CS 354
Machine Organization and Programming

Lecture 24

Michael Doescher
Summer 2020

Cache Memories

Cache memories
A Store :: To Store

Memory Hierarchy

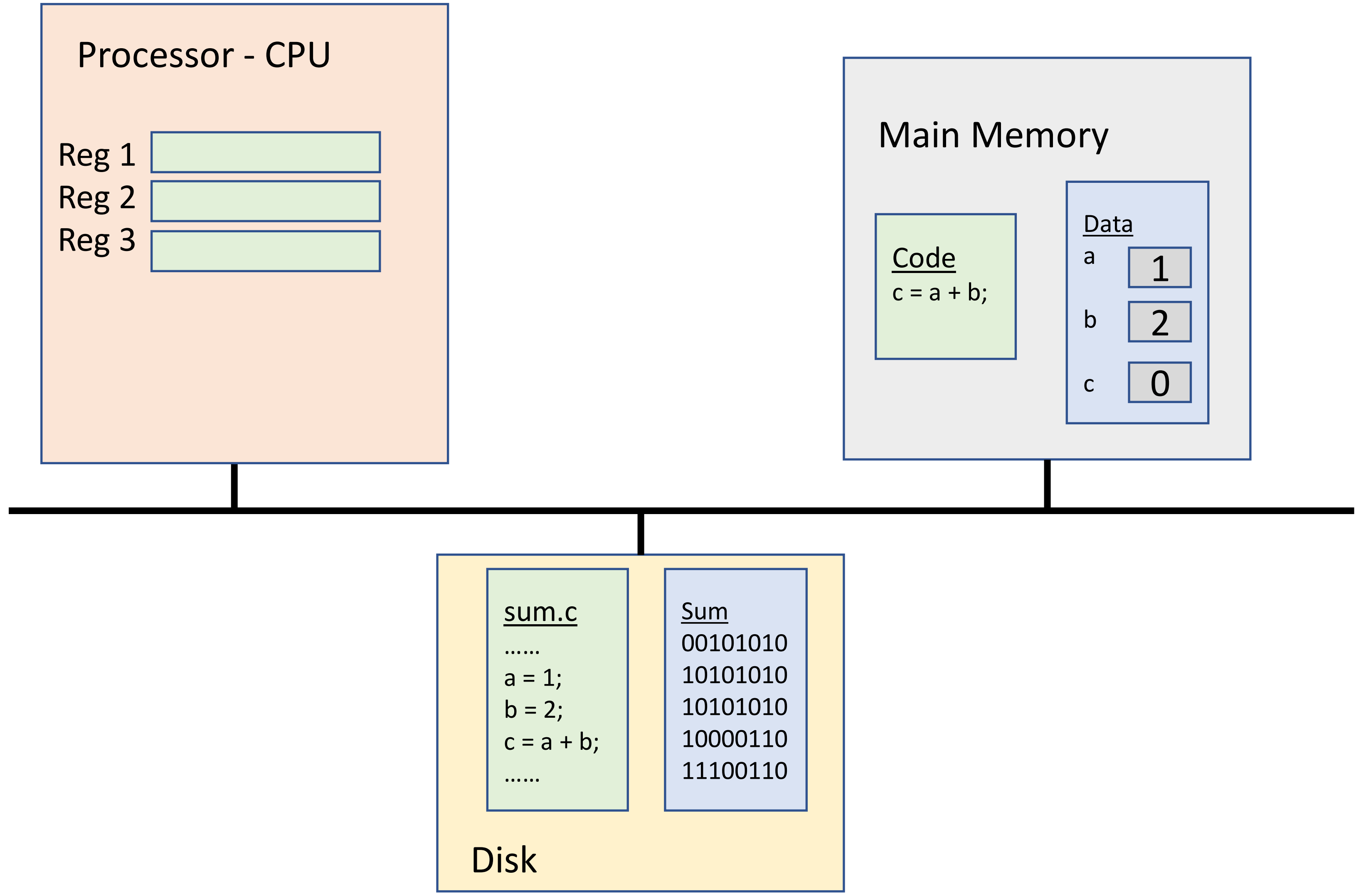
Locality

Caches

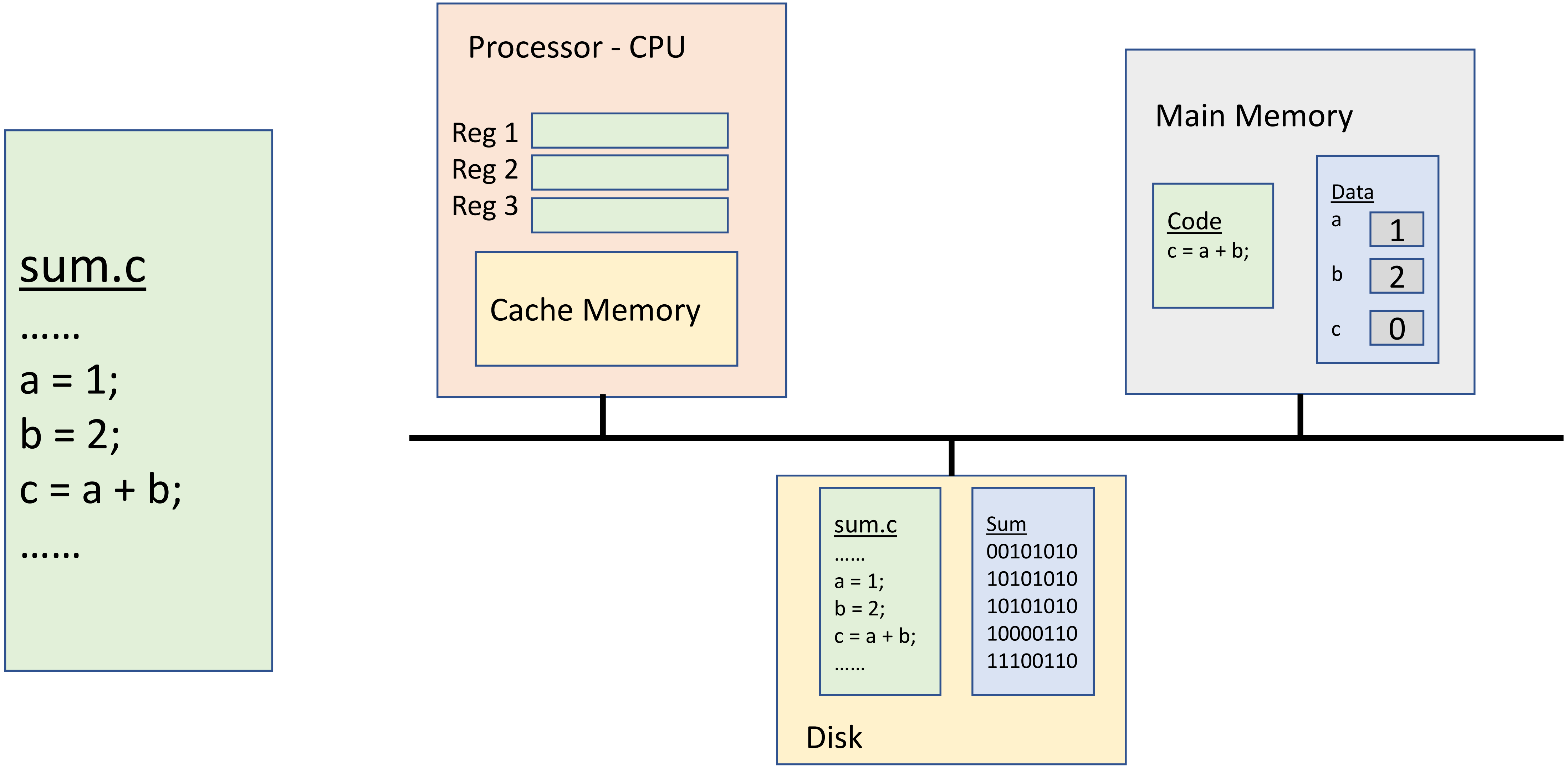
Writing Cache Efficient Code

What happens when we run a program?

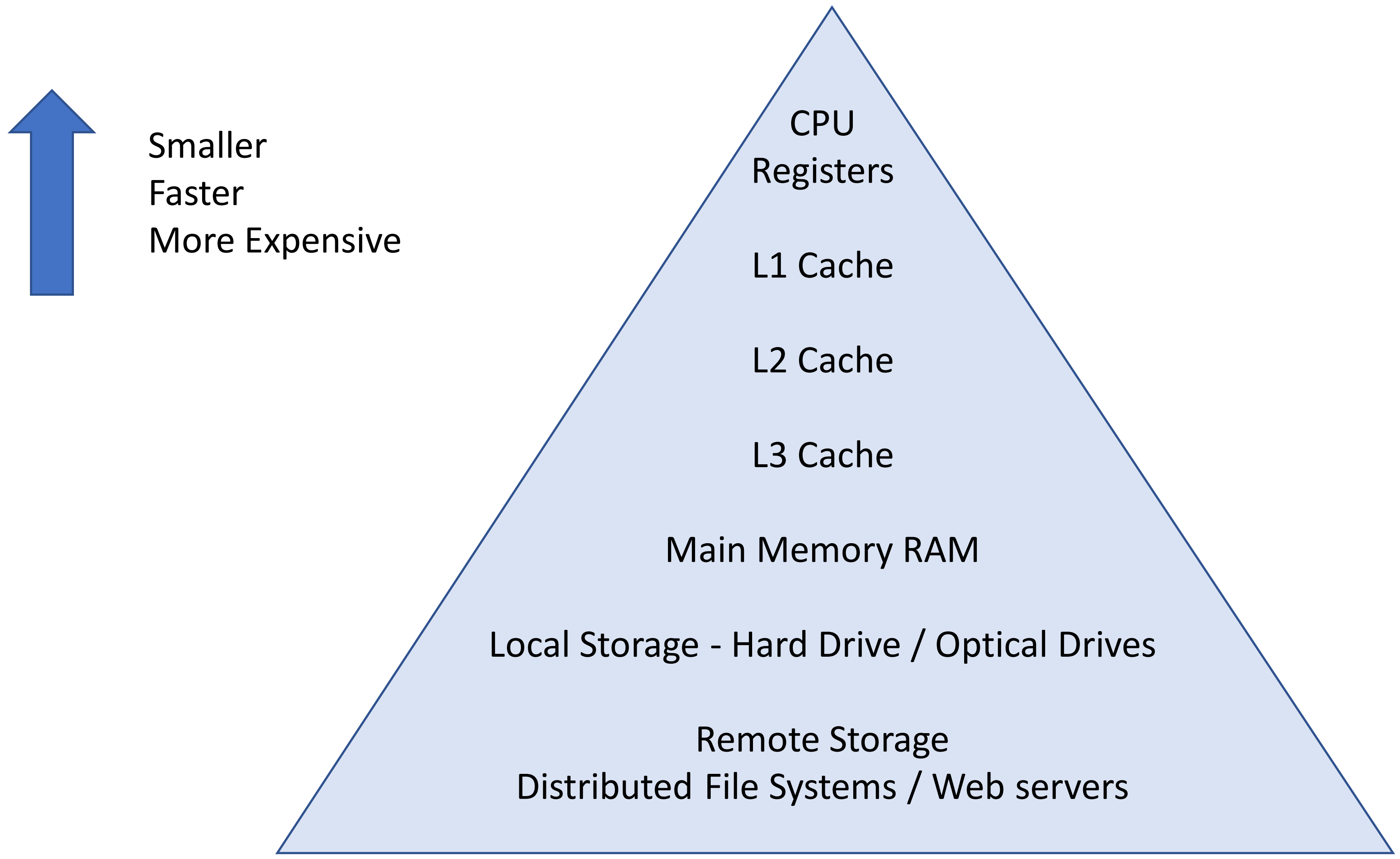
```
sum.c
……
a = 1;
b = 2;
c = a + b;
……
```



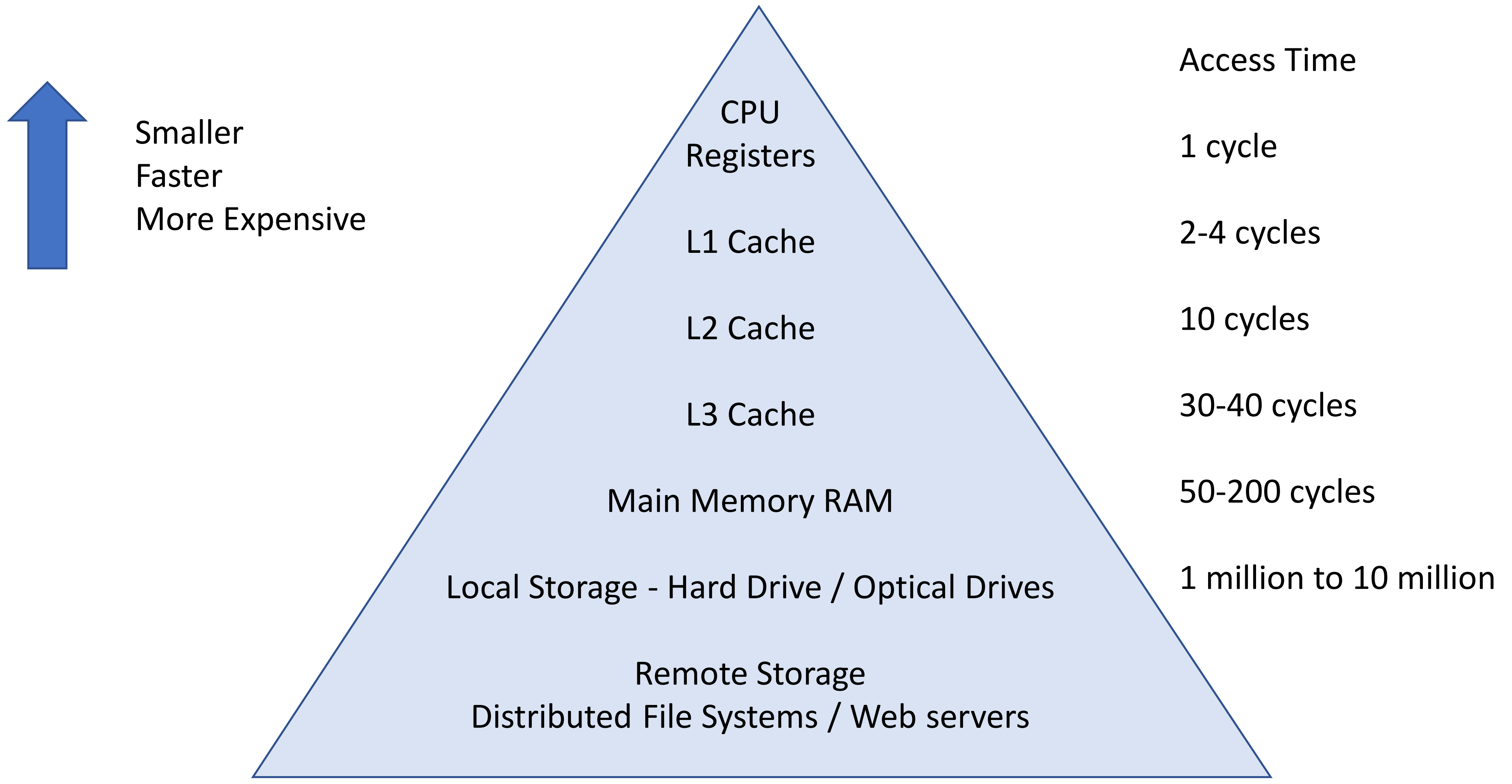
What happens when we run a program?



Memory Hierarchy



Memory Hierarchy



Locality

Temporal

Spatial

Locality

Temporal
access the same data many times in a short time period

Spatial
access sets of data items stored at nearby memory addresses

Locality

Temporal
access the same data many times in a short time period

Spatial
access sets of data items stored at nearby memory addresses

```
for (int i=0; i<n; i++) {
        sum+= arr[i]
}
```

Locality

Temporal
access the same data many times in a short time period

i, n, sum

Spatial
access sets of data items stored at nearby memory addresses
arr[ ]

```
for (int i=0; i<n; i++) {
        sum+= arr[i]
}
```

Locality

Temporal
access the same data many times in a short time period

Spatial
access sets of data items stored at nearby memory addresses

```
for (int i=0; i<n; i++) {
        sum+= arr[i]
}
```

Cache memories that can take advantage of locality can make your code run more quickly by avoiding costly access to slow memory

Reference Patterns

```
arr
[  11, 12, 13, 14, 15
   21, 22, 23, 24, 25
   31, 32, 33, 34, 35 ]
```

addr	**Val**
0x100	11
0x104	12
0x108	13
0x10C	14
0x110	15
0x114	21
0x118	22
0x11C	23
0x120	24
0x124	25
0x128	31
0x12C	32
0x130	33
0x134	34
0x138	35

Reference Patterns

```
arr
[  11, 12, 13, 14, 15
   21, 22, 23, 24, 25
   31, 32, 33, 34, 35 ]
```

```
Row order
for (int i=0; i<3;i++)
    for (int j=0; j<5; j++)
       sum+= arr[i][j]
```

addr	**Val**
0x100	11
0x104	12
0x108	13
0x10C	14
0x110	15
0x114	21
0x118	22
0x11C	23
0x120	24
0x124	25
0x128	31
0x12C	32
0x130	33
0x134	34
0x138	35

Reference Patterns

```
arr
[  11, 12, 13, 14, 15
   21, 22, 23, 24, 25
   31, 32, 33, 34, 35 ]
```

```
Row order
for (int i=0; i<3;i++)
    for (int j=0; j<5; j++)
       sum+= arr[i][j]
```

```
Column order
for (int j=0; j<5; j++)
    for (int i=0; i<3;i++)
       sum+= arr[i][j]
```

addr	**Val**
0x100	11
0x104	12
0x108	13
0x10C	14
0x110	15
0x114	21
0x118	22
0x11C	23
0x120	24
0x124	25
0x128	31
0x12C	32
0x130	33
0x134	34
0x138	35

Reference Patterns

```
arr
[  11, 12, 13, 14, 15
   21, 22, 23, 24, 25
   31, 32, 33, 34, 35 ]
```

```
Row order
for (int i=0; i<3;i++)
    for (int j=0; j<5; j++)
       sum+= arr[i][j]
```

```
Column order
for (int j=0; j<5; j++)
    for (int i=0; i<3;i++)
       sum+= arr[i][j]
```

```
stride 1
sequential
reference pattern
```

addr	**Val**
0x100	11
0x104	12
0x108	13
0x10C	14
0x110	15
0x114	21
0x118	22
0x11C	23
0x120	24
0x124	25
0x128	31
0x12C	32
0x130	33
0x134	34
0x138	35

Reference Patterns

```
arr
[  11, 12, 13, 14, 15
   21, 22, 23, 24, 25
   31, 32, 33, 34, 35 ]
```

```
Row order
for (int i=0; i<3;i++)
    for (int j=0; j<5; j++)
       sum+= arr[i][j]
```

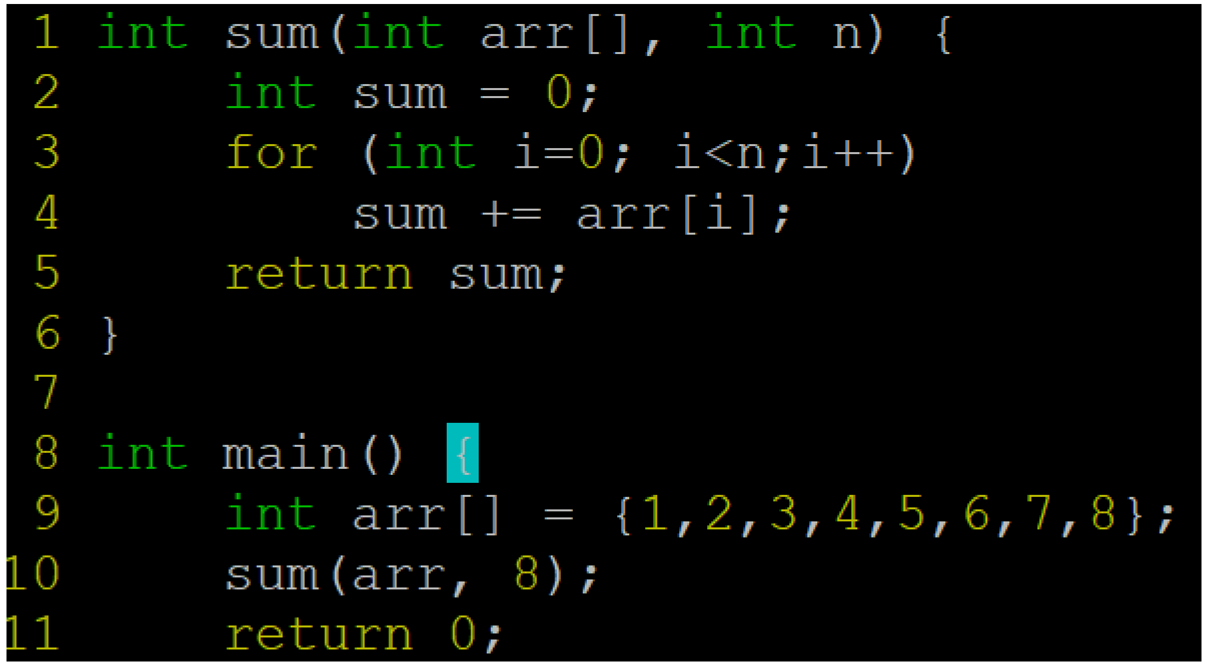
stride 1
sequential
reference pattern

```
Column order
for (int j=0; j<5; j++)
    for (int i=0; i<3;i++)
       sum+= arr[i][j]
```

stride k
reference pattern

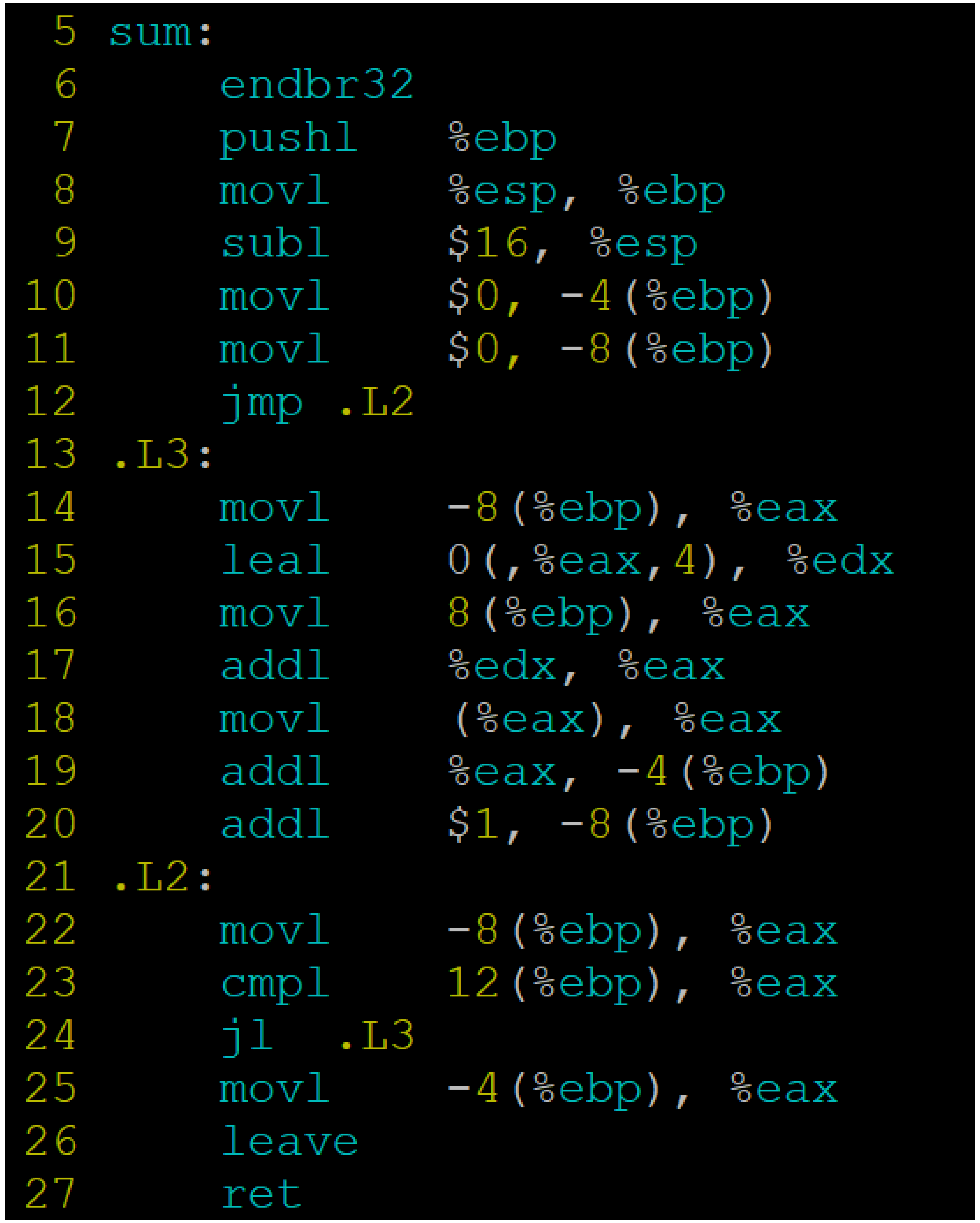
addr	Val
0x100	11
0x104	12
0x108	13
0x10C	14
0x110	15
0x114	21
0x118	22
0x11C	23
0x120	24
0x124	25
0x128	31
0x12C	32
0x130	33
0x134	34
0x138	35

Locality applies to Instructions too



```c
int sum(int arr[], int n) {
    int sum = 0;
    for (int i=0; i<n;i++)
        sum += arr[i];
    return sum;
}

int main() {
    int arr[] = {1,2,3,4,5,6,7,8};
    sum(arr, 8);
    return 0;
```



```
sum:
        endbr32
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    $0, -4(%ebp)
        movl    $0, -8(%ebp)
        jmp .L2
.L3:
        movl    -8(%ebp), %eax
        leal    0(,%eax,4), %edx
        movl    8(%ebp), %eax
        addl    %edx, %eax
        movl    (%eax), %eax
        addl    %eax, -4(%ebp)
        addl    $1, -8(%ebp)
.L2:
        movl    -8(%ebp), %eax
        cmpl    12(%ebp), %eax
        jl  .L3
        movl    -4(%ebp), %eax
        leave
        ret
```

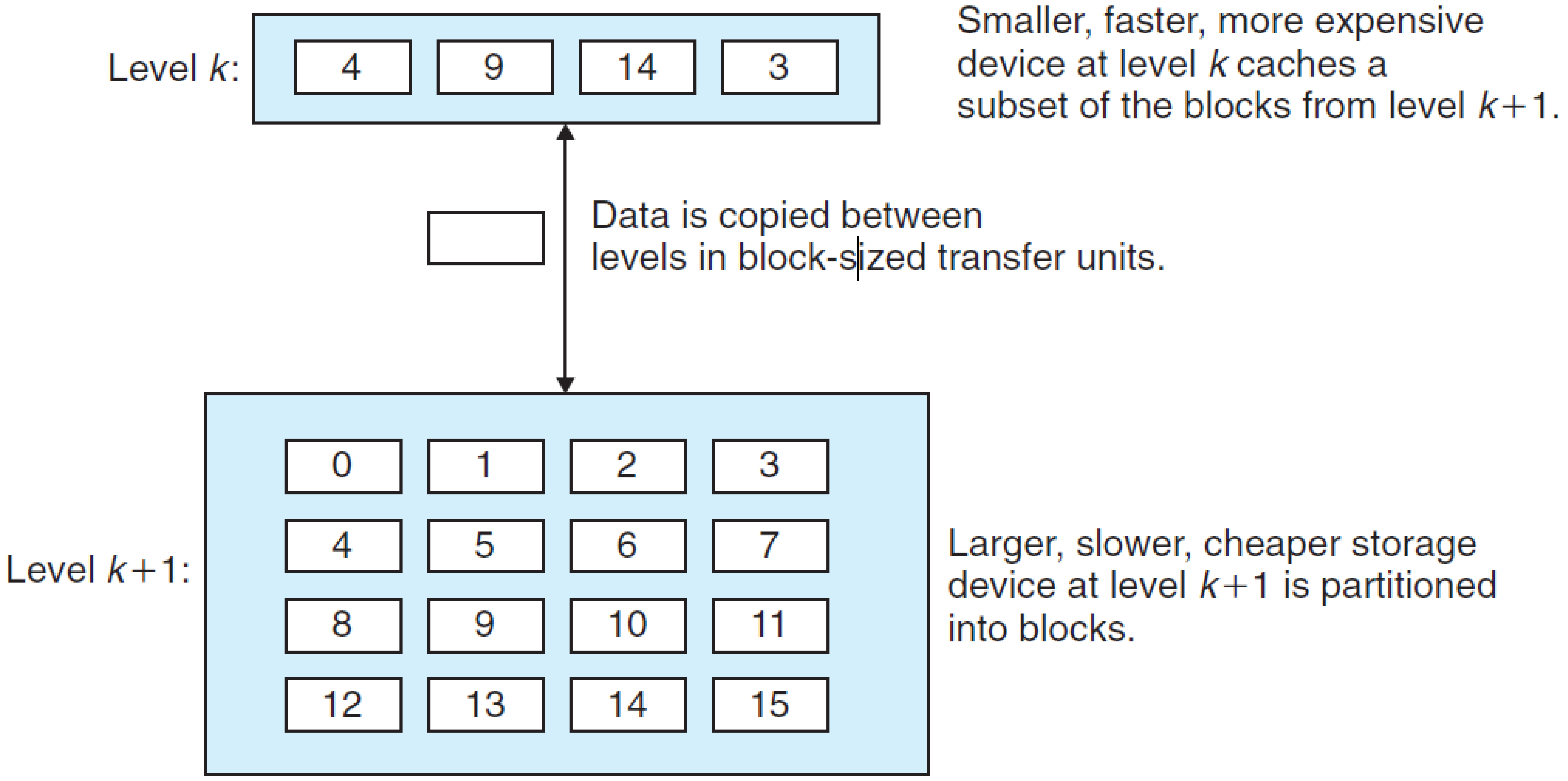
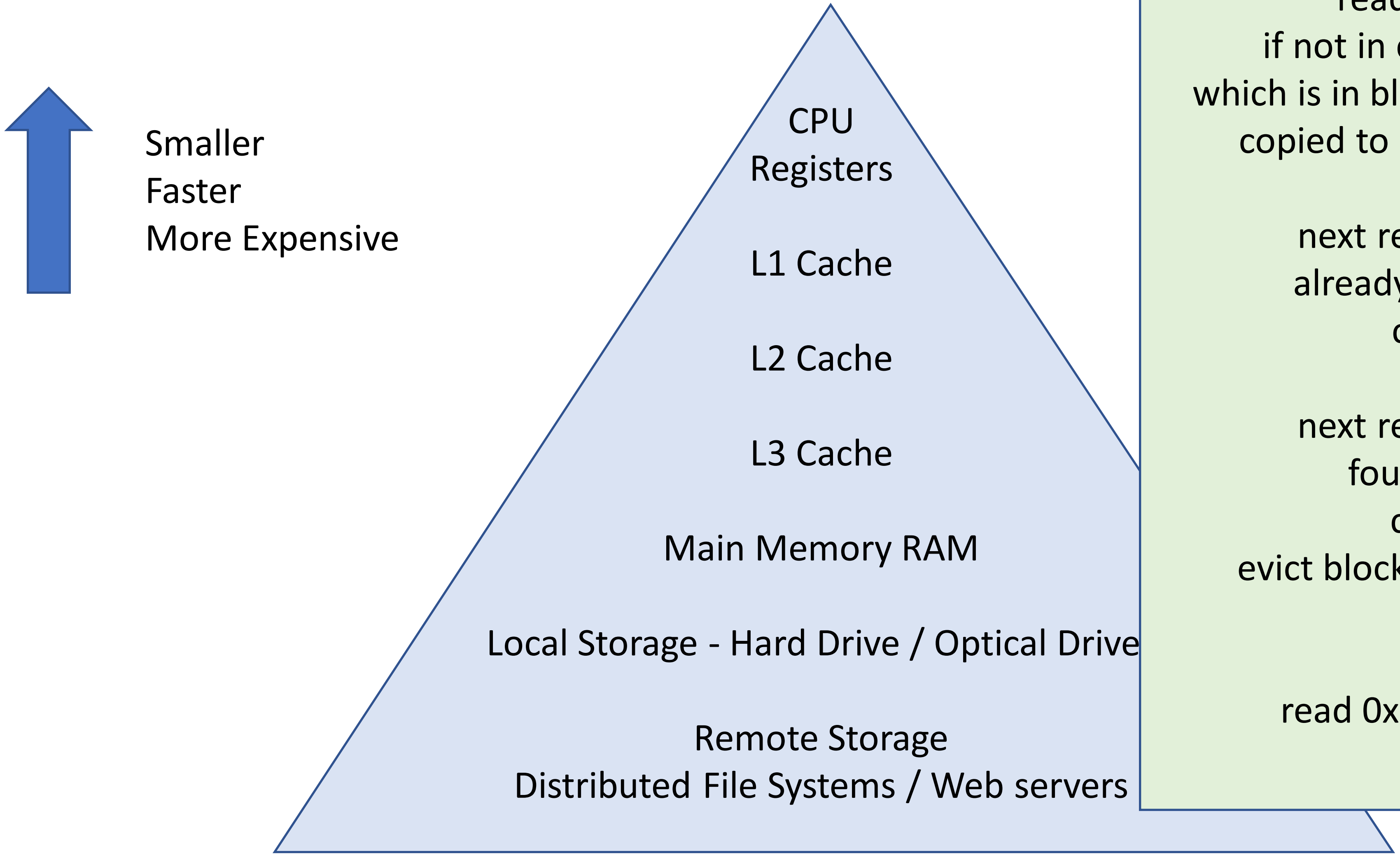

Level *k*:
4
9
14
3
Smaller, faster, more expensive device at level *k* caches a subset of the blocks from level *k*+1.
Data is copied between levels in block-sized transfer units.
Level *k*+1:
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Larger, slower, cheaper storage device at level *k*+1 is partitioned into blocks.

Figure 6.24 **The basic principle of caching in a memory hierarchy.**

Memory Hierarchy



read 0x12345678
if not in cache : cache miss
which is in block 4 of main memory
copied to L3, then L2, then L1

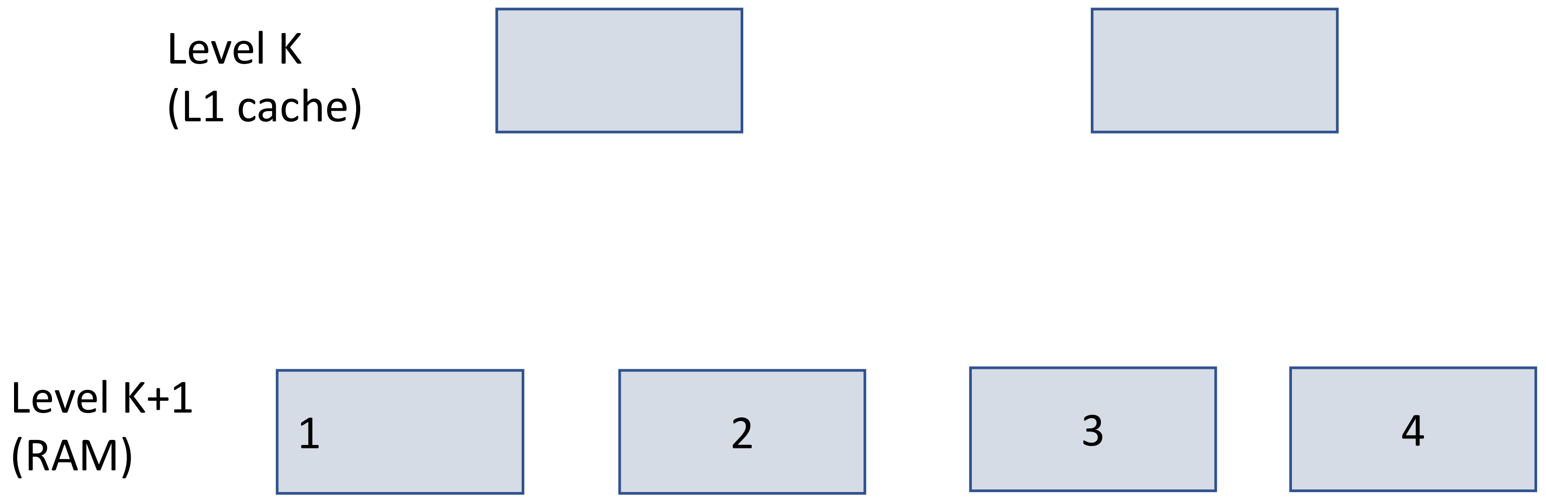
next read 0x12345679
already in block 4 in L1
cache hit!!

next read 0x87658321
found in block 8
cache miss
evict block 4 and copy block 8

read 0x12345678 again?

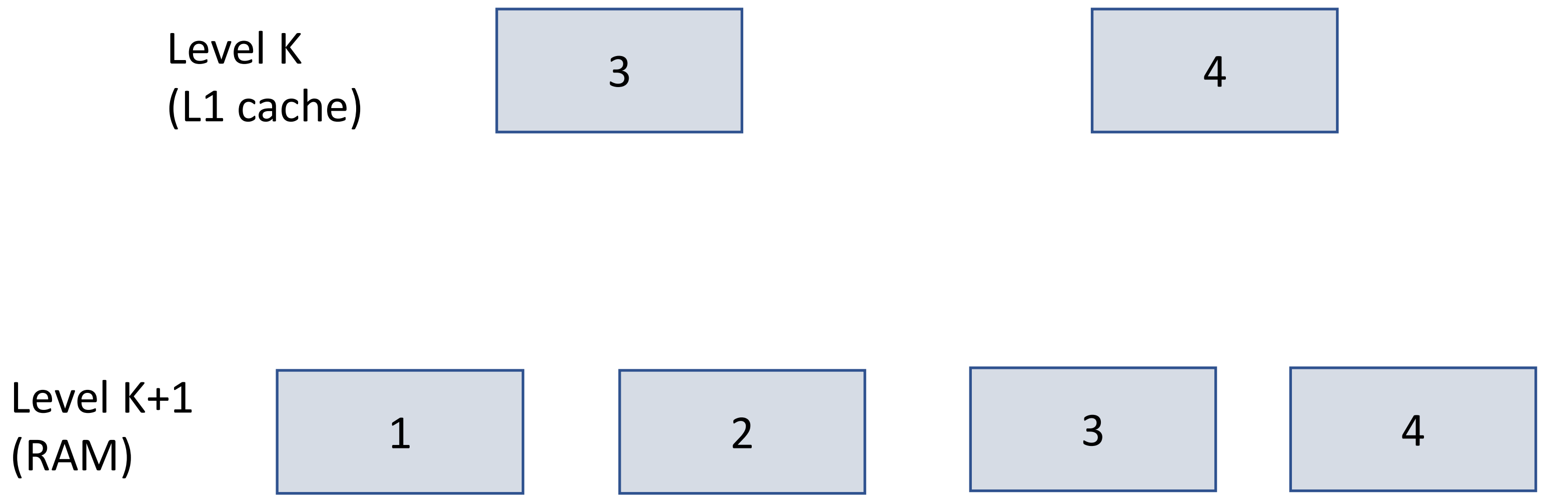
Example

CPU Registers
L1 Cache
RAM



Example

CPU Registers
L1 Cache
RAM



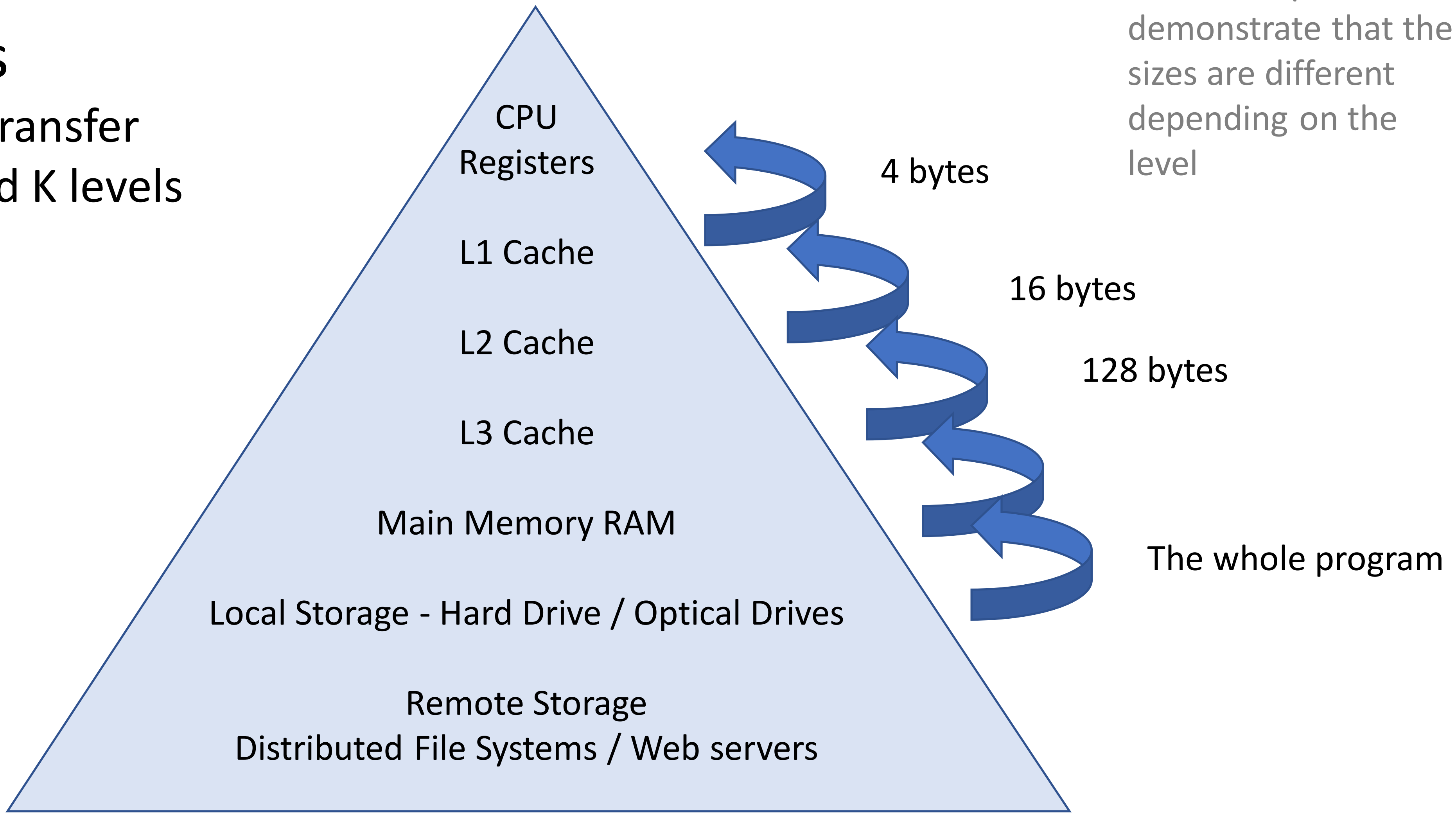
Access Pattern:
1, 2, 2,
1, 3, 1, 3, 1,
1, 2, 3, 4, 1, 2, 3, 4

Cache Misses:
1. Cold Miss (Compulsory miss)
2. Conflict misses
3. Capacity Misses (working set is larger than the cache)

Blocks

Blocks

Units of data transfer between K+1 and K levels



I made these numbers up to demonstrate that the sizes are different depending on the level

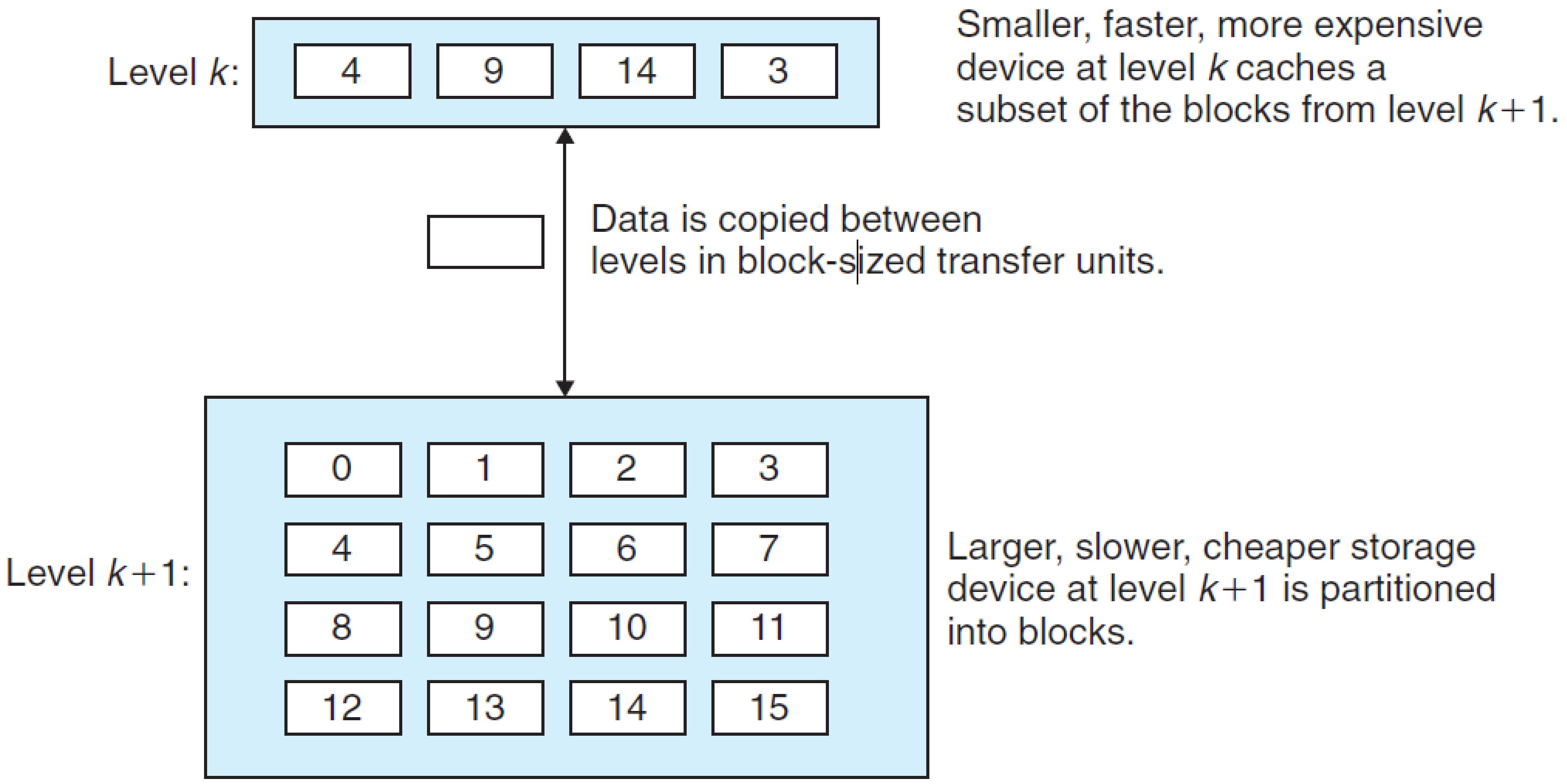
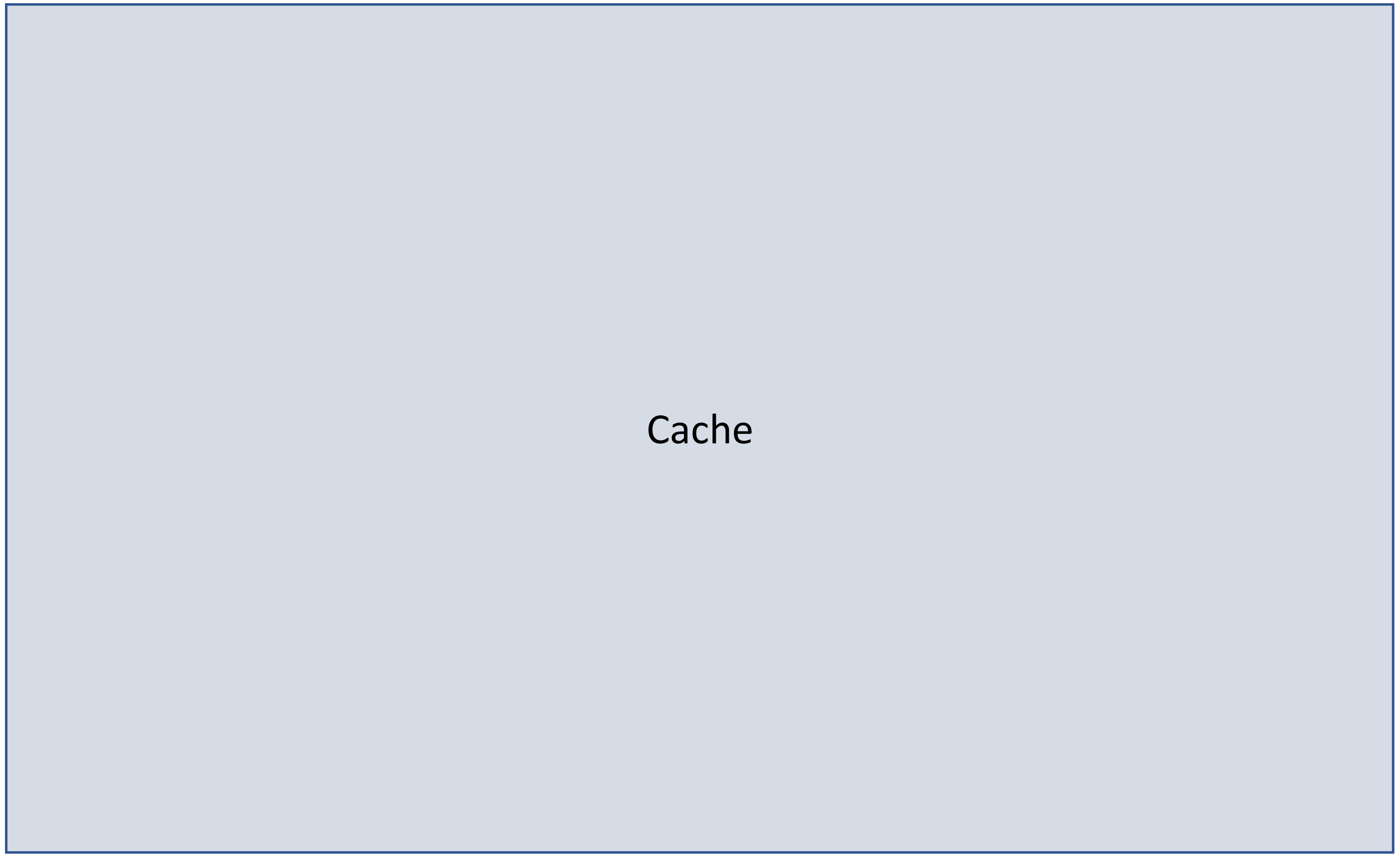

Level k:
4
9
14
3
Smaller, faster, more expensive device at level k caches a subset of the blocks from level k+1.
Data is copied between levels in block-sized transfer units.
Level k+1:
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Larger, slower, cheaper storage device at level k+1 is partitioned into blocks.

Figure 6.24 **The basic principle of caching in a memory hierarchy.**

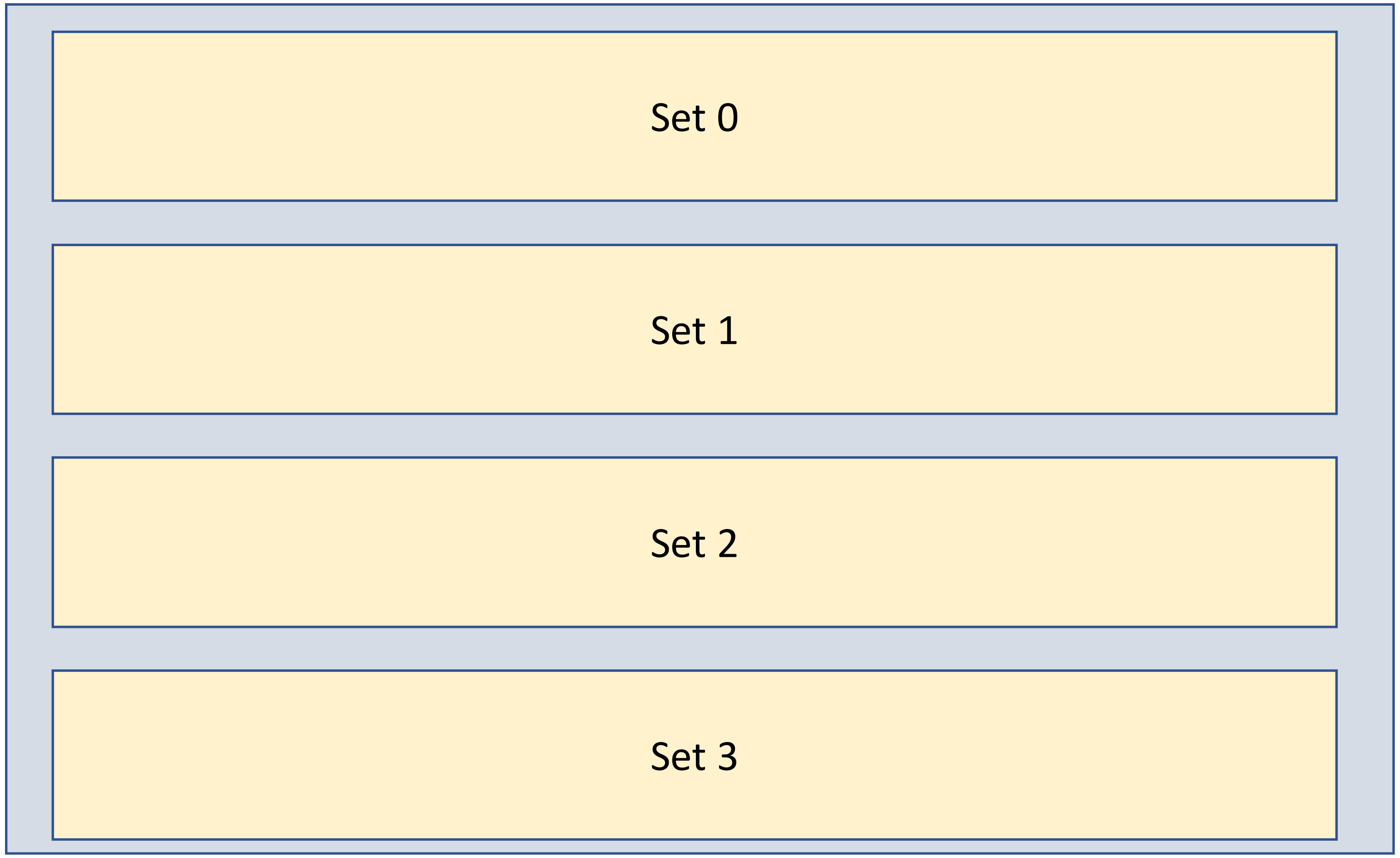
Type	What cached	Where cached	Latency (cycles)	Managed by
CPU registers	4-byte or 8-byte word	On-chip CPU registers	0	Compiler
TLB	Address translations	On-chip TLB	0	Hardware MMU
L1 cache	64-byte block	On-chip L1 cache	1	Hardware
L2 cache	64-byte block	On/off-chip L2 cache	10	Hardware
L3 cache	64-byte block	On/off-chip L3 cache	30	Hardware
Virtual memory	4-KB page	Main memory	100	Hardware + OS
Buffer cache	Parts of files	Main memory	100	OS
Disk cache	Disk sectors	Disk controller	100,000	Controller firmware
Network cache	Parts of files	Local disk	10,000,000	AFS/NFS client
Browser cache	Web pages	Local disk	10,000,000	Web browser
Web cache	Web pages	Remote server disks	1,000,000,000	Web proxy server

Figure 6.25 **The ubiquity of caching in modern computer systems.** Acronyms: TLB: translation lookaside buffer, MMU: memory management unit, OS: operating system, AFS: Andrew File System, NFS: Network File System.

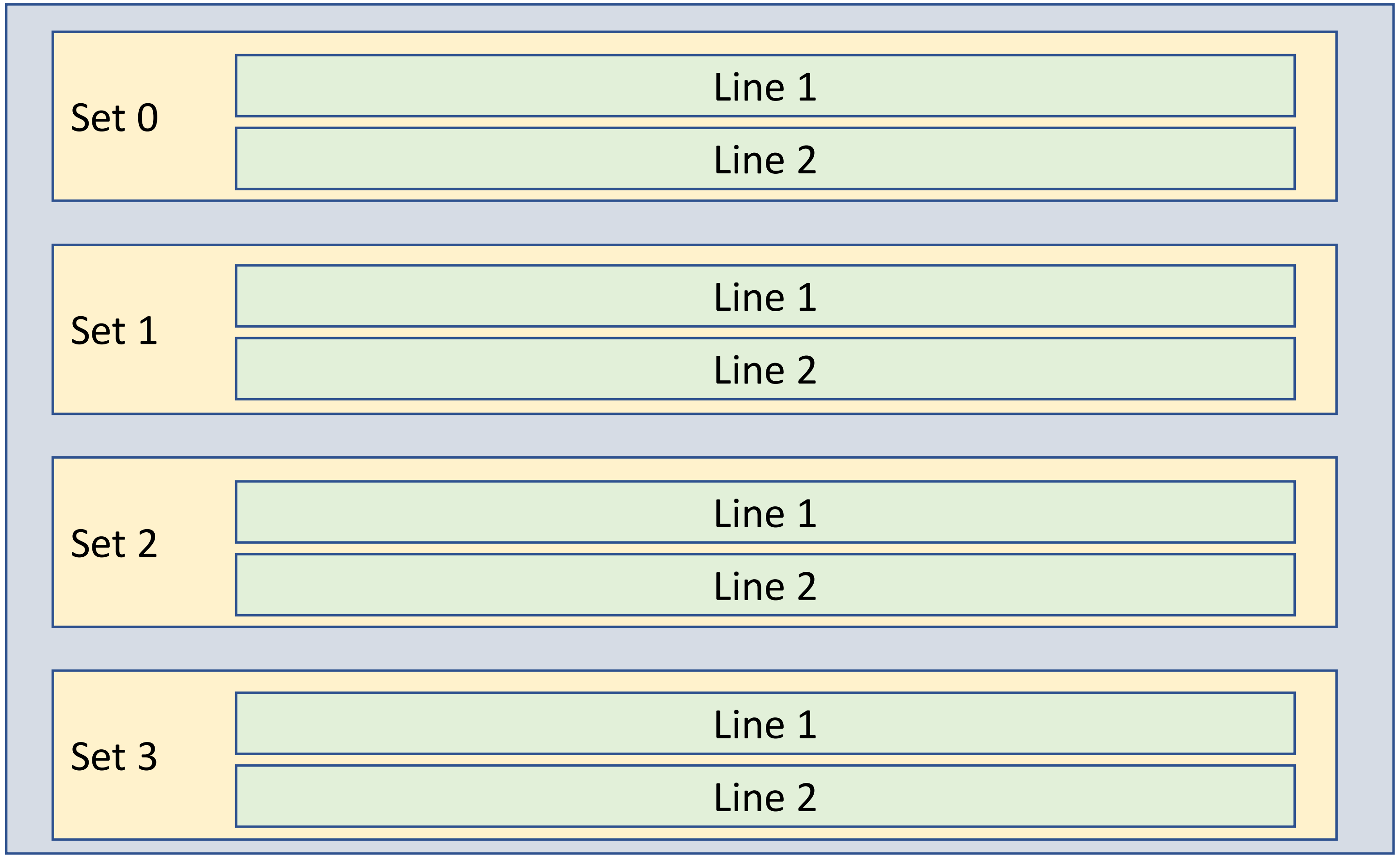
Cache Organization



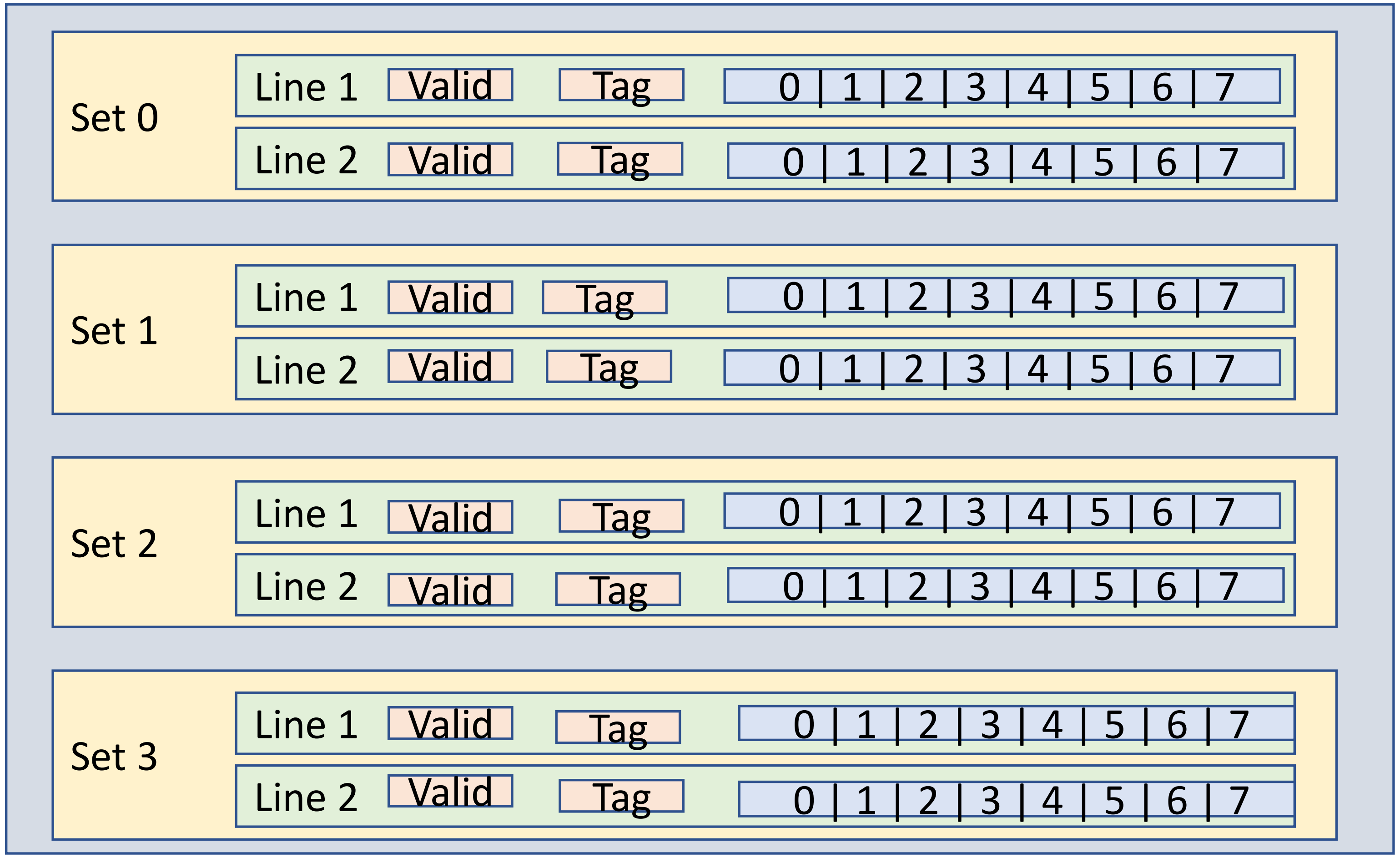
Cache Organization



Cache Organization

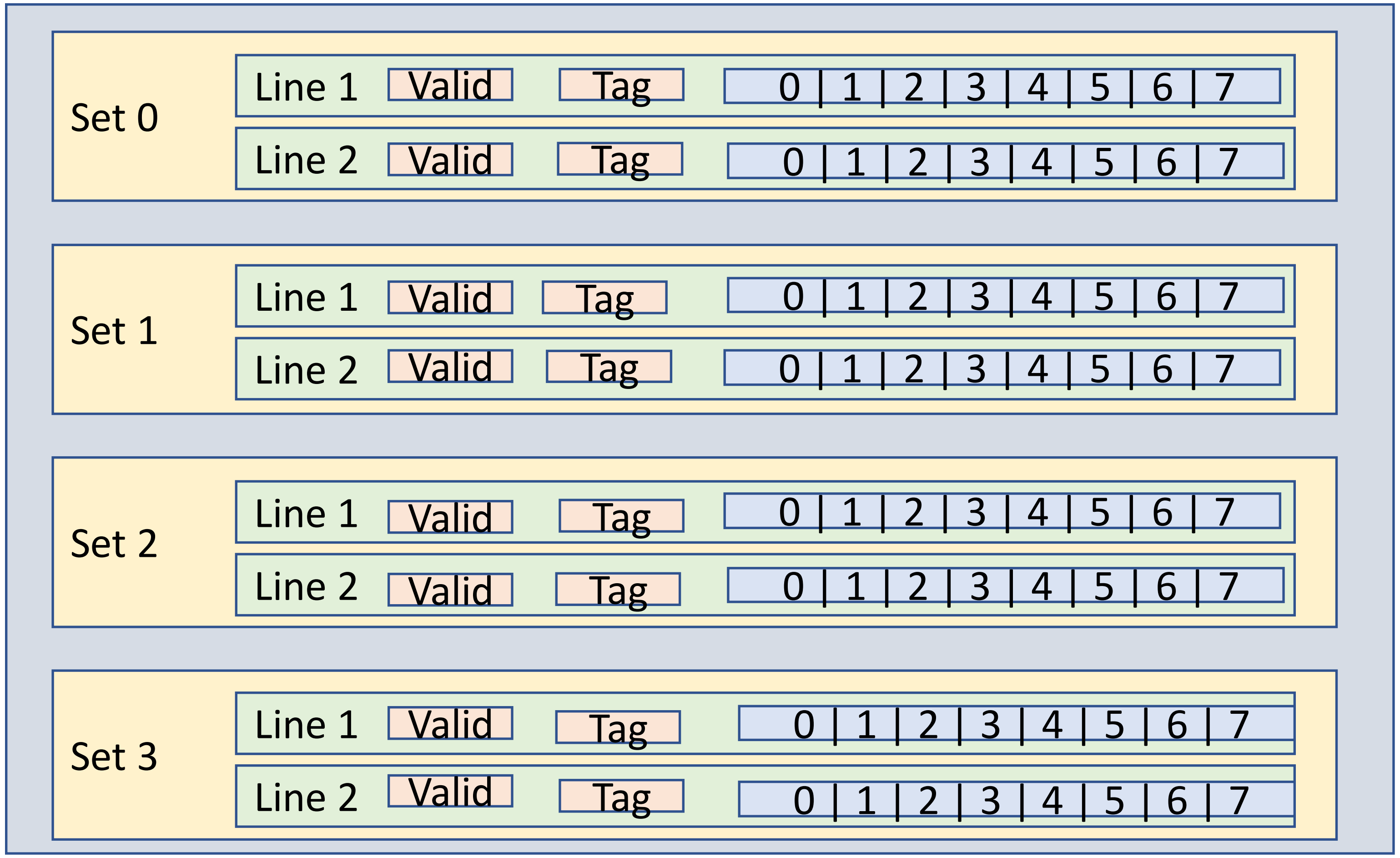


Cache Organization


Set 0
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 1
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 2
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 3
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

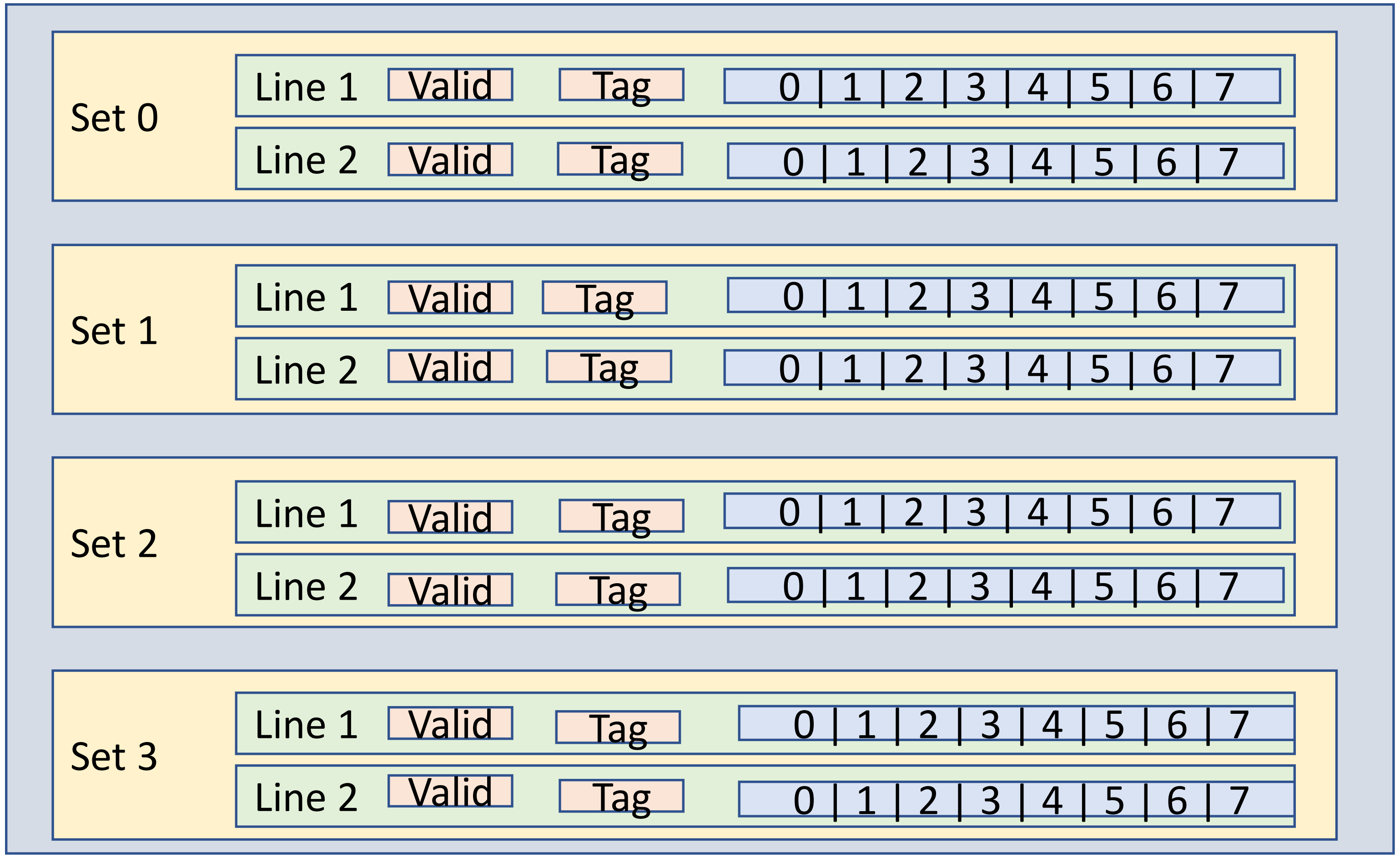
Address Partitioning

9 bit address
addr: 101110001


Set 0
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 1
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 2
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 3
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Address Partitioning

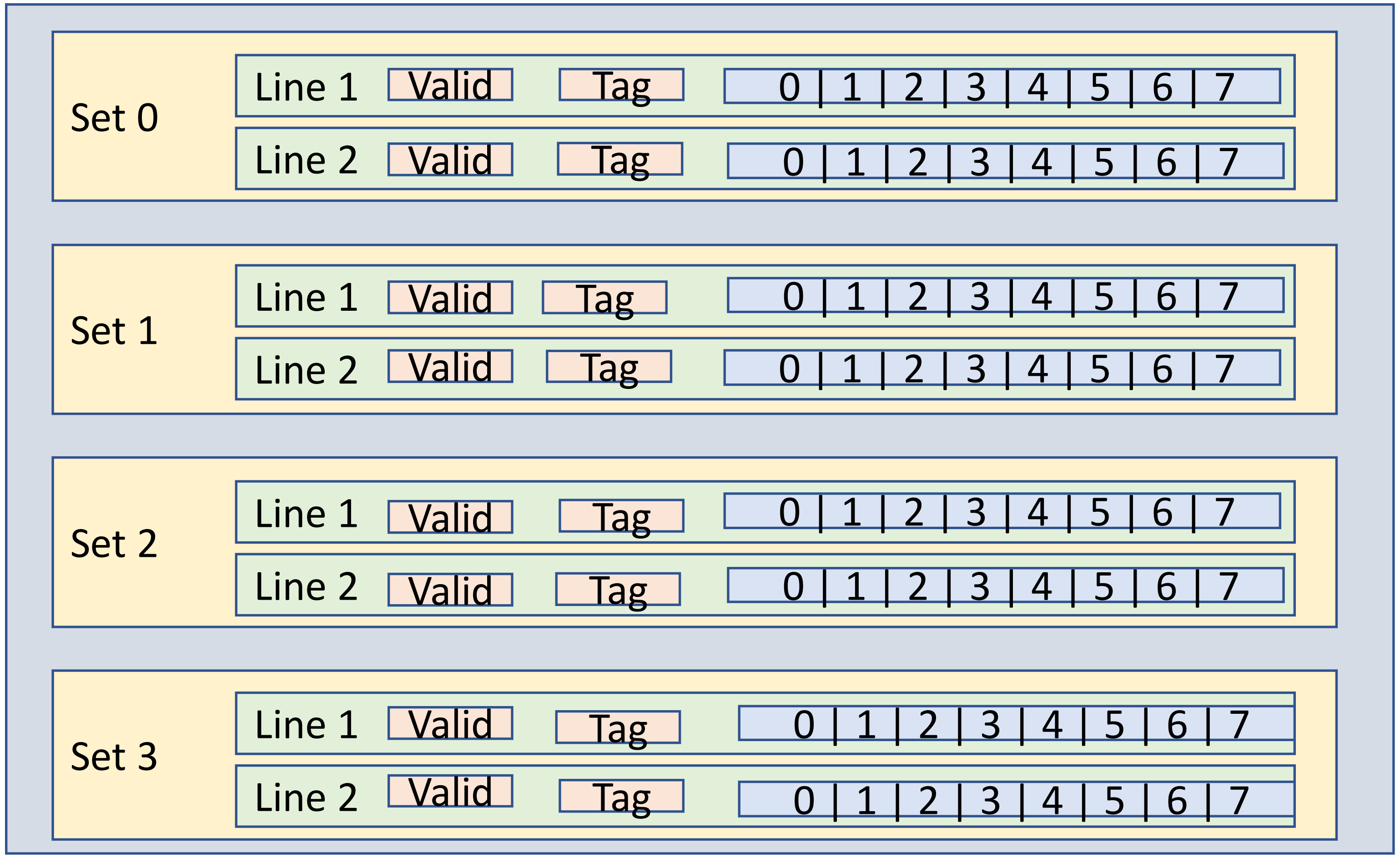
9 bit address
addr: 1011 10 001



Address Partitioning

9 bit address
addr: 1011 10 $\boxed{001}$

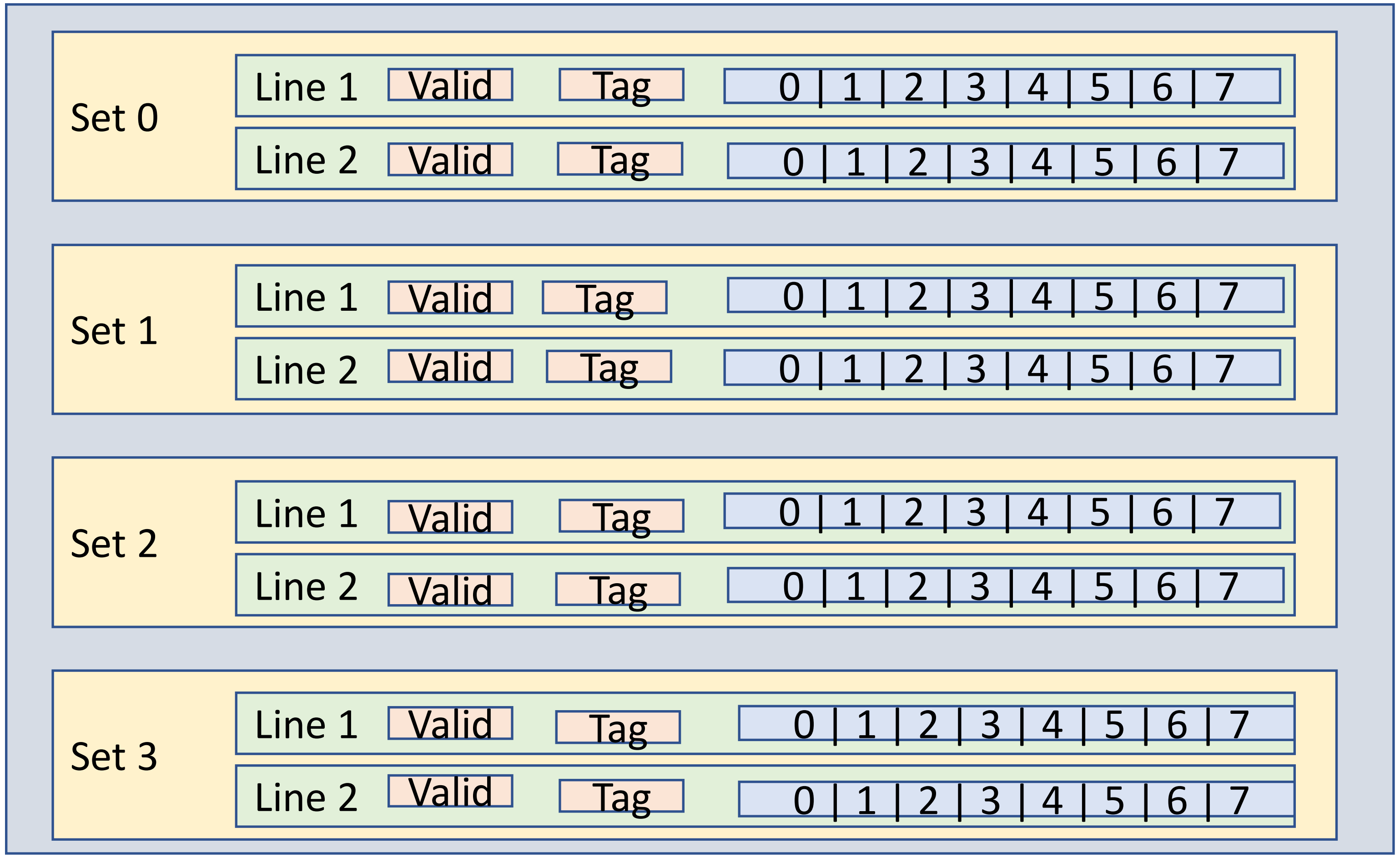
001 Byte Offset



Address Partitioning

9 bit address
addr: 1011 $\boxed{10}\boxed{001}$

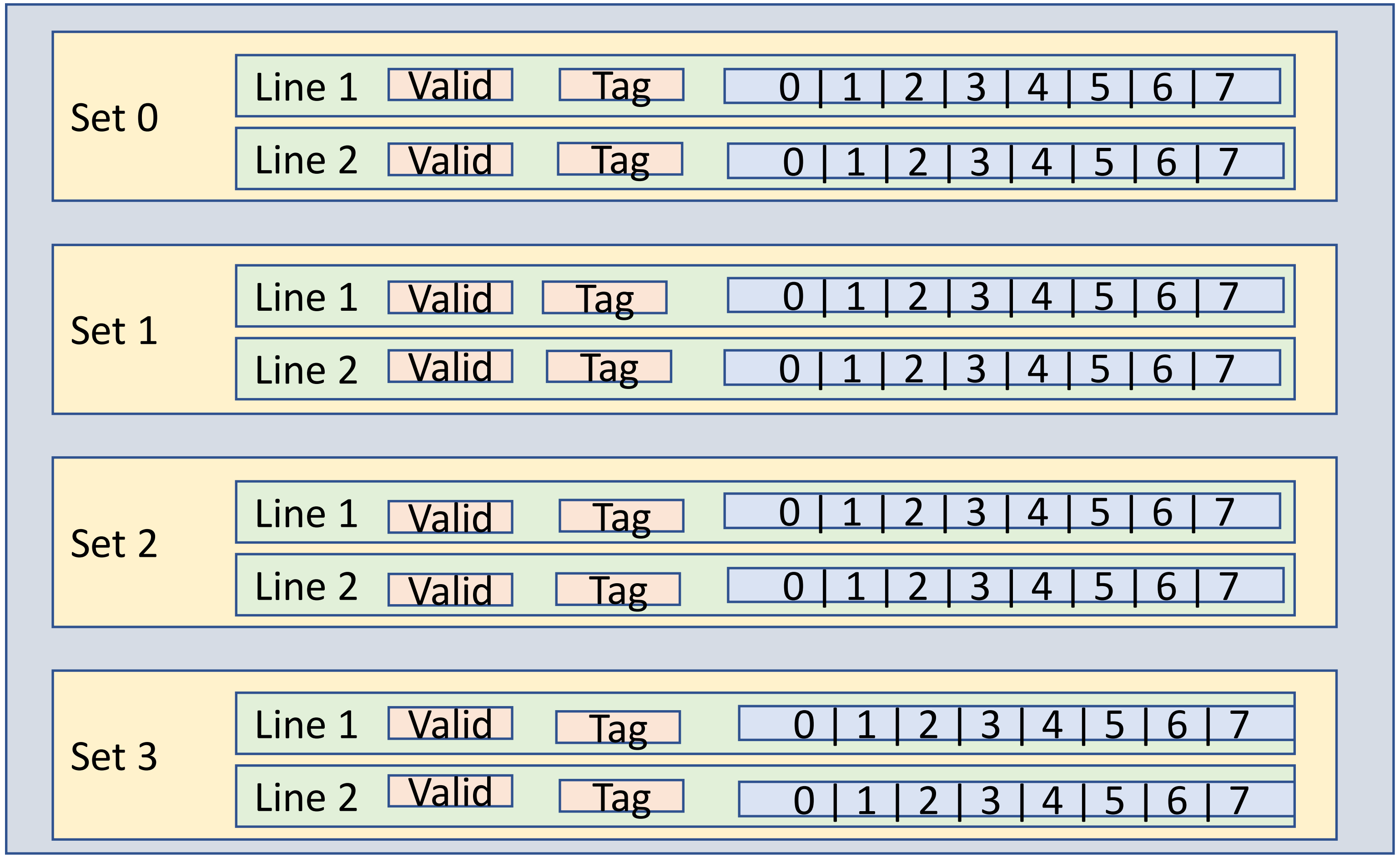
001 Byte Offset
10 Set Index


Set 0
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 1
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 2
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 3
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Address Partitioning

9 bit address
addr: 1011 10 001

001 Byte Offset
10 Set Index
1011 Tag


Set 0
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 1
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 2
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 3
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Address Partitioning

9 bit address

addr: $\boxed{1011}\boxed{10}\boxed{001}$

001 Byte Offset

10 Set Index

1011 Tag

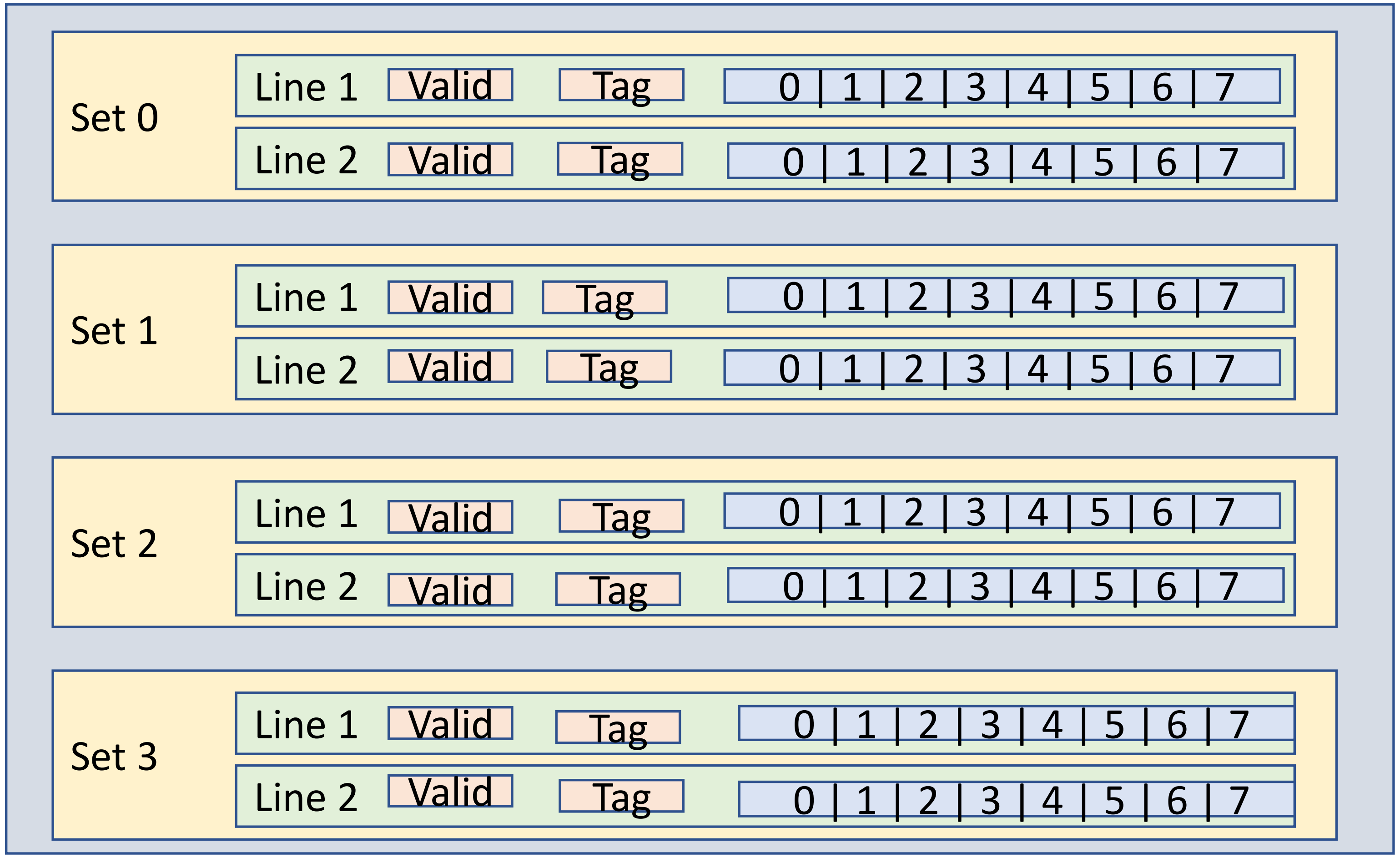
S = 4 // sets

E = 2 // lines

B = 8 // byes per line

m = 9 // number of bits in addr

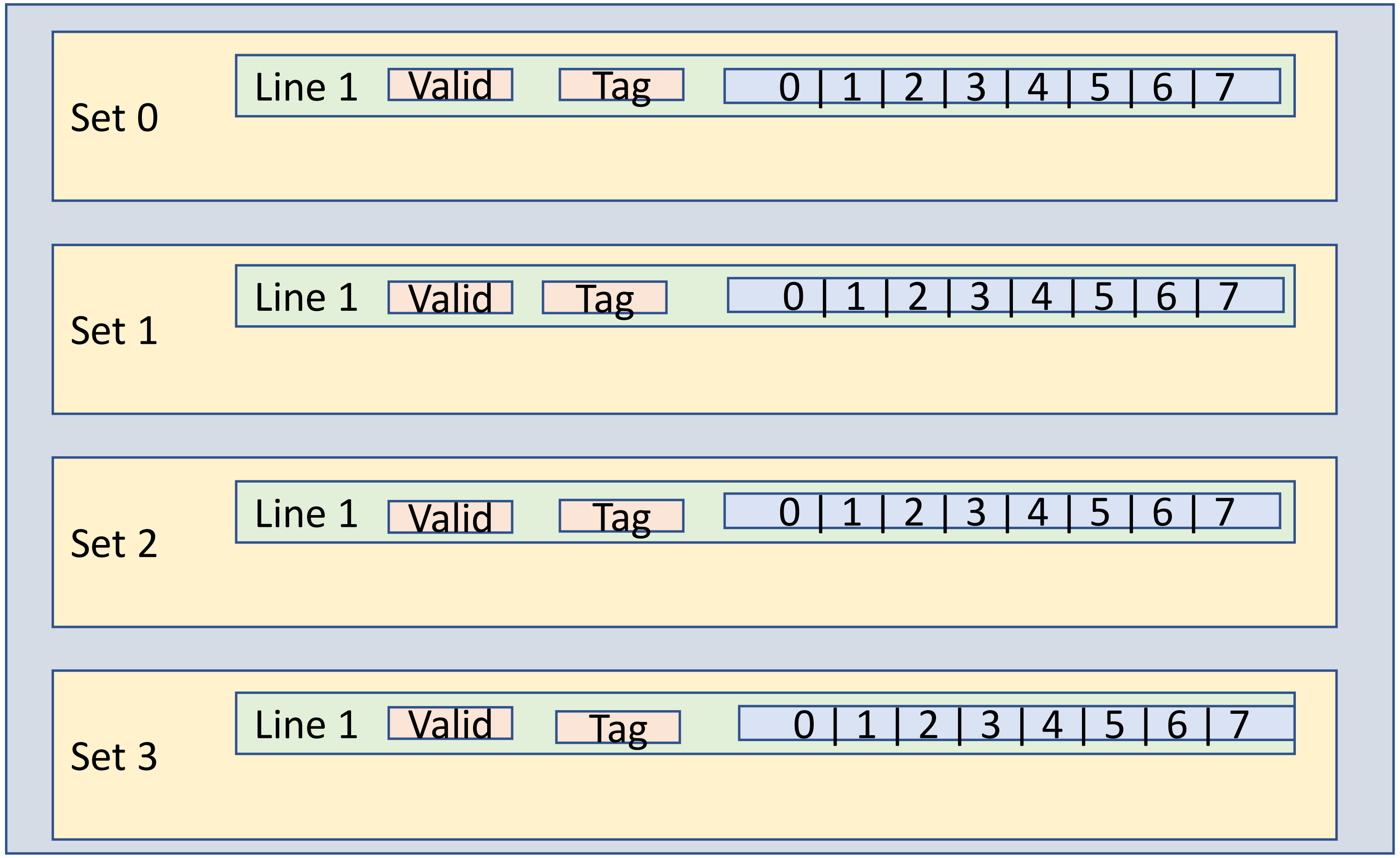
C = S*E*B = 4*2*8 = 64



Set 0
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 1
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 2
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 3
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

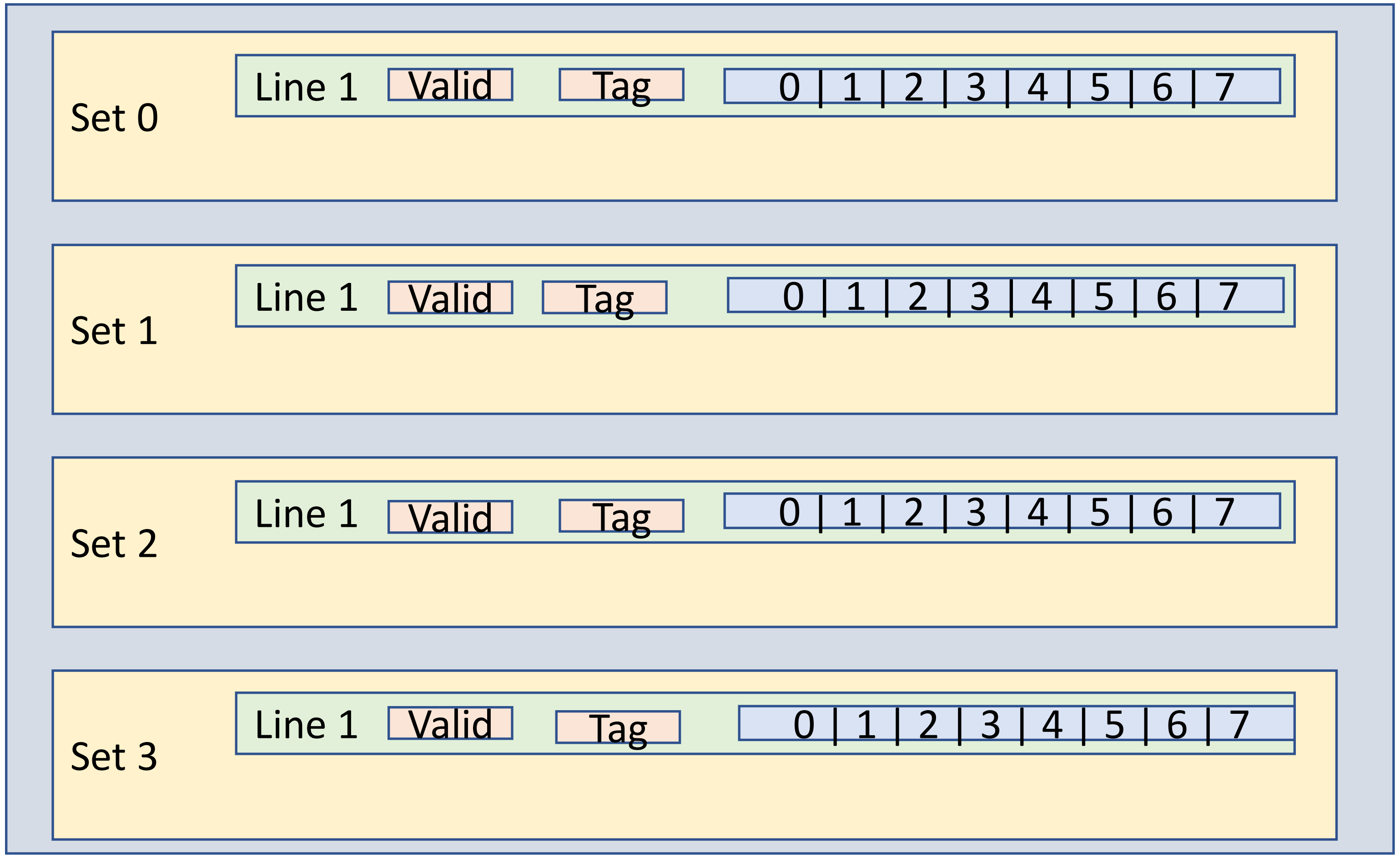
Direct Mapped Cache

Only 1 line per set



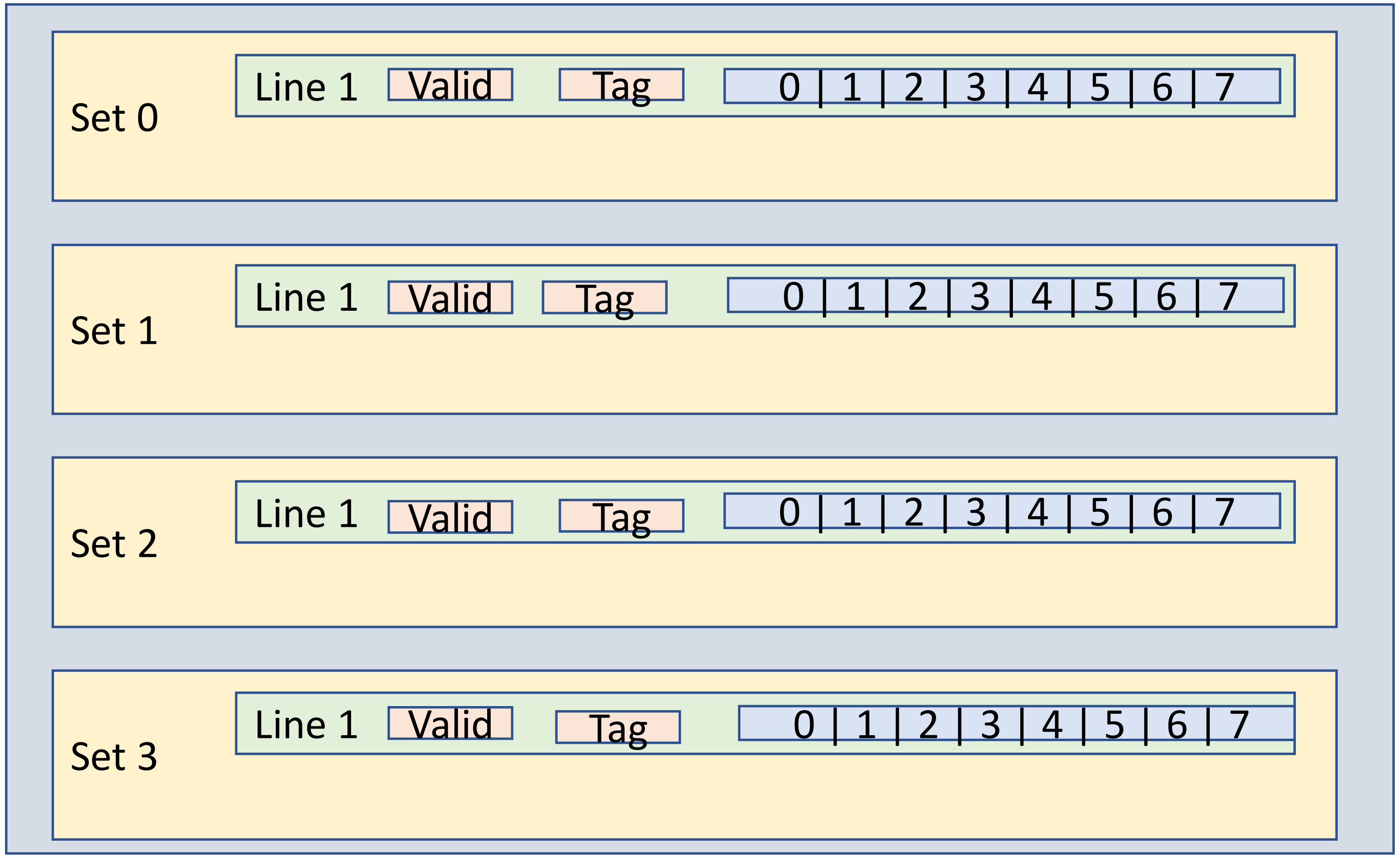
Read examples

Read
0x171

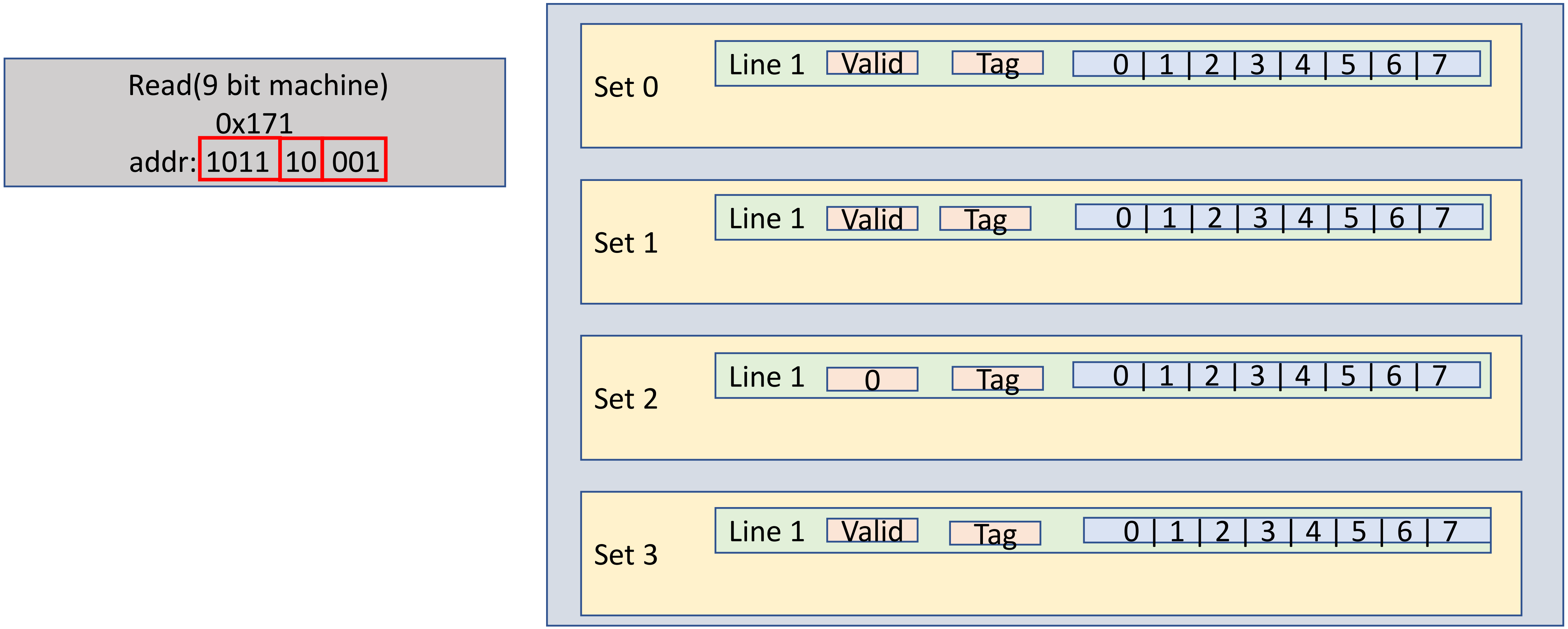


Read examples

Read (9 bit machine)
0x171
addr: 000101110001



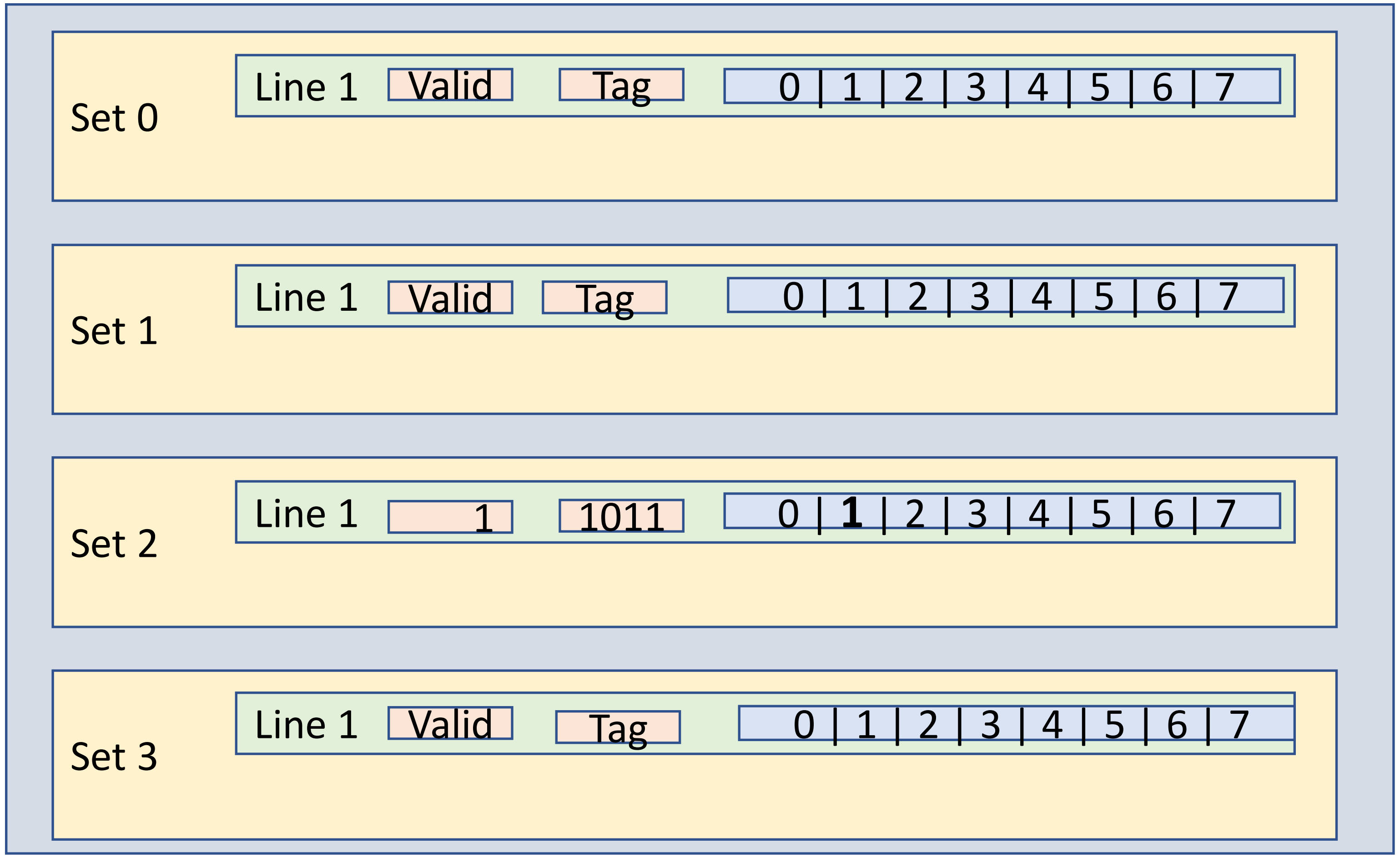
Read examples


Read(9 bit machine)
0x171
addr: 1011 10 001
Set 0
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 1
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 2
Line 1 0 Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 3
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Read examples

Read(9 bit machine)
0x171
addr: $\boxed{1011}\boxed{10}\boxed{001}$

Set: 10 = 2
Byte 001 = 1
Tag = 1011

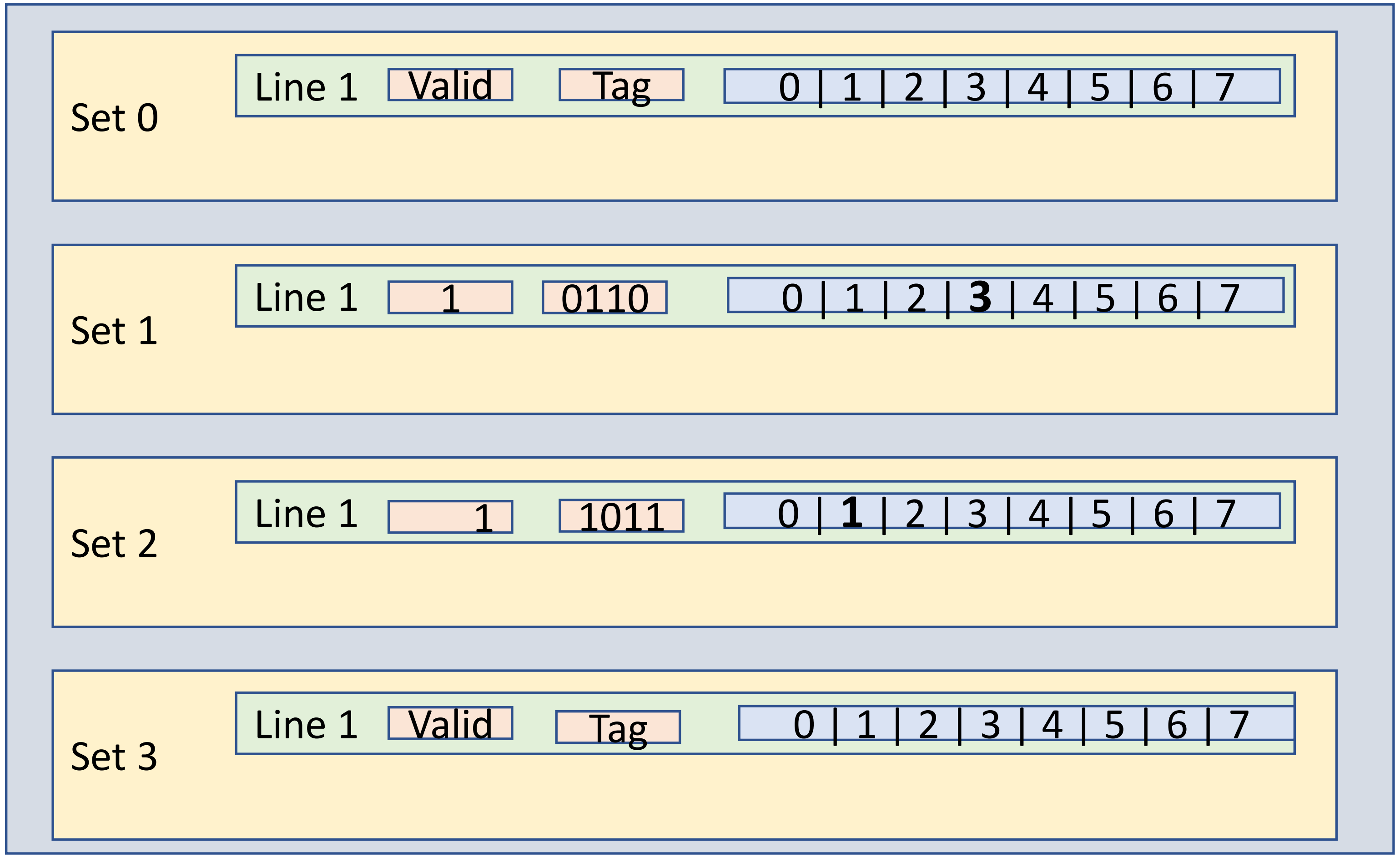

Set 0
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 1
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Set 2
Line 1 1 1011 0 | **1** | 2 | 3 | 4 | 5 | 6 | 7
Set 3
Line 1 Valid Tag 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

Read examples

Read(9 bit machine)

addr: $\boxed{0110}\boxed{01}\boxed{011}$

Set: 01 = 1
Byte 011 = 3
Tag = 0110



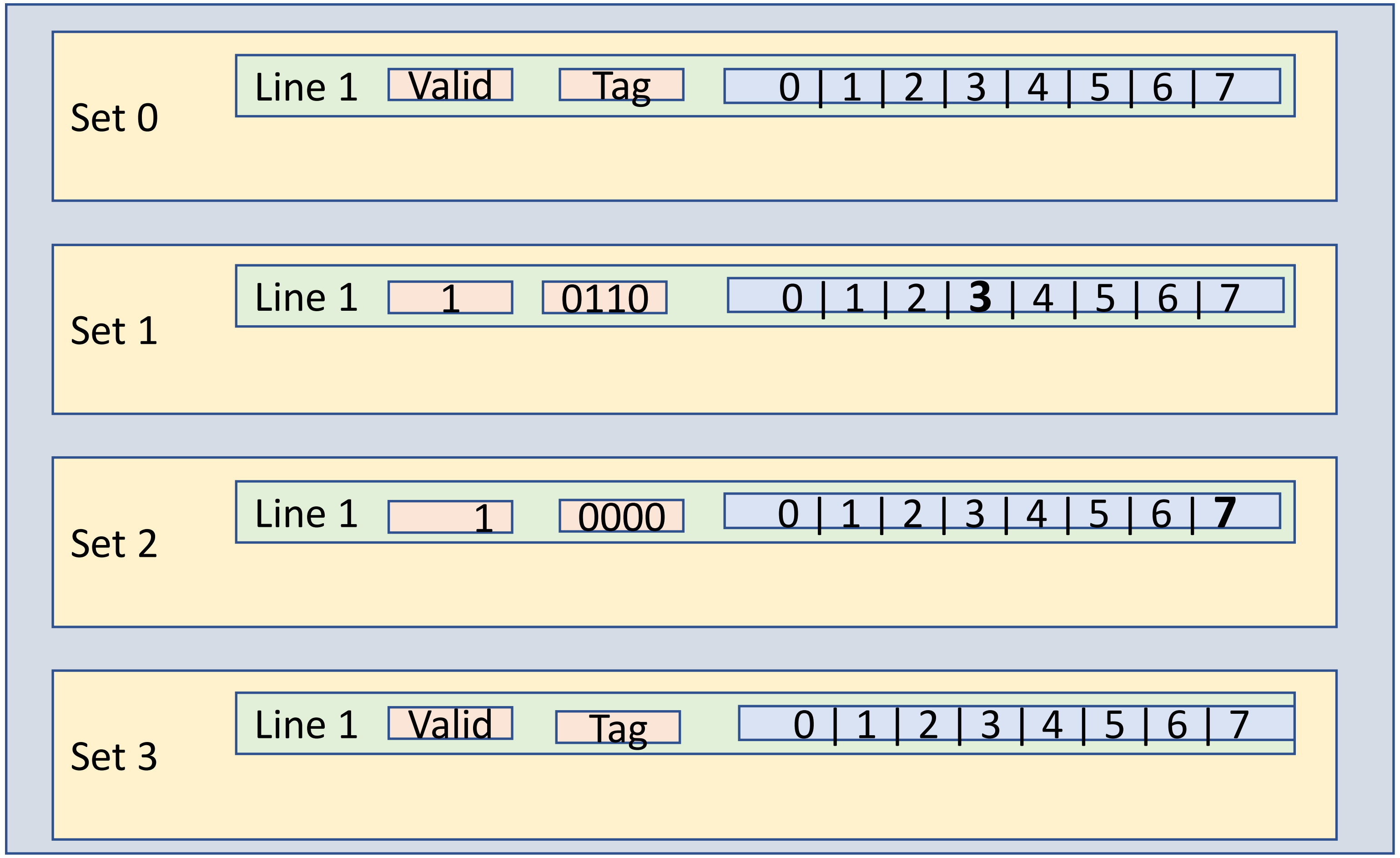
Read examples

Read(9 bit machine)

addr: $\boxed{0000}\boxed{10}\boxed{111}$

Set: 10 = 2
Byte 001 = 7
Tag = 0000

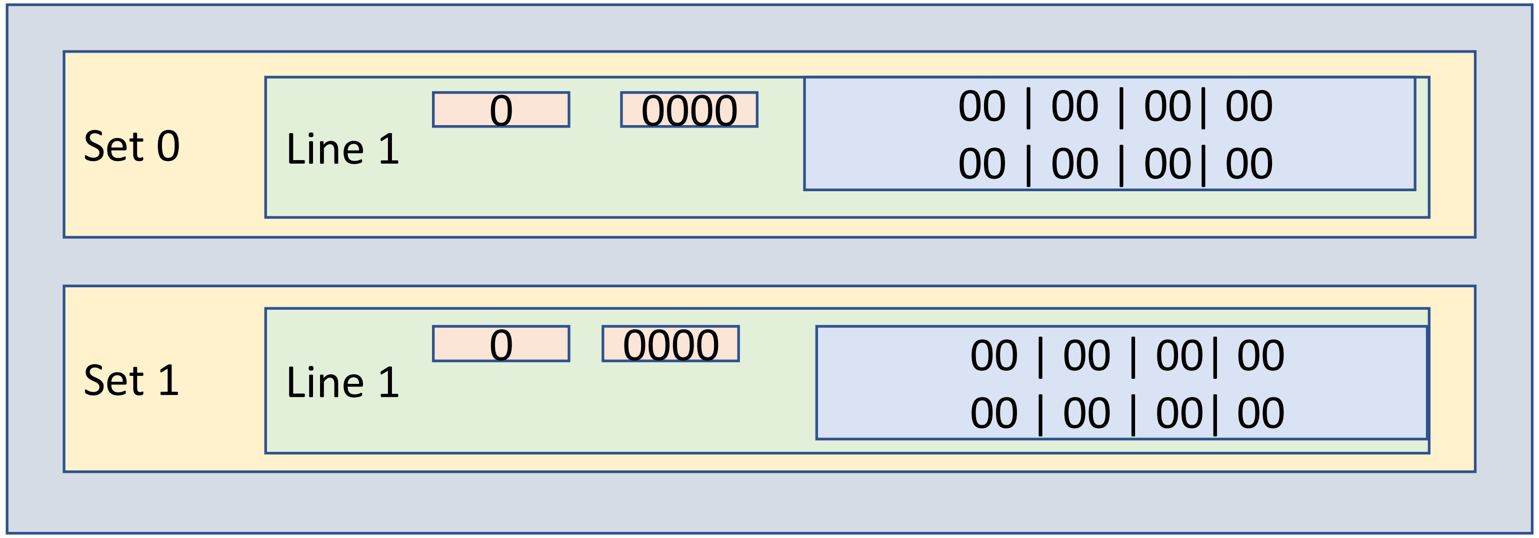
Cache Miss!!!
valid, but tag is wrong
Evict and replace



Thrashing

```
int x[4] = {1,2,3,4};
int y[4] = {4,3,2,1};
for (i=0; i<4; i++)
    sum+=x[i]*y[i];
```

addr	Val
x0: 0x00	1
x1: 0x04	2
x2: 0x08	3
x3: 0x0C	4
y0: 0x10	4
y1: 0x14	3
y2: 0x18	2
y3: 0x1C	1



```
addr: 0000 0000
```

```
Hits:   0
Misses: 8
```

Set Associative Caches
E>1 and S>1

```
int x[4] = {1,2,3,4};
int y[4] = {4,3,2,1};
for (i=0; i<4; i++)
    sum+=x[i]*y[i];
```

Set 0	Line 1	1	0000	1 | 2
	Line 2	1	0001	4 | 3
Set 1	Line 1	1	0000	3 | 4
	Line 2	1	0001	2 | 1

addr	Val
x0: 0x00	1
x1: 0x04	2
x2: 0x08	3
x3: 0x0C	4
y0: 0x10	4
y1: 0x14	3
y2: 0x18	2
y3: 0x1C	1

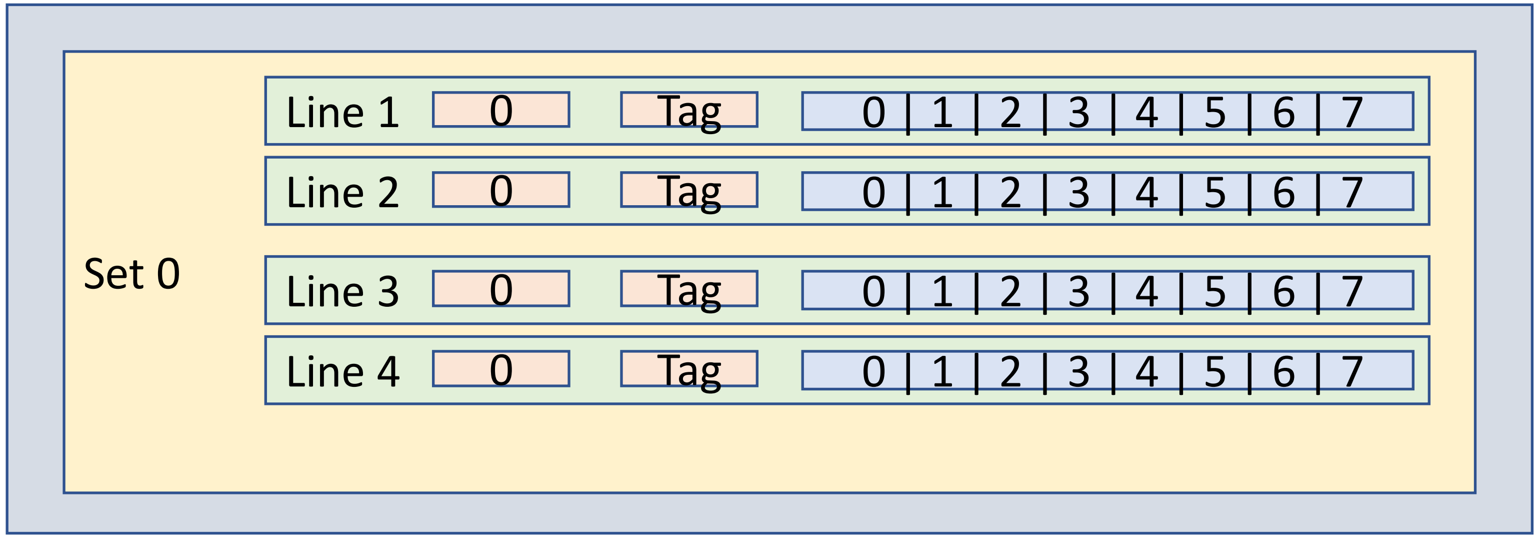
```
addr: 0001  1  100
```

```
Hits:    4
Misses:  4
```

Fully Associative Caches

S = 1

```
int x[4] = {1,2,3,4};
int y[4] = {4,3,2,1};
for (i=0; i<4; i++)
    sum+=x[i]*y[i];
```



Set 0
Line 1 | 0 | Tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 2 | 0 | Tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 3 | 0 | Tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
Line 4 | 0 | Tag | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

addr	Val
x0: 0x00	1
x1: 0x04	2
x2: 0x08	3
x3: 0x0C	4
y0: 0x10	4
y1: 0x14	3
y2: 0x18	2
y3: 0x1C	1

```
addr:
```

```
Hits:   4
Misses: 4
```

Writes

If in cache at level K

1. Write Through
    1. if we write something to memory it also writes to the level below it
    2. L1 -> L2
2. Write Back
    1. only writes to level K, but marks valid block as dirty
    2. main memory -> Hard Drive

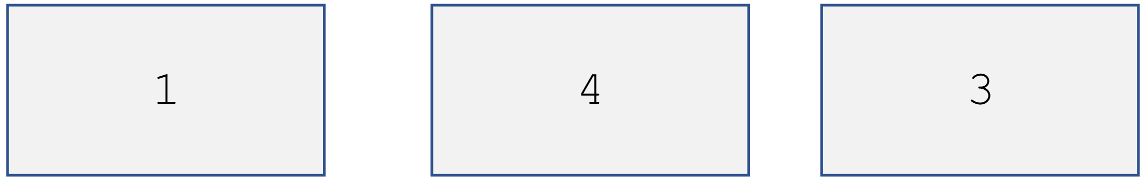
If not in cache at level K

1. Write-Allocate
    1. move the block up to level K from K+1
    2. and write as either write through or write back depending on policy
2. No-Write-Allocate
    1. just directly write to level K+1
    2. don’t move the block up to level K

Eviction Policy

1. Random
2. Least Recently Used (LRU)
3. Least Frequently Used (LFU)

LRU with access pattern: 0, 1, 2, 1, 0, 3, 4, 1



LFU with access pattern: 0 0 0 1 2 1 3 3 3 3 4