB $2 ← M[10]

SB $1 → M[5]

SB $1 → M[10]

SB $1 → M[5]

SB $1 → M[10]

M

M

Hit

M

M

Hit

Hit

Cheap subsequent updates!

lru	V	d	tag	data
0	1	1	101	29
				28
1	1	1	010	71
				29

Cache

Register File

x0	
x1	29
x2	33
x3	

Misses: 4

Hits: 3

Reads: 8

Writes: 2

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

Write-Back (REF 8,9)

Instructions:

…

SB $1 → M[5]

LB $2 ← M[10]

SB $1 → M[5]

SB $1 → M[10]

SB $1 → M[5]

SB $1 → M[10]

M

M

Hit

M

M

Hit

Hit

Hit

Hit

lru	V	d	tag	data
0	1	1	101	29
				28
1	1	1	010	71
				29

Cache

Register File

x0	
x1	29
x2	33
x3	

Misses: 4

Hits: 3

Reads: 8

Writes: 2

Memory

0	78
1	29
2	120
3	123
4	71
5	150
6	162
7	173
8	18
9	21
10	33
11	28
12	19
13	200
14	210
15	225

How Many Memory References?

Write-back performance

- How many reads?
  - Each miss (read or write) reads a block from mem
  - 4 misses → 8 mem reads
- How many writes?
  - *Some* evictions write a block to mem
  - 1 dirty eviction → 2 mem writes

Write-back vs. Write-through Example

Assume: large associative cache, 16-byte lines

N 4-byte words

```
for (i=1; i<n; i++)
   A[0] += A[i];
```

Write-thru: n reads (n/4 cache lines)
n writes
Write-back: n reads (n/4 cache lines)
4 writes (one cache line)

```
for (i=0; i<n; i++)
     B[i] = A[i]
```

Write-thru: n reads (n/4 cache lines)
n writes
Write-back: n reads (n/4 cache lines)
n writes (n/4 cache lines)

So is write back just better?

Short Answer: Yes (fewer writes is a good thing)

Long Answer: It’s complicated.

- Evictions require entire line be written back to memory (vs. just the data that was written)
- Write-back can lead to incoherent caches on multi-core processors

Optimization: Write Buffering

- Q: Writes to main memory are **slow!**
- A: Use a **write-back buffer**
  - A small queue holding dirty lines
  - Add to end upon eviction
  - Remove from front upon completion

- Q: When does it help?
- A: short bursts of writes (but not sustained writes)
- A: fast eviction reduces miss penalty

Write-through vs. Write-back

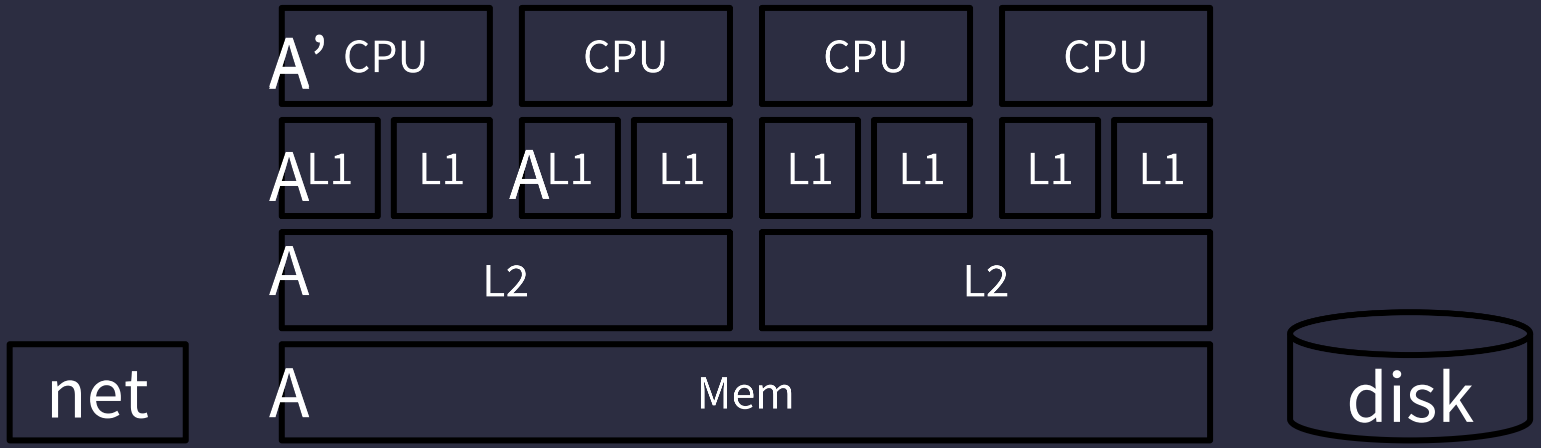
- Write-through is slower
  - But simpler (memory always consistent)

- Write-back is almost always faster
  - write-back buffer hides large eviction cost
  - But what about multiple cores with separate caches but sharing memory?
- **Write-back requires a cache coherency protocol**
  - Inconsistent views of memory
  - Need to “snoop” in each other’s caches
  - Extremely complex protocols, very hard to get right

Cache-coherency

Q: Multiple readers and writers?

A: Potentially inconsistent views of memory



Cache coherency protocol

- May need to **snoop** on other CPU’s cache activity
- **Invalidate** cache line when other CPU writes
- **Flush** write-back caches before other CPU reads
- Or the reverse: Before writing/reading…
- Extremely complex protocols, very hard to get right

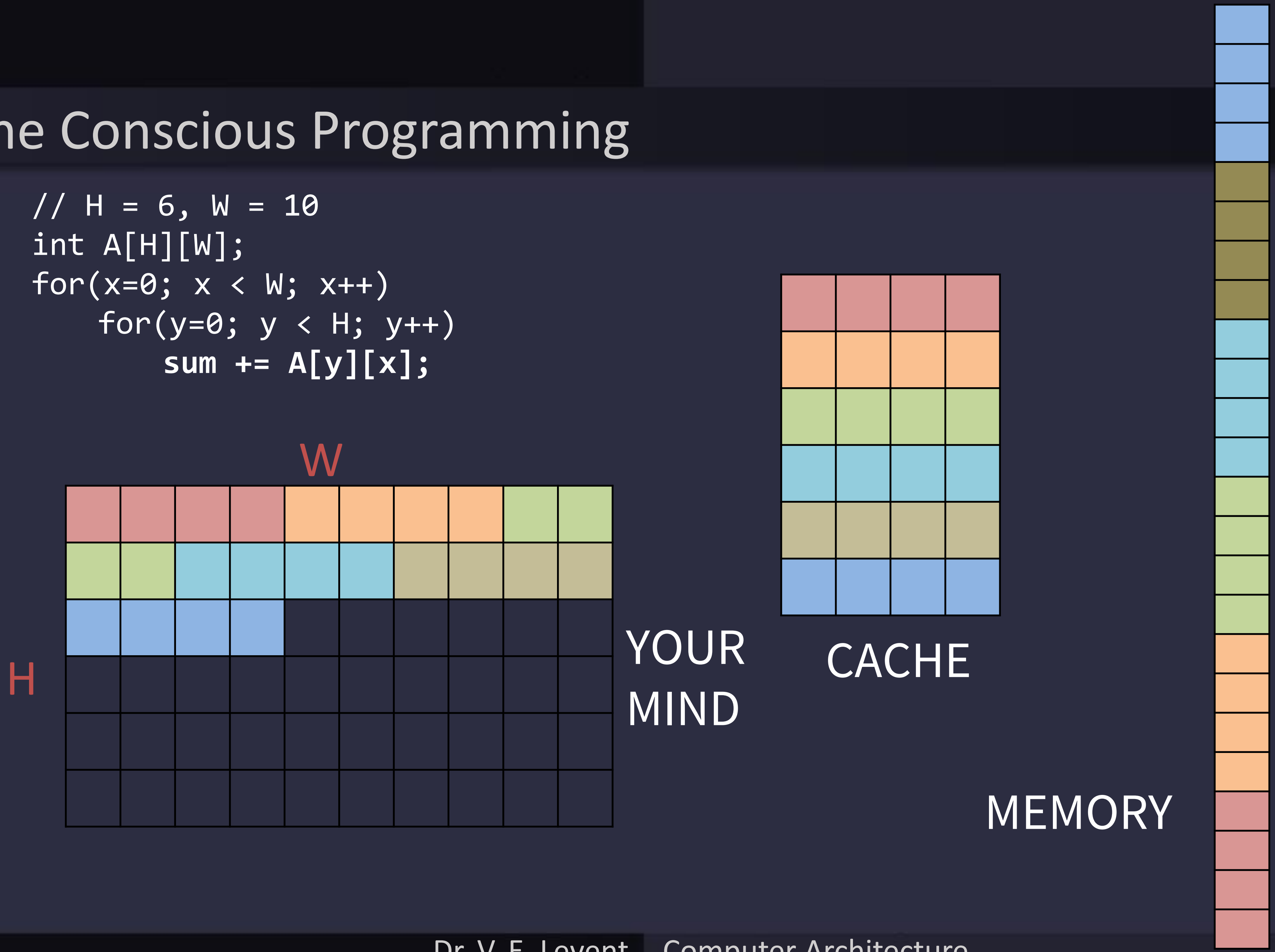
Takeaway

- Write-through policy with write allocate
  - Cache miss: read entire block from memory
  - Write: write only updated item to memory
  - Eviction: no need to write to memory
  - **Slower, but cleaner**

- Write-back policy with write allocate
  - Cache miss: read entire block from memory
    - **But may need to write dirty cacheline first**
  - Write: nothing to memory
  - Eviction: have to write to memory *entire cacheline* because don’t know what is dirty (only 1 dirty bit)
  - **Faster, but more complicated, especially with multicore**

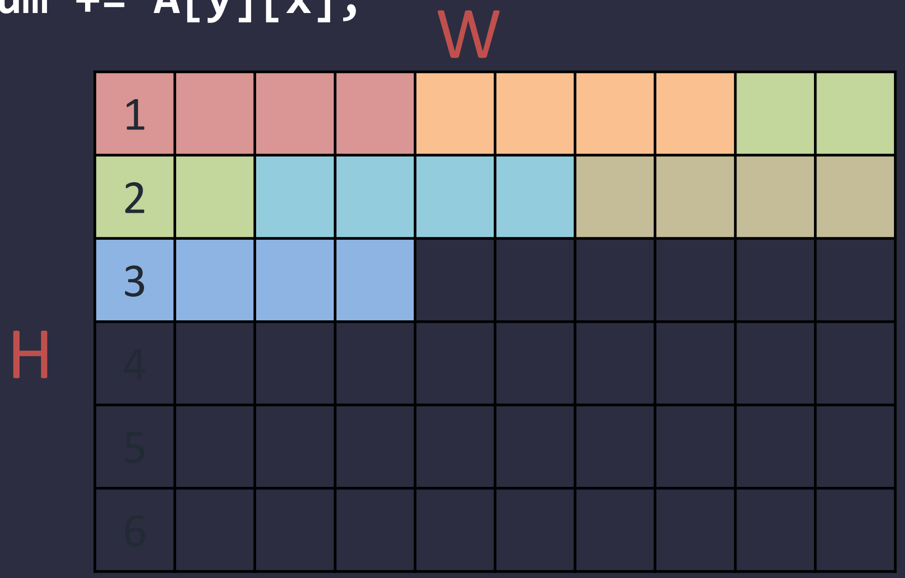
Cache Conscious Programming

```
// H = 6, W = 10
int A[H][W];
for(x=0; x < W; x++)
    for(y=0; y < H; y++)
        sum += A[y][x];
```

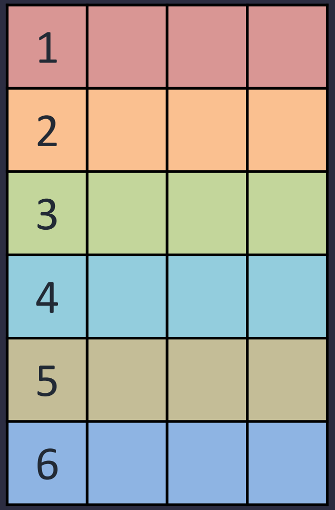


Cache Conscious Programming

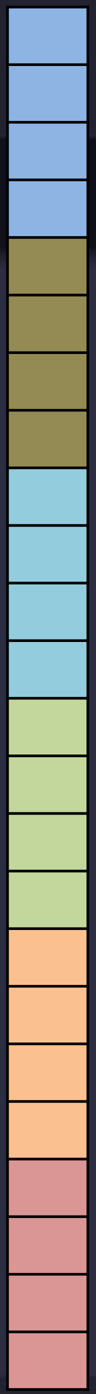
```
// H = 6, W = 10
int A[H][W];
for(x=0; x < W; x++)
    for(y=0; y < H; y++)
        sum += A[y][x];
```



YOUR MIND



CACHE

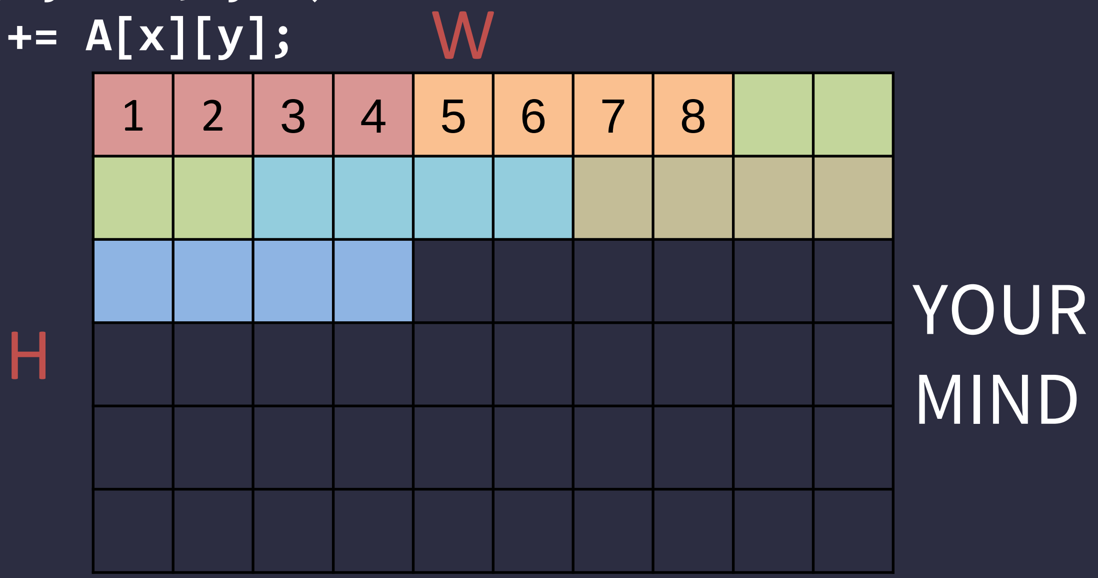


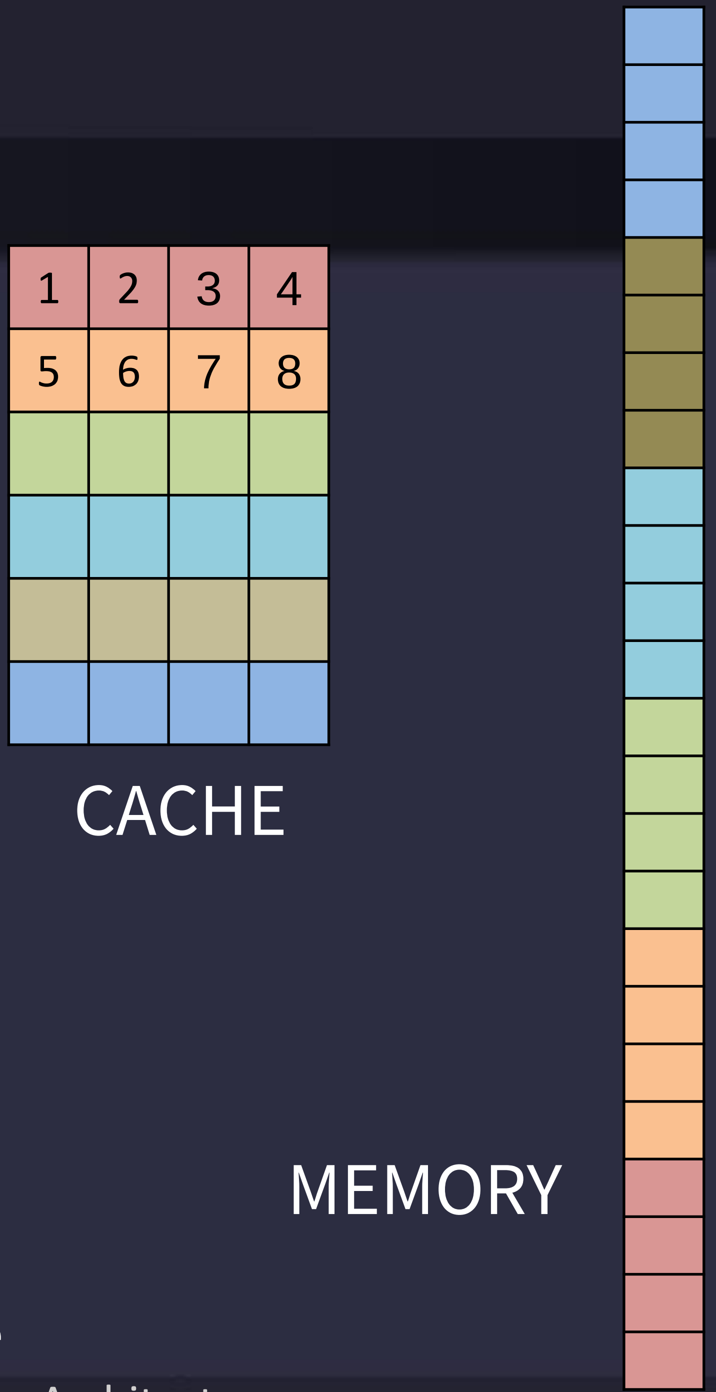
MEMORY

Every access a cache miss! (unless *entire* matrix fits in cache)

Cache Conscious Programming

```
// H = 6, W = 10
int A[H][W];
for(x=0; x < H; x++)
    for(y=0; y < W; y++)
        sum += A[x][y];
```





- Block size = 4 → 75% hit rate
- Block size = 8 → 87.5% hit rate
- Block size = 16 → 93.75% hit rate
- And you can easily prefetch to warm the cache

A Real Example

- Dual 32K L1 Instruction caches
  - 8-way set associative
  - 64 sets
  - 64 byte line size
- Dual 32K L1 Data caches
  - Same as above
- Single 6M L2 Unified cache
  - 24-way set associative (!!!)
  - 4096 sets
  - 64 byte line size
- 4GB Main memory
- 1TB Disk