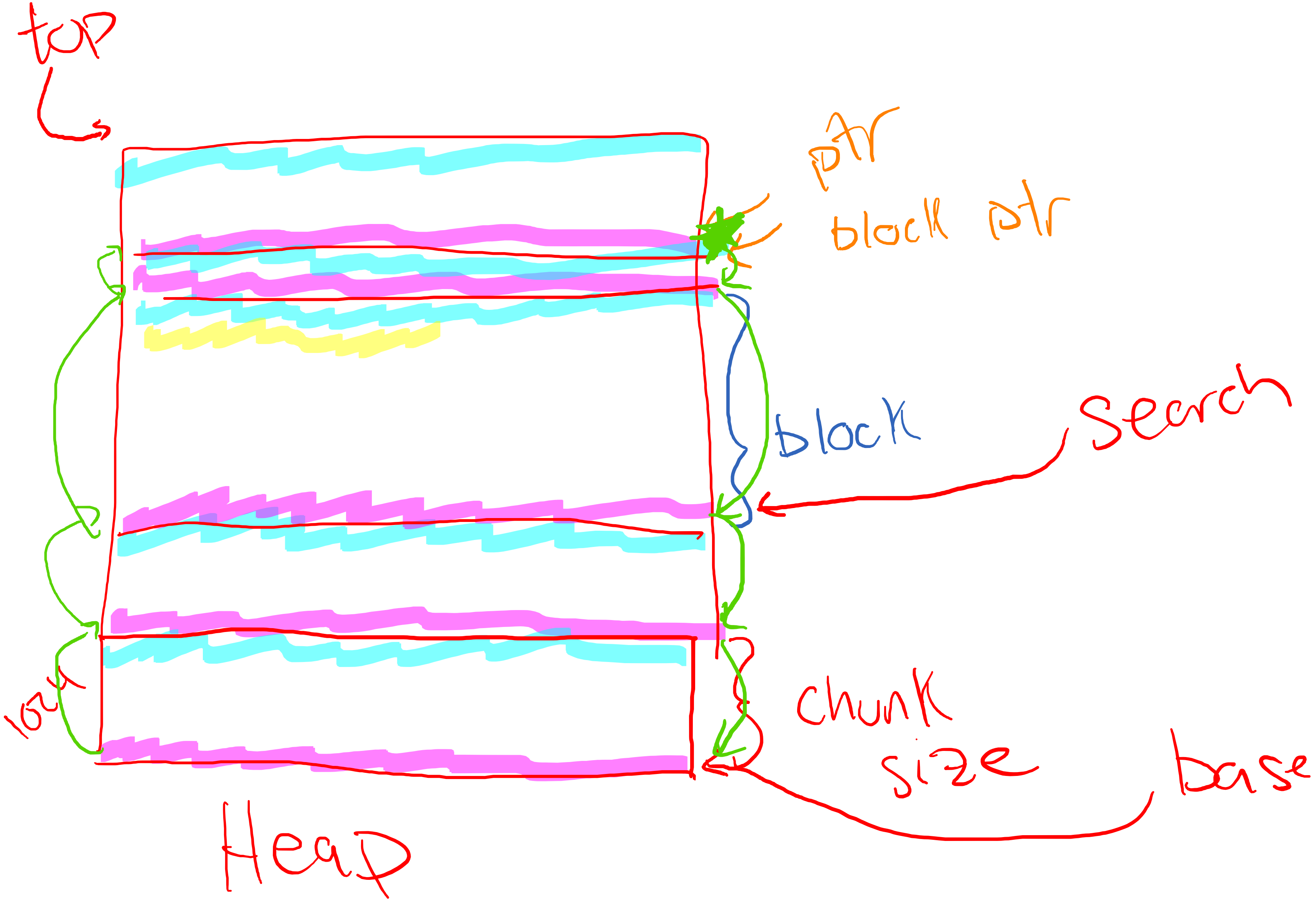
Cache

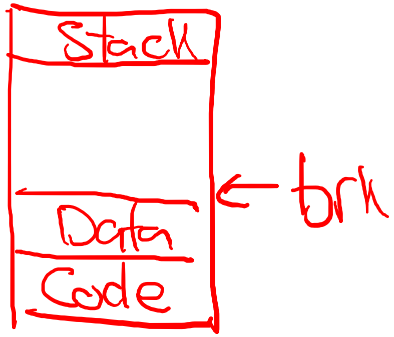
Introduction and Direct Mapped

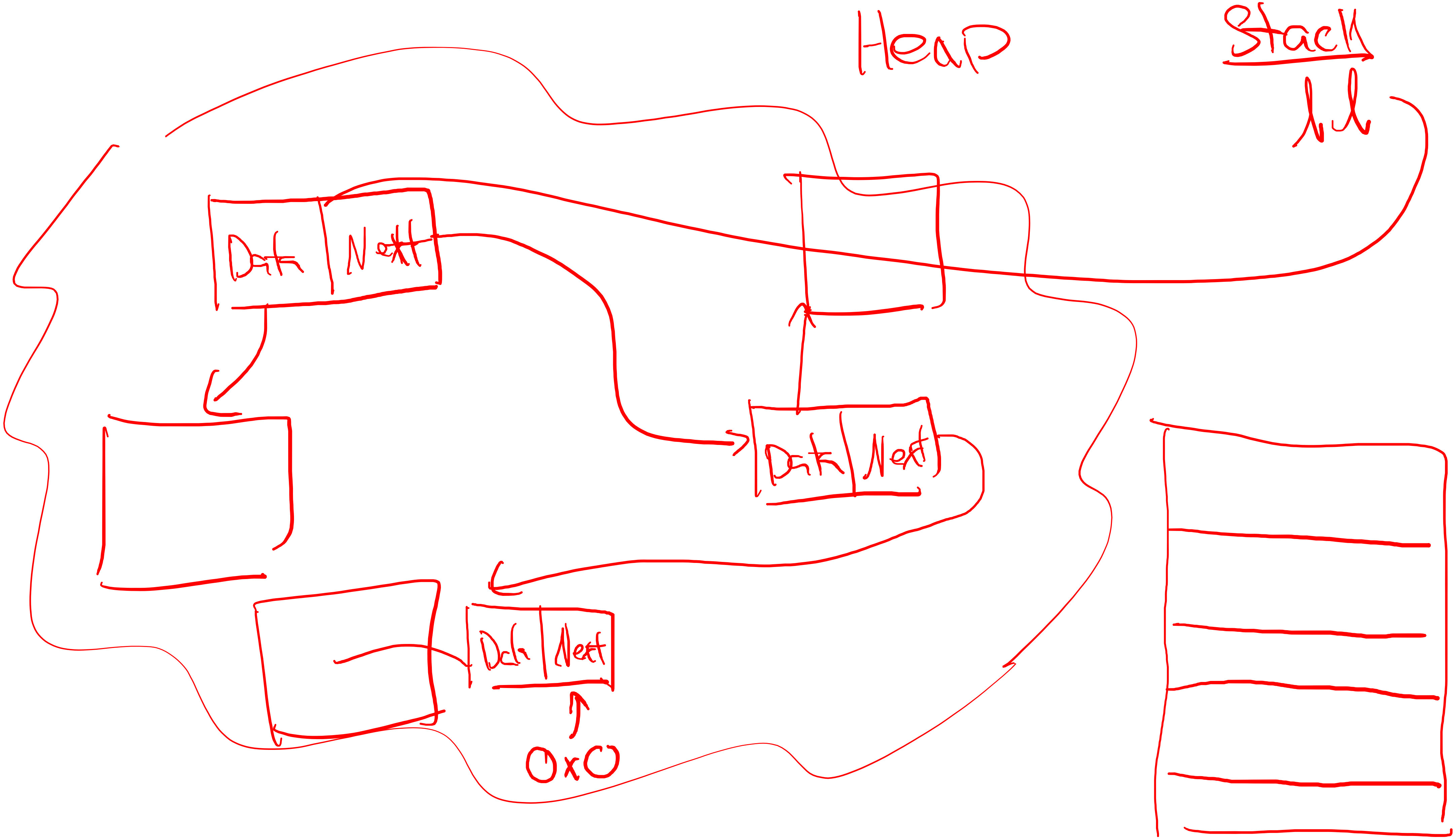
Drawing: Dynamic Memory

- Take three minutes to draw “heap” memory

- Some reminders
  - Implicit lists
  - Headers and footers
  - User (payload) pointers
  - Blocks and block pointers
  - Alignment

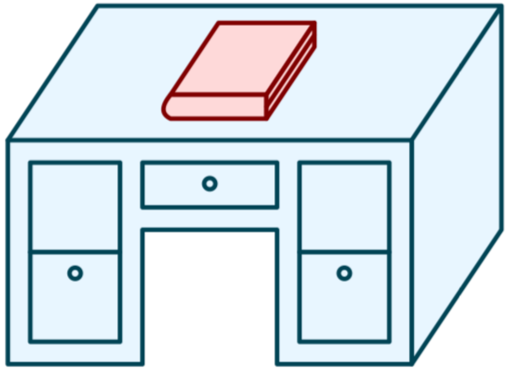

top
ptr
block ptr
block
search
1024
chunk size
base
Heap


Stack
brk
Data
Code

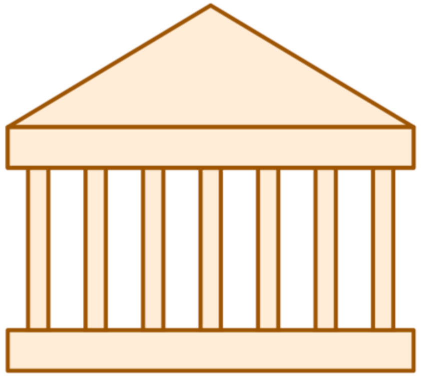

Heap
Stack
l.l
Data
Next
Data
Next
Data
Next
0x0

The “Book” Cache Analogy

- You’ve decided to learn more about computer systems than is covered in this course.
- The library contains all the books you want, but you prefer to study at home.
- You have the following constraints:



Desk
(can hold one book)



Library
(can hold many books)

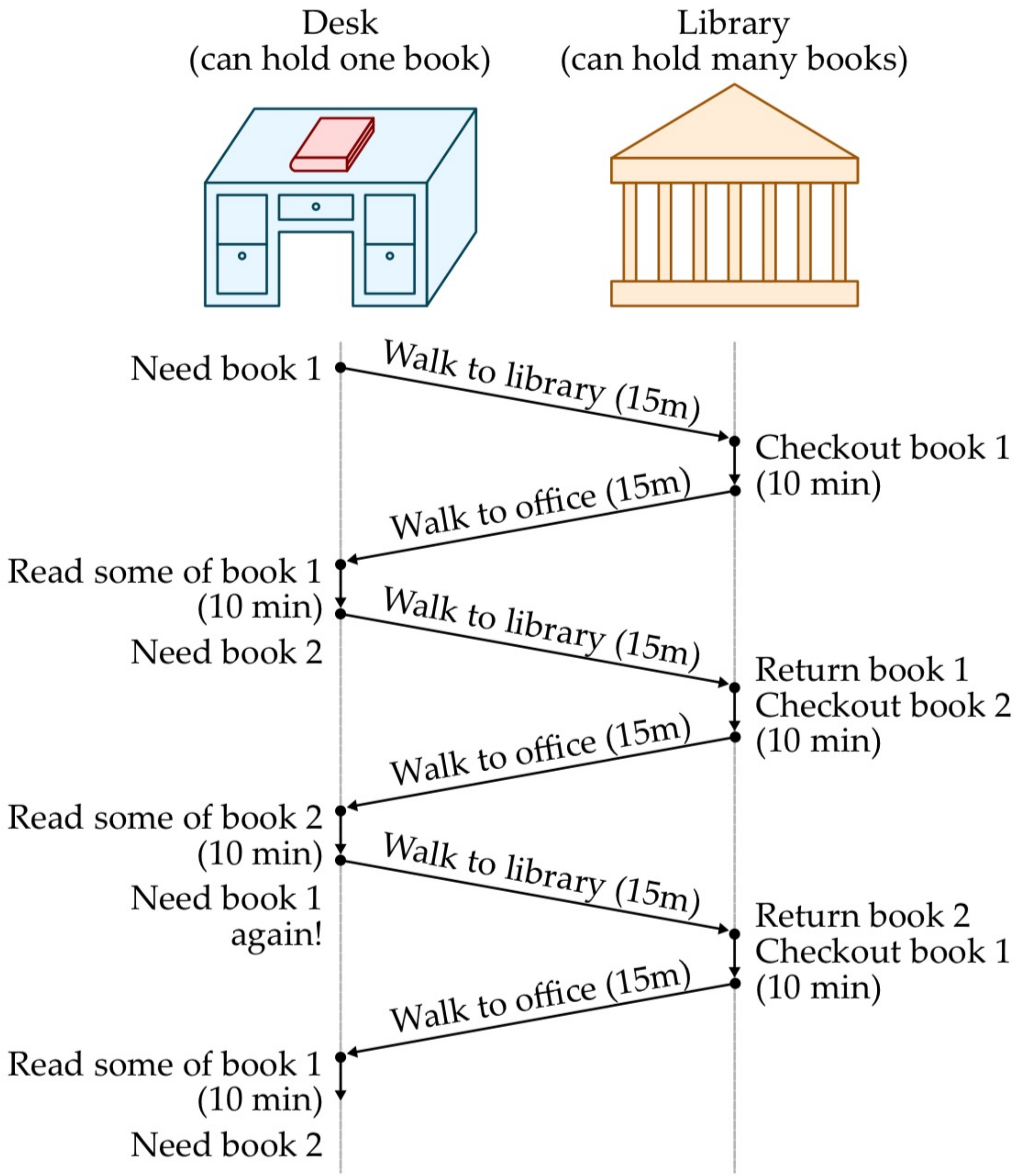
Life without Cache

Latency: average time to access a book

$$L = 15m + 10m + 15m = 40m$$

Throughput: books read per time period

$$T = \frac{1b}{50m} = 1.2\frac{b}{hr}$$



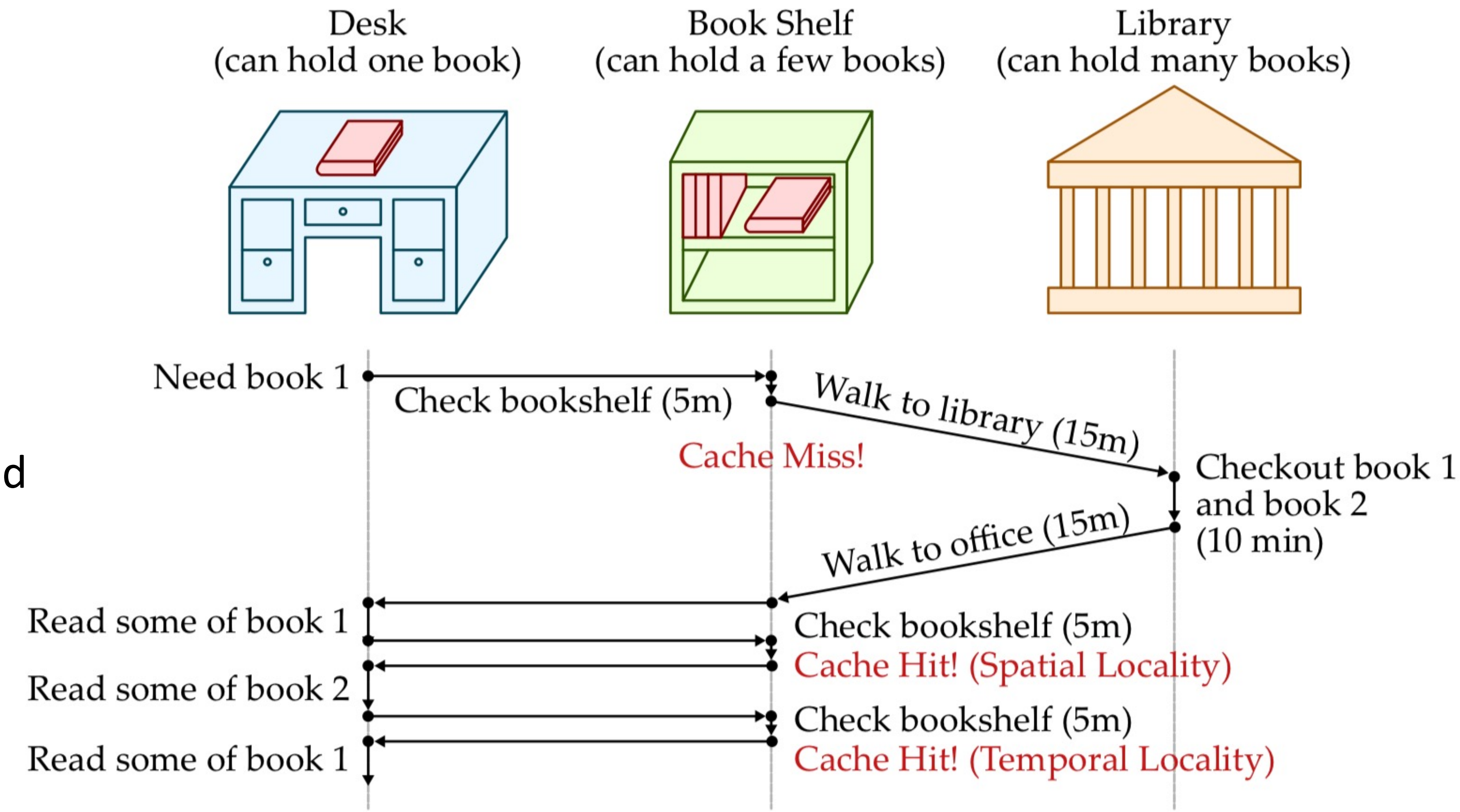
Life with Cache

Latency: average time to access a book

$$L = \frac{45 + 5 + 5}{3} \approx 18m$$

Throughput: books read per time period

$$T = \frac{3b}{85m} \approx 2\,\frac{b}{hr}$$



Caching Vocabulary

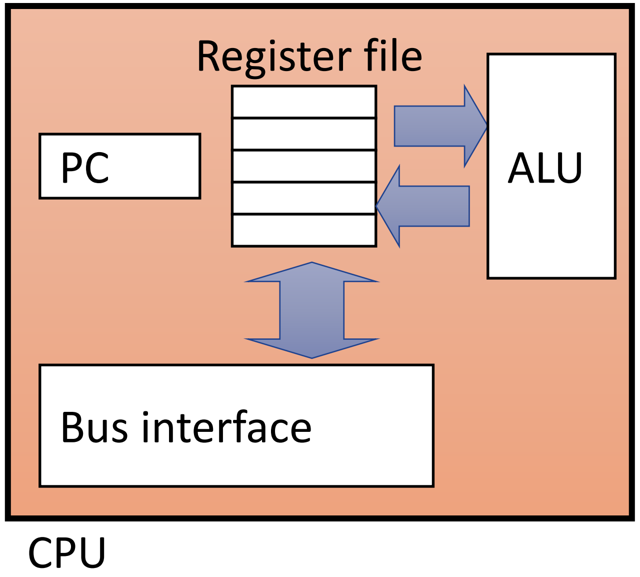
- Size: the total number of bytes that can be stored in the cache

- Cache Hit: the desired value is in the cache and quickly returned
- Hite rate: the fraction of accesses that are hits
- Hit time: the time to process a hit

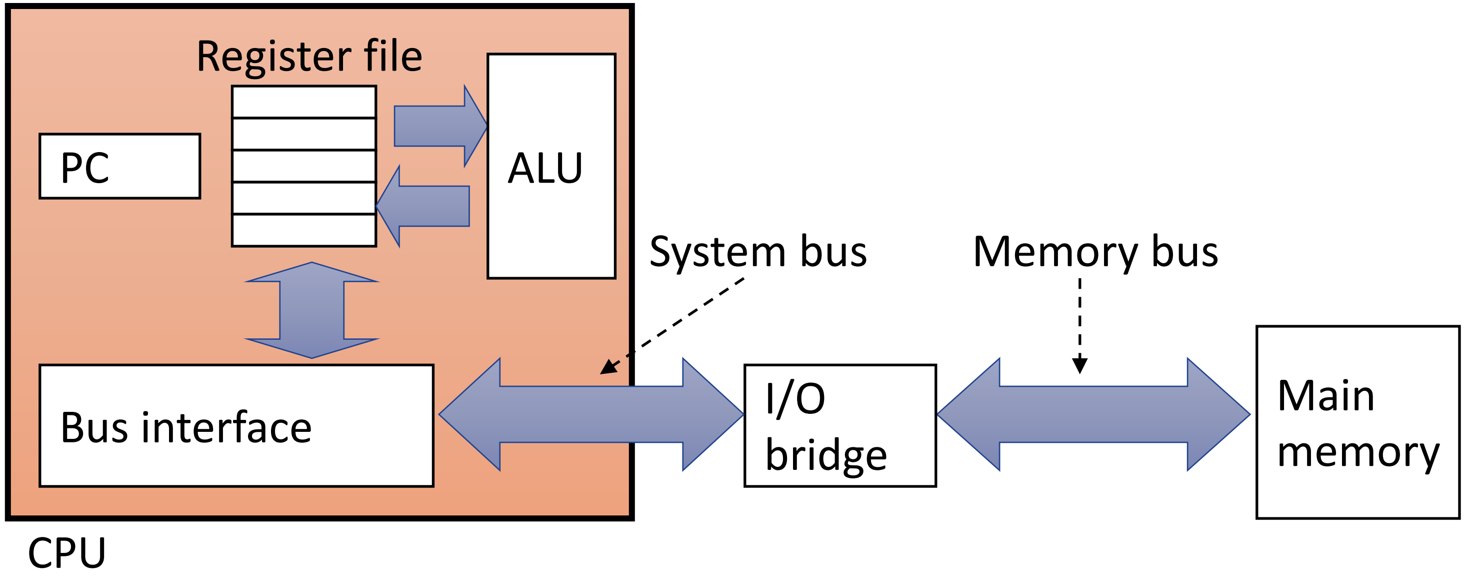
- Cache Miss: the desired value is not in the cache and must be fetched elsewhere
- Miss rate: the fraction of accesses that are misses
- Miss penalty: the additional time to process a miss

- Average access time: hit-time + miss-rate * miss-penalty

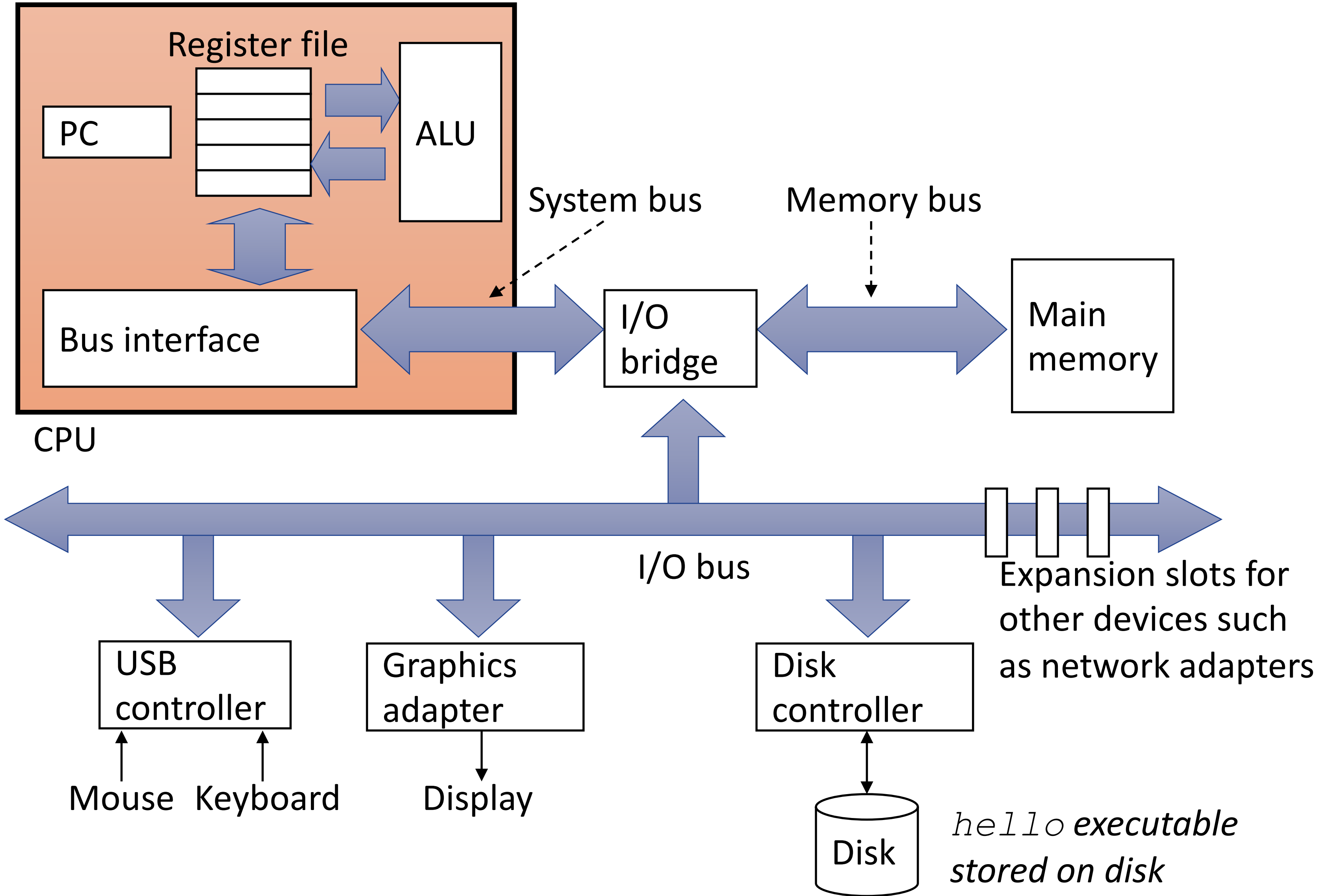
A Computer System



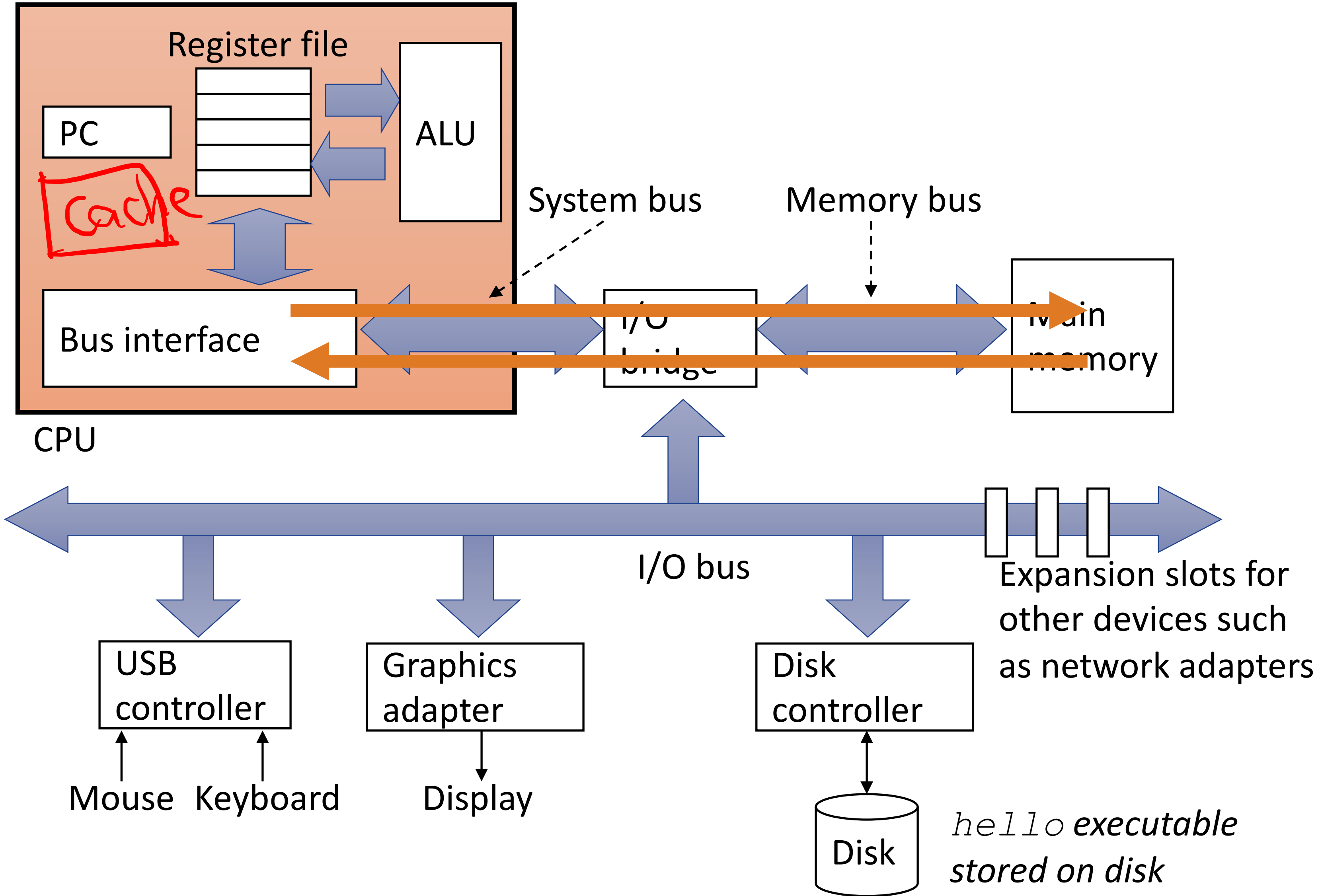
A Computer System



A Computer System

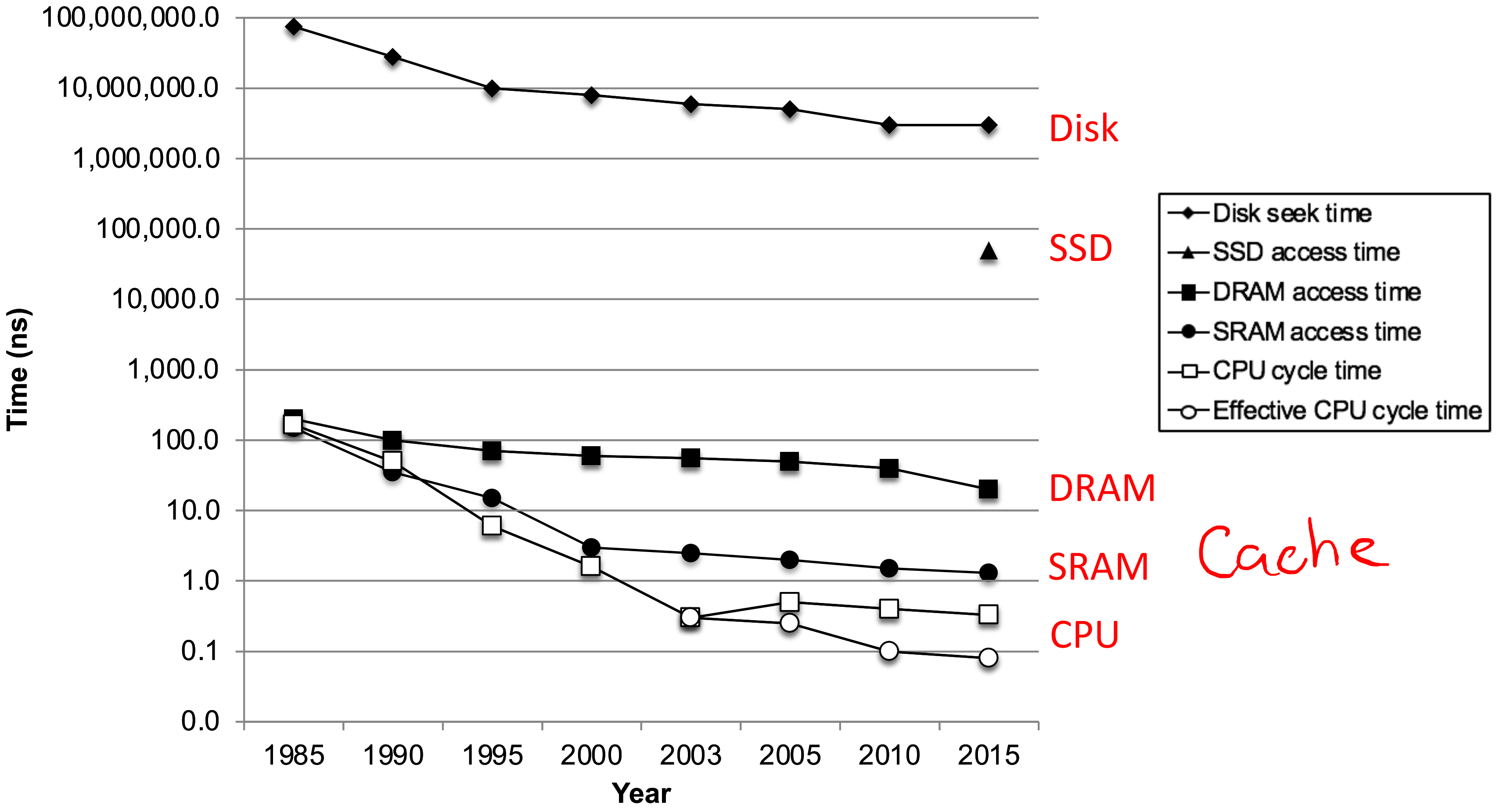


A Computer System



`hello` executable stored on disk

The CPU-Memory Gap



Disk

SSD

DRAM

SRAM

CPU

Cache

Caching

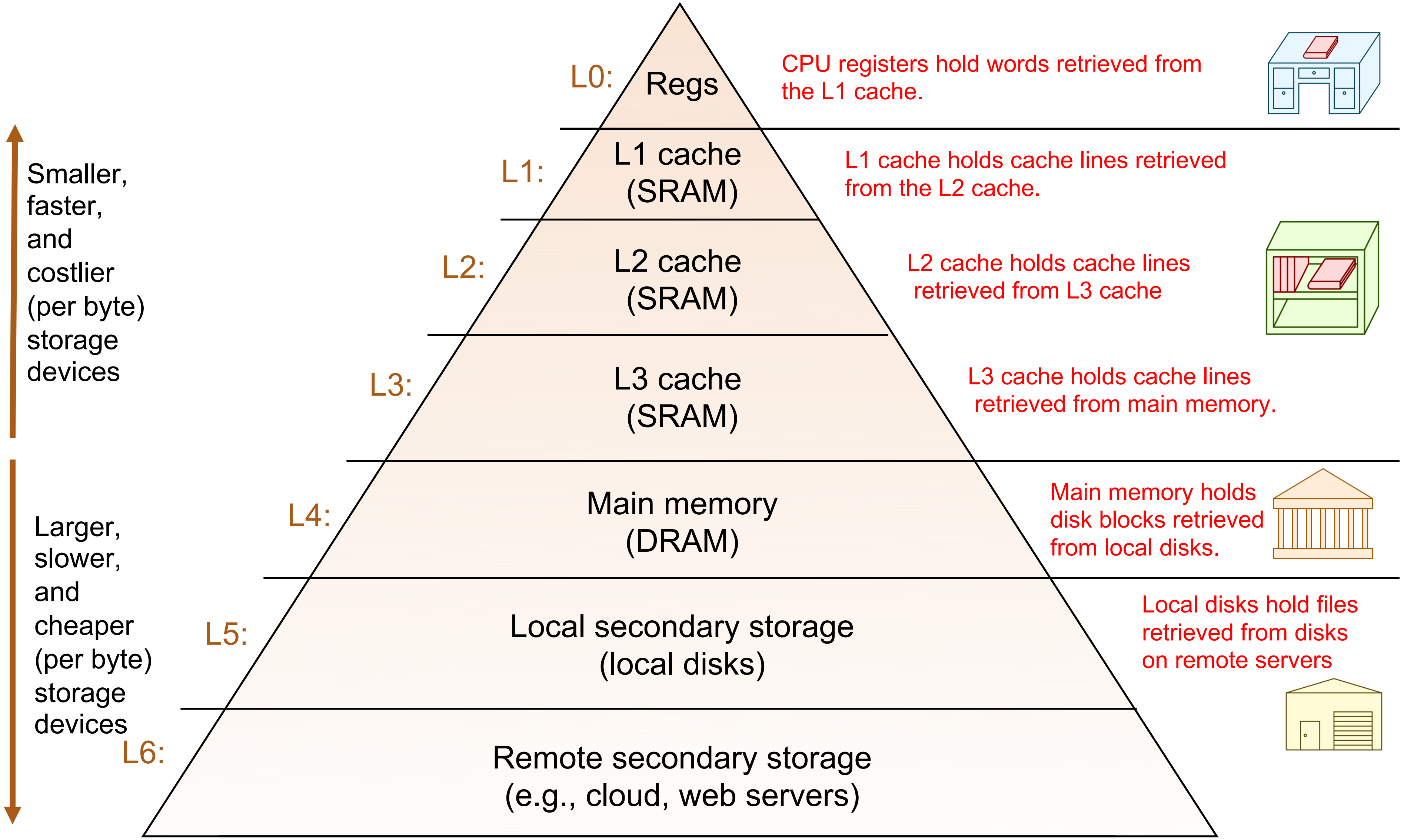
- Keep some memory values nearby in fast memory

L1 L2 L3 L4

- Modern systems have 3 or even 4 levels of caches

- Cache idea is widely used:
    - Disk controllers
    - Webpage loading
    - (Virtual memory: main memory is a “cache” for the disk)

Memory Hierarchy



Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices
L0:
Regs
L1:
L1 cache (SRAM)
L2:
L2 cache (SRAM)
L3:
L3 cache (SRAM)
L4:
Main memory (DRAM)
L5:
Local secondary storage (local disks)
L6:
Remote secondary storage (e.g., cloud, web servers)
CPU registers hold words retrieved from the L1 cache.
L1 cache holds cache lines retrieved from the L2 cache.
L2 cache holds cache lines retrieved from L3 cache
L3 cache holds cache lines retrieved from main memory.
Main memory holds disk blocks retrieved from local disks.
Local disks hold files retrieved from disks on remote servers

Latency numbers every programmer should know (2020)

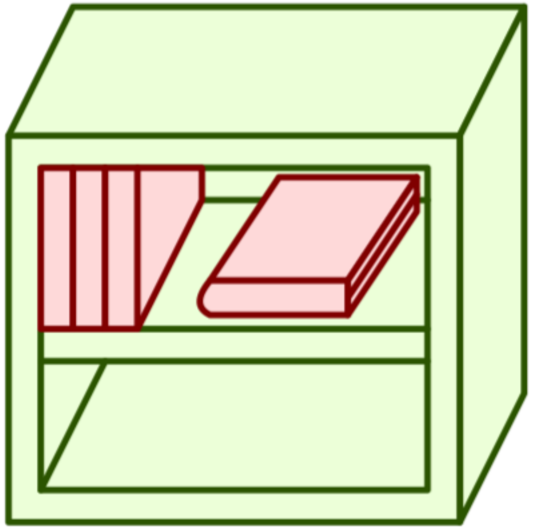
L1 cache reference	1 ns	
Branch mispredict	3 ns	
L2 cache reference	4 ns	
Main memory reference	100 ns	
Memory 1MB sequential read	3,000 ns	3 µs
SSD random read	16,000 ns	16 µs
SSD 1MB sequential read	49,000 ns	49 µs
Magnetic Disk seek	2,000,000 ns	2 ms
Magnetic Disk 1MB sequential read	825,000 ns	825 µs
Round trip in Datacenter	500,000 ns	500 µs
Round trip CA<->Europe	150,000,000 ns	150 ms

Caching Strategies

How should we decide which books to keep in the bookshelf?

Alternatively

How should we decide which books to evict from the bookshelf?



Example Access Patterns

constant addr

seq. of addr

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

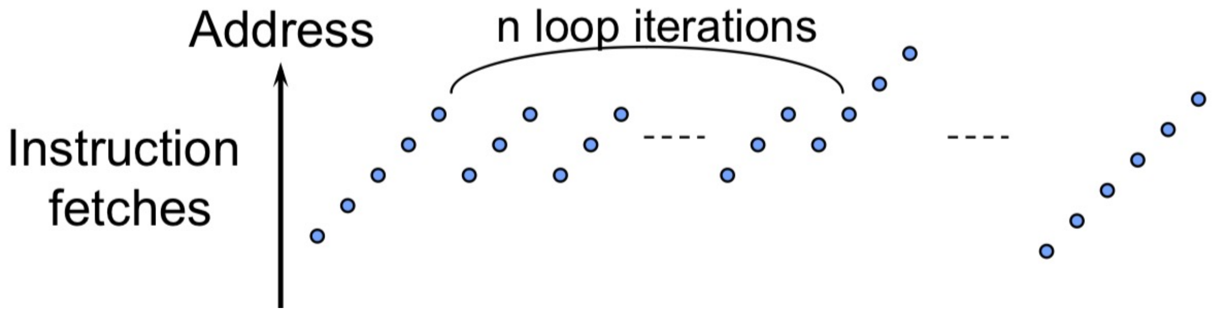
Data references

- Reference array elements in succession.
- Reference variable `sum` each iteration.

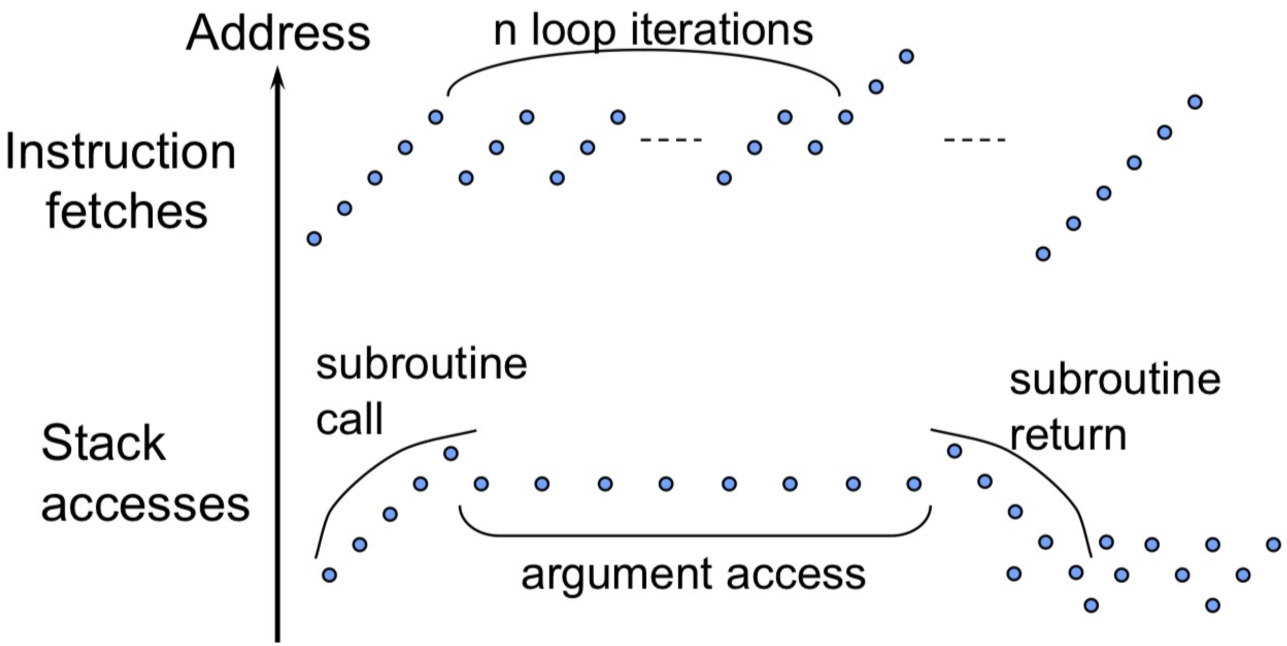
Instruction references

- Reference instructions in sequence.
- Cycle through loop repeatedly.

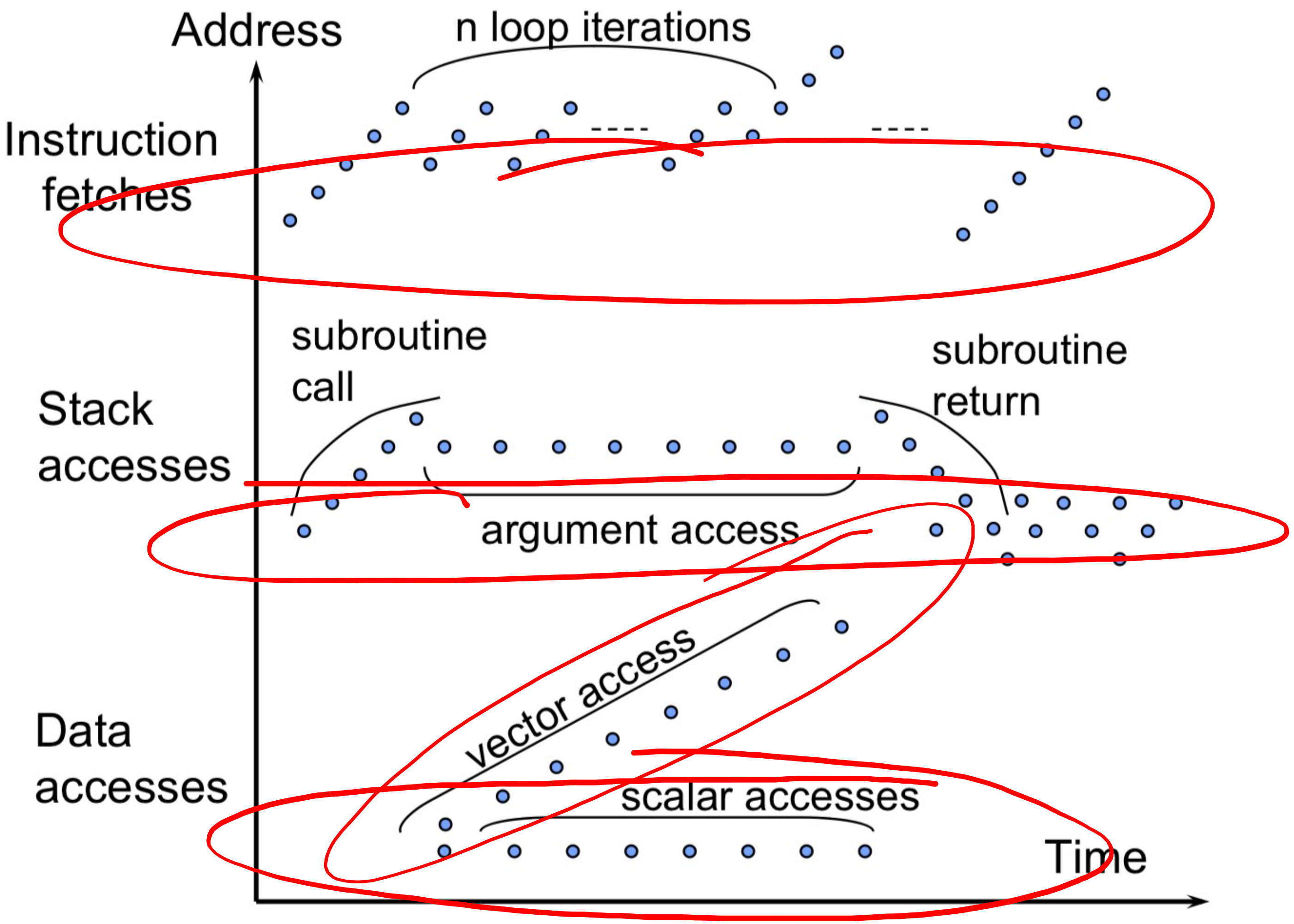
Example Access Patterns



Example Access Patterns



Example Access Patterns

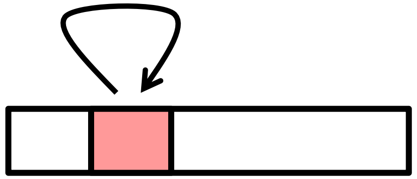


Principle of Locality

Programs tend to use data and instructions with addresses near or equal to those they have used recently

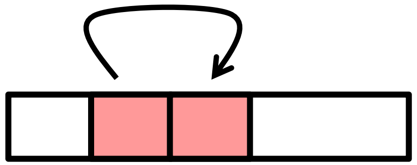
Temporal locality:

- Recently referenced items are likely to be referenced again soon

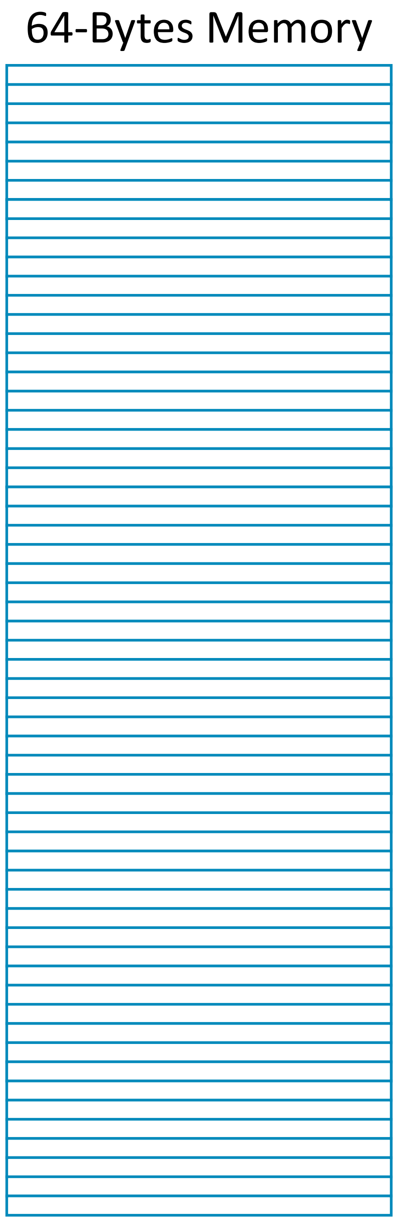


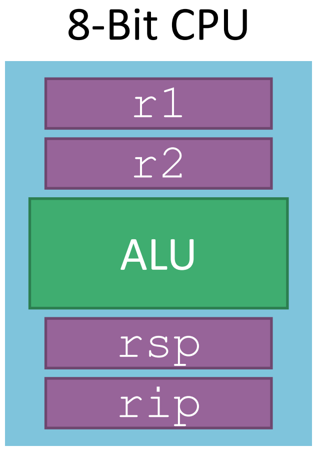
Spatial locality:

- Items with nearby addresses tend to be referenced close together in time



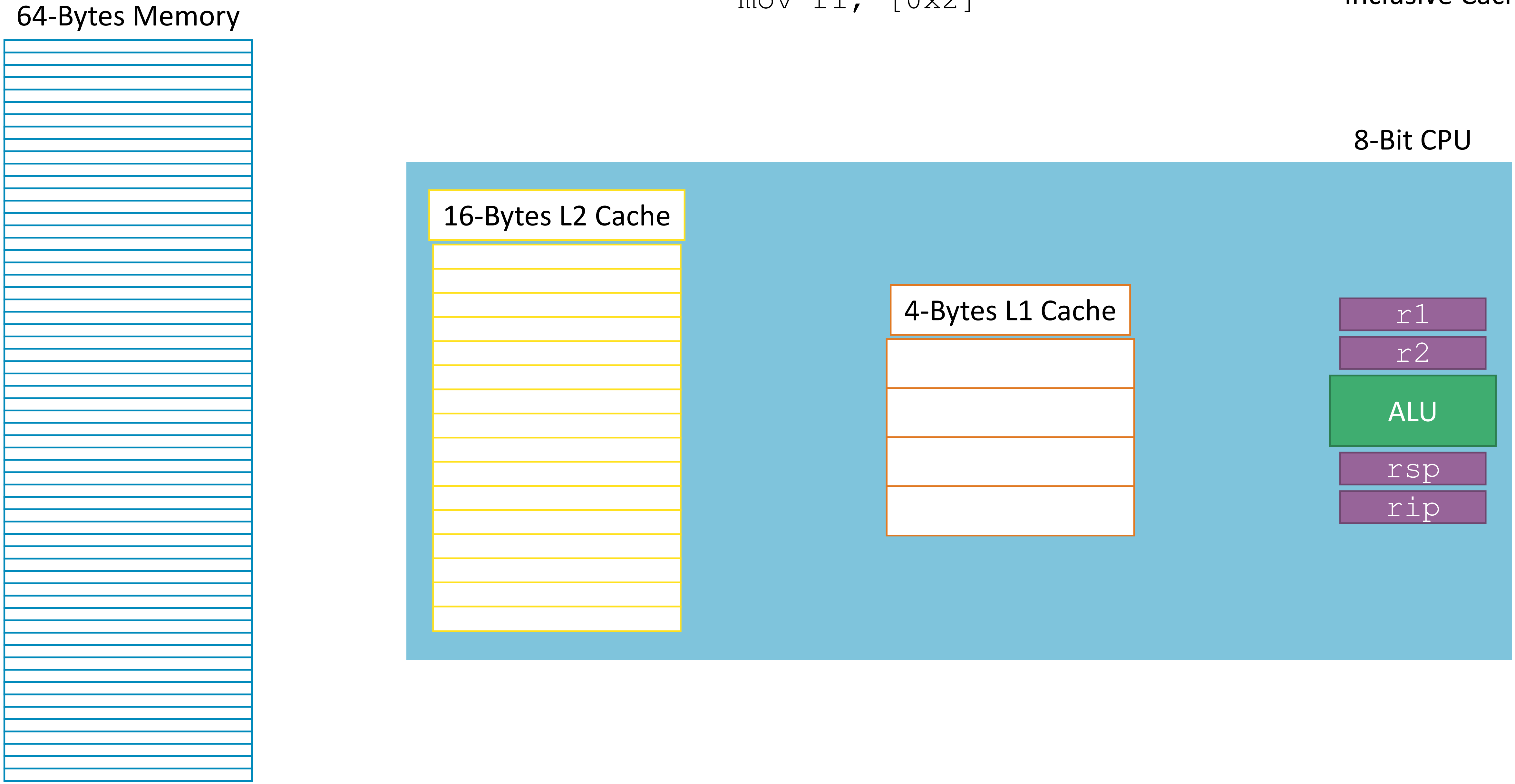
Direct-Mapped,
Inclusive Cache





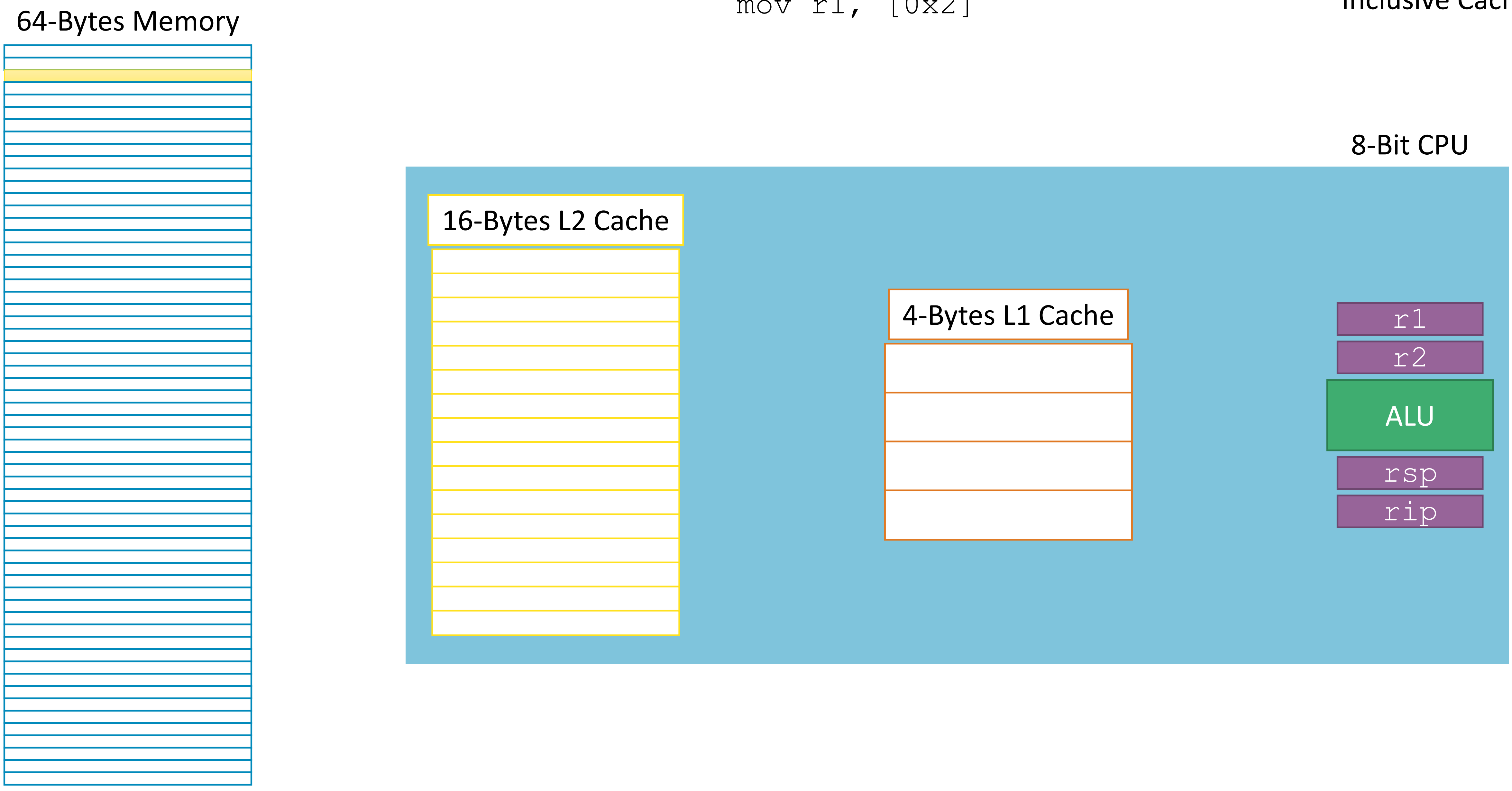
Direct-Mapped, Inclusive Cache

```
mov r1, [0x2]
```



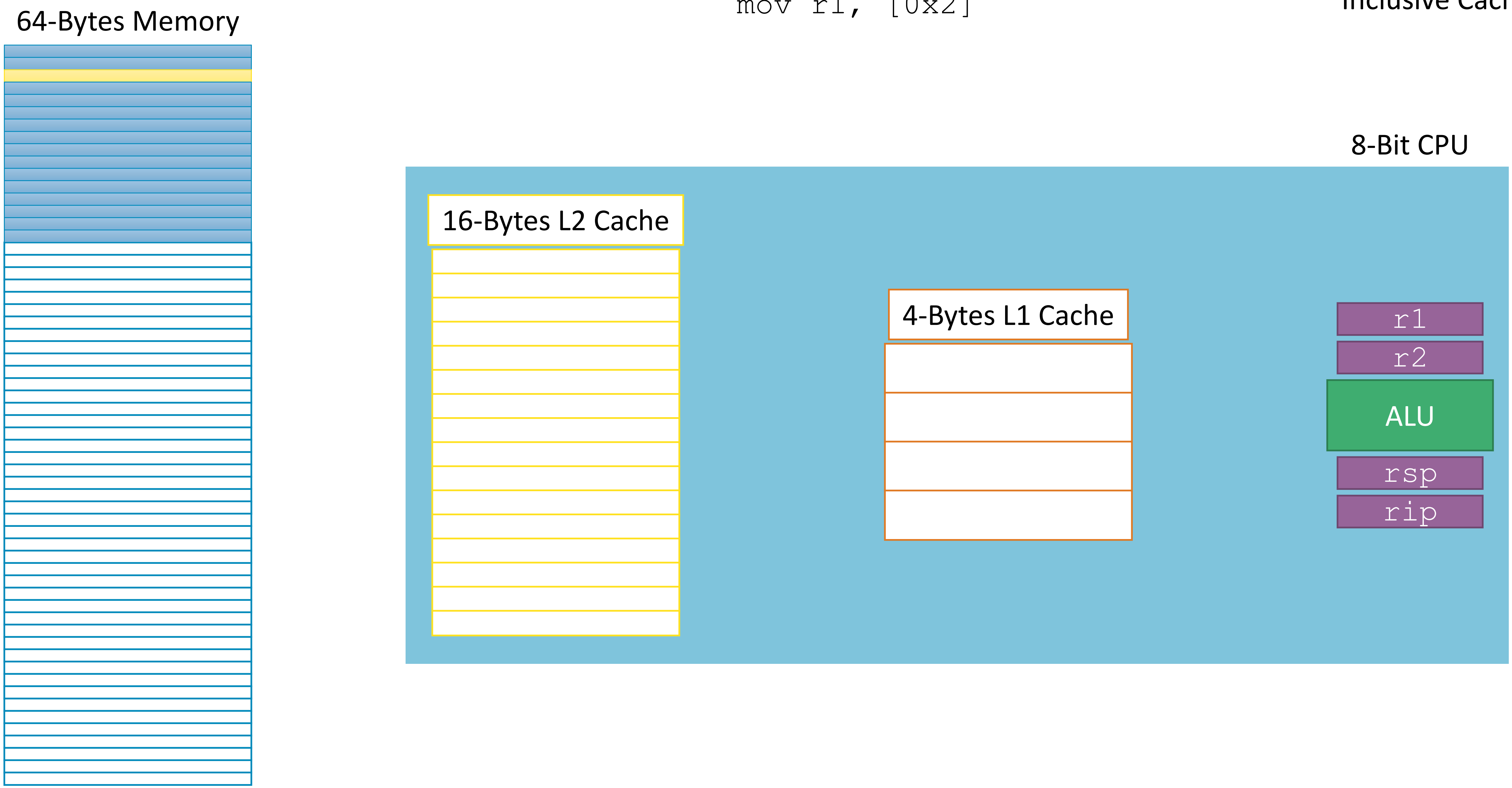
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
```



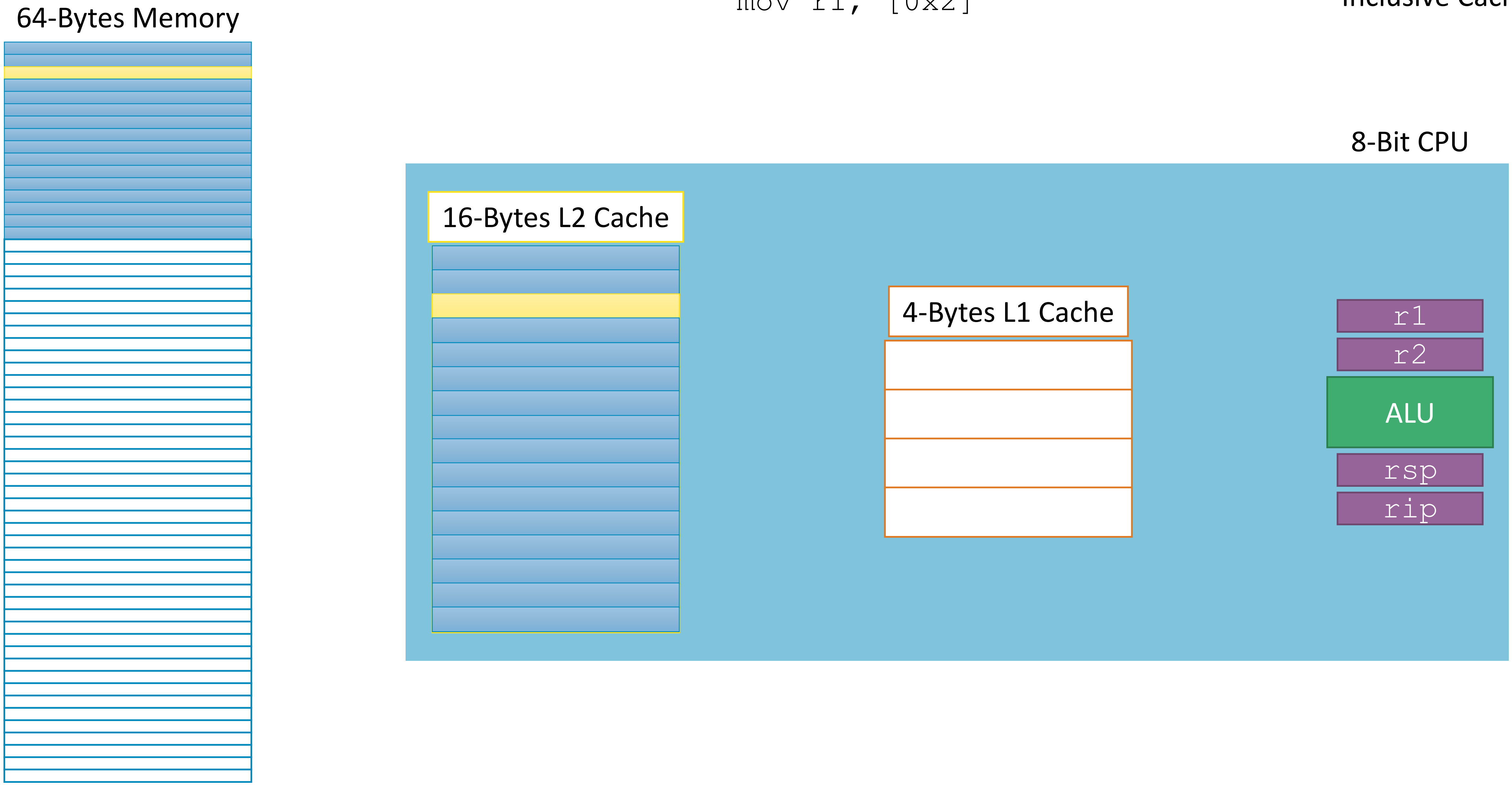
Direct-Mapped, Inclusive Cache

```
mov r1, [0x2]
```



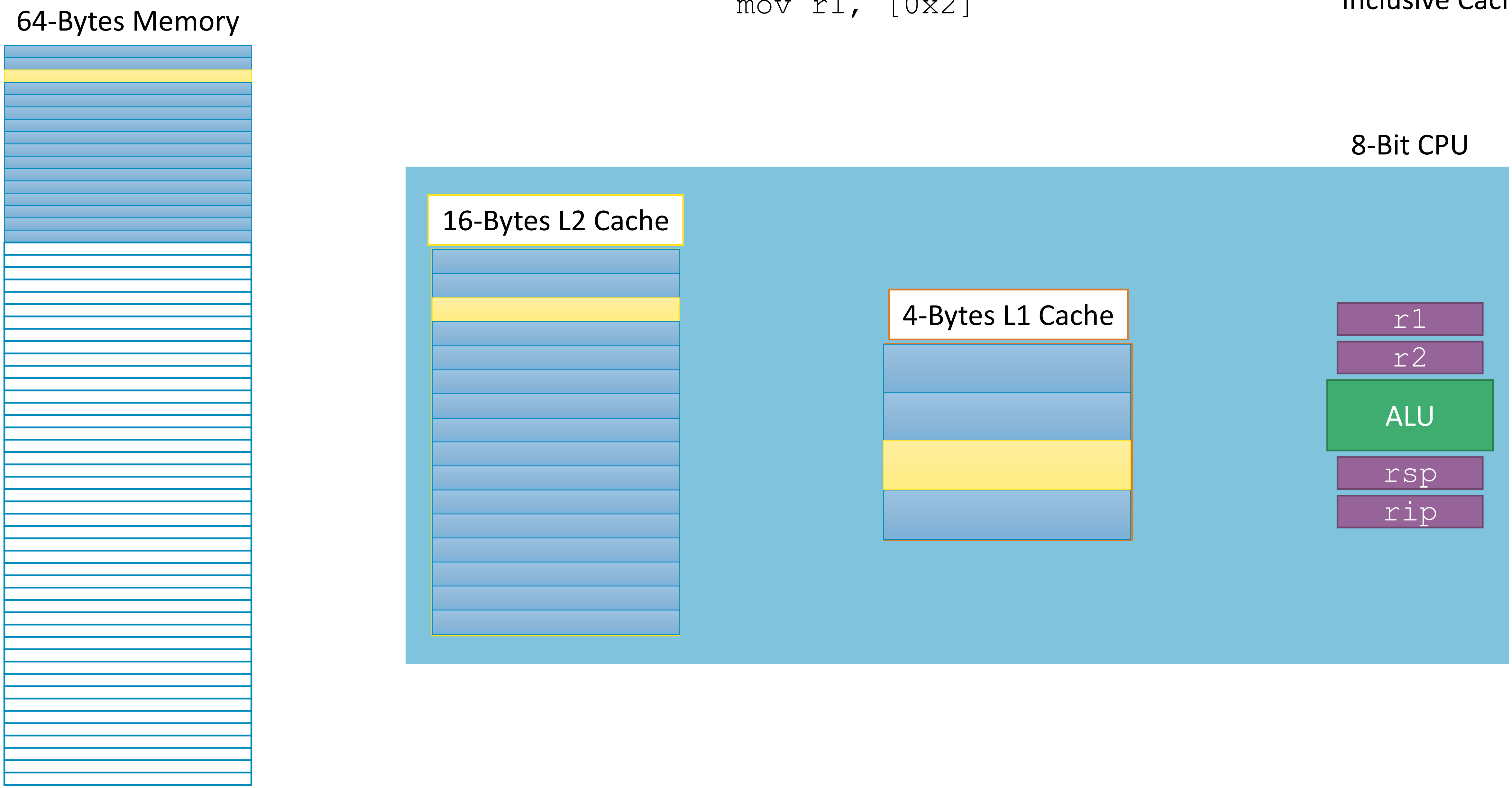
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
```



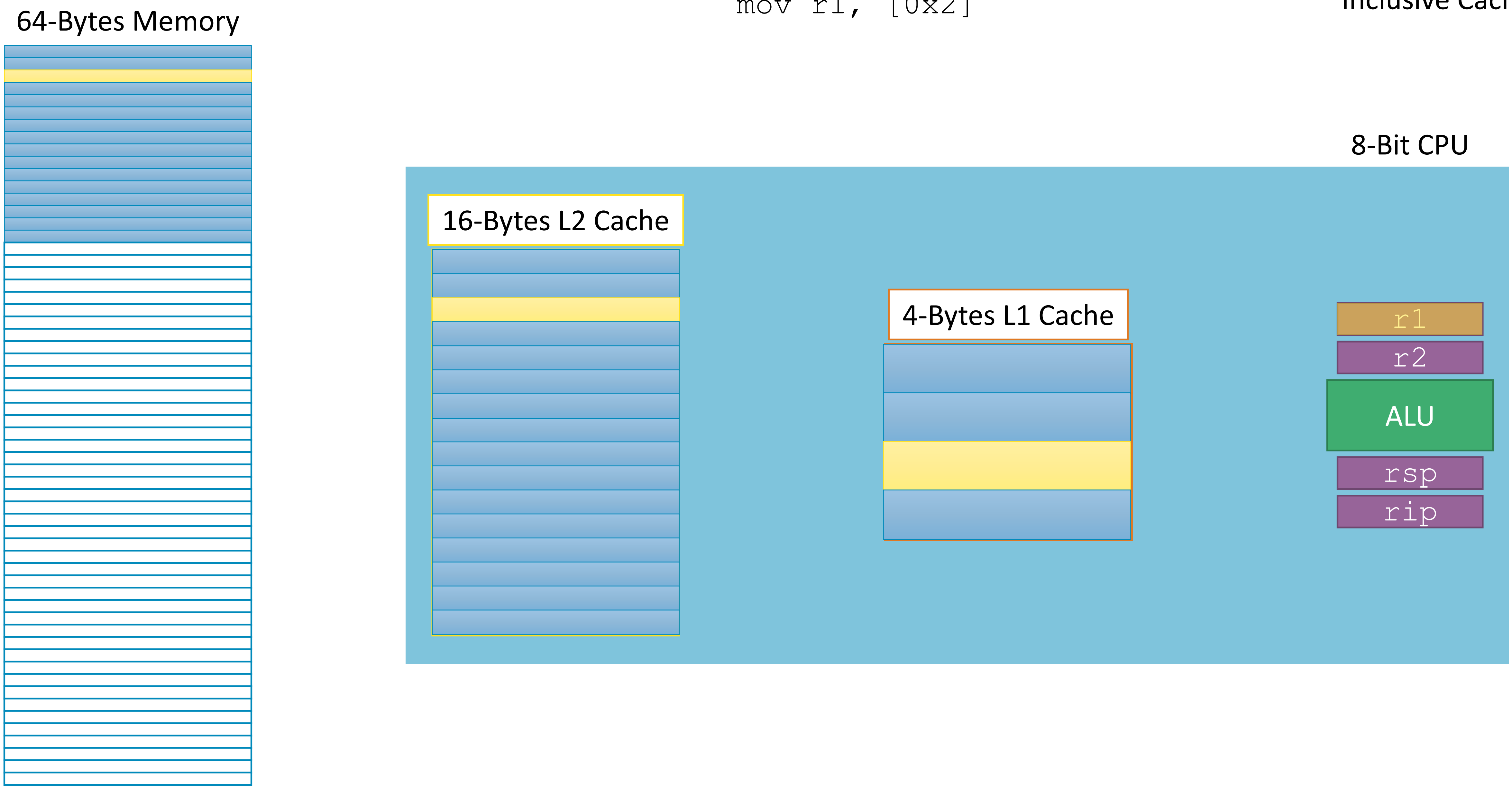
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
```



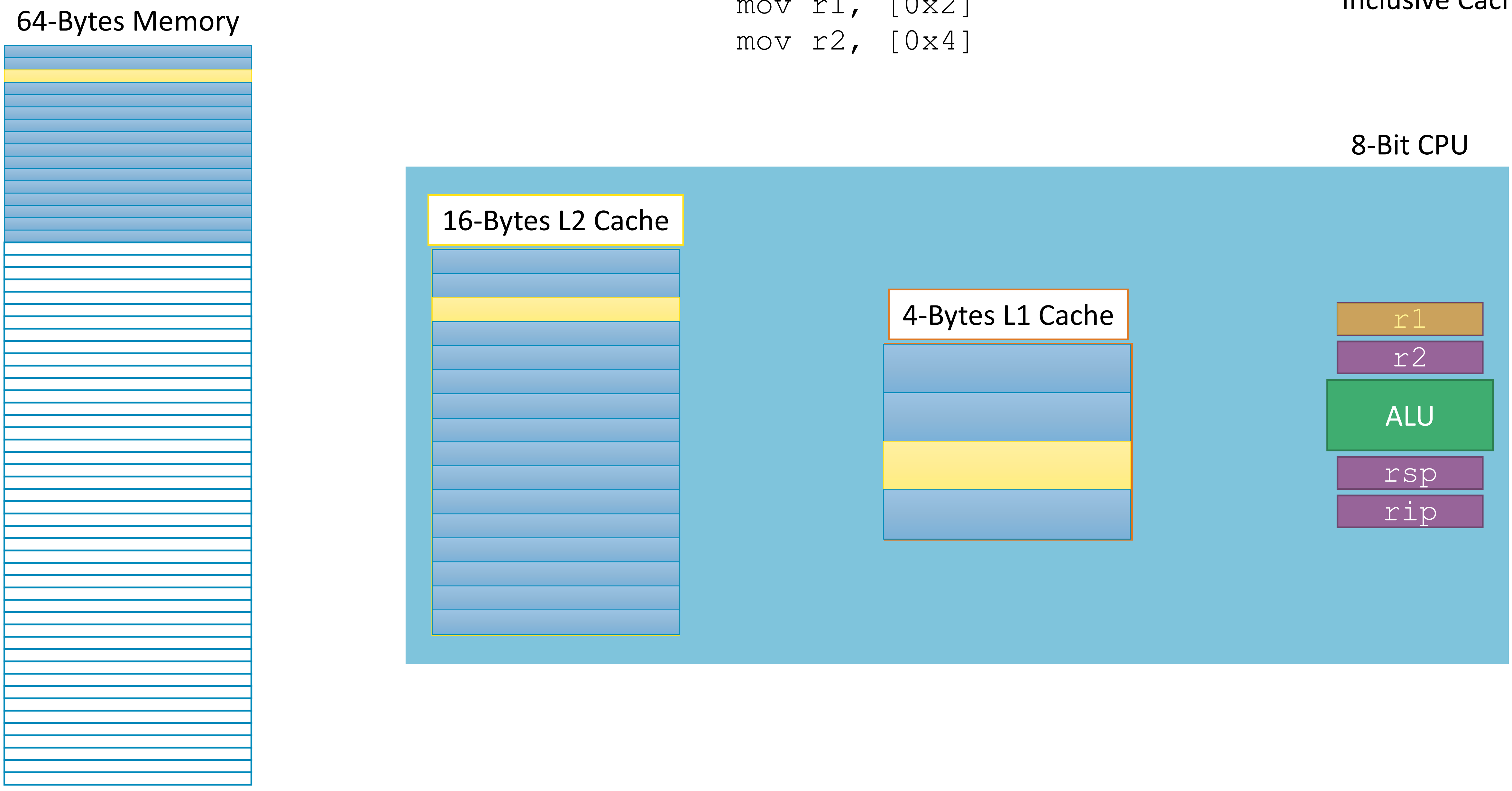
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
```



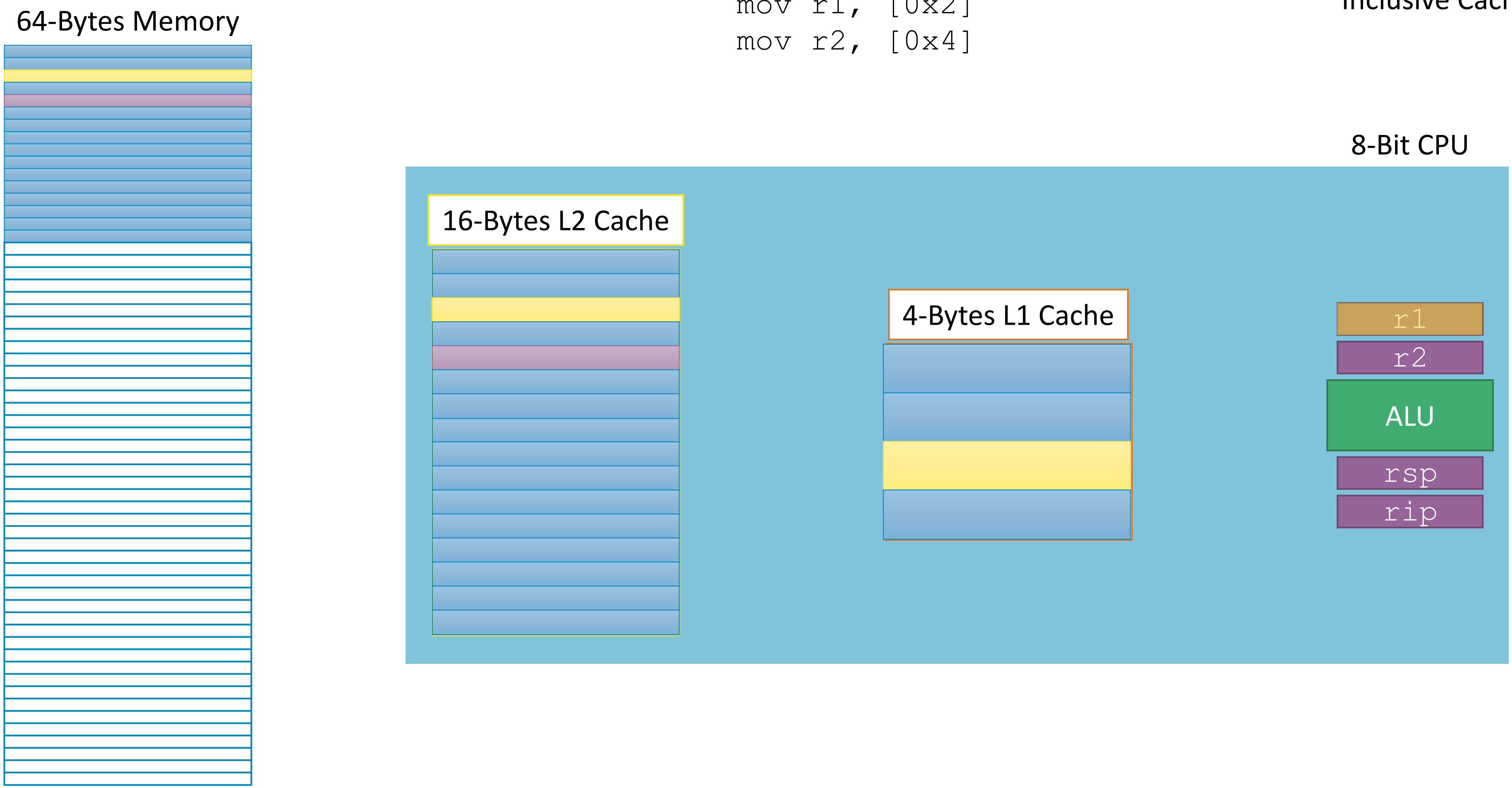
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
```



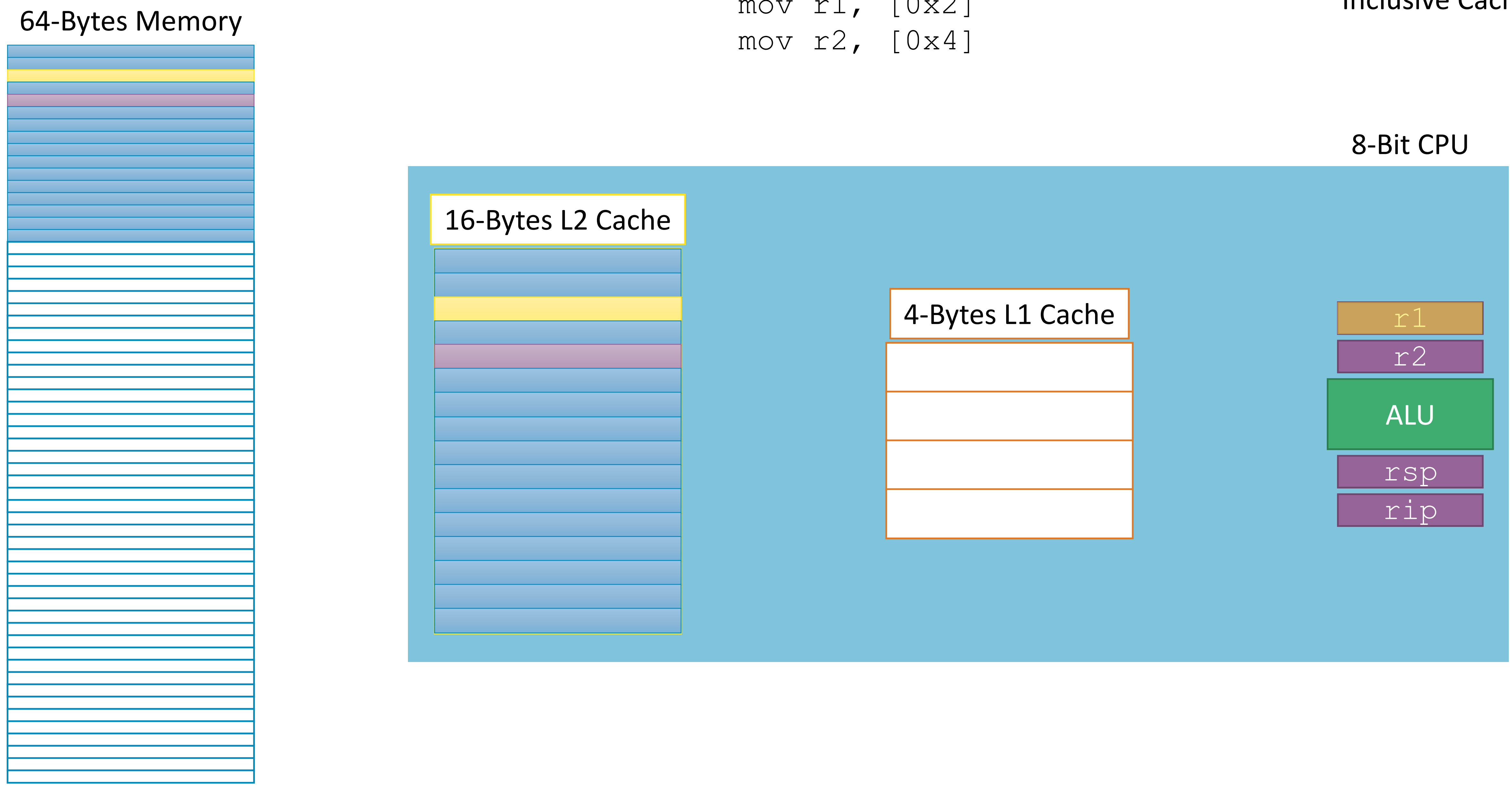
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
```



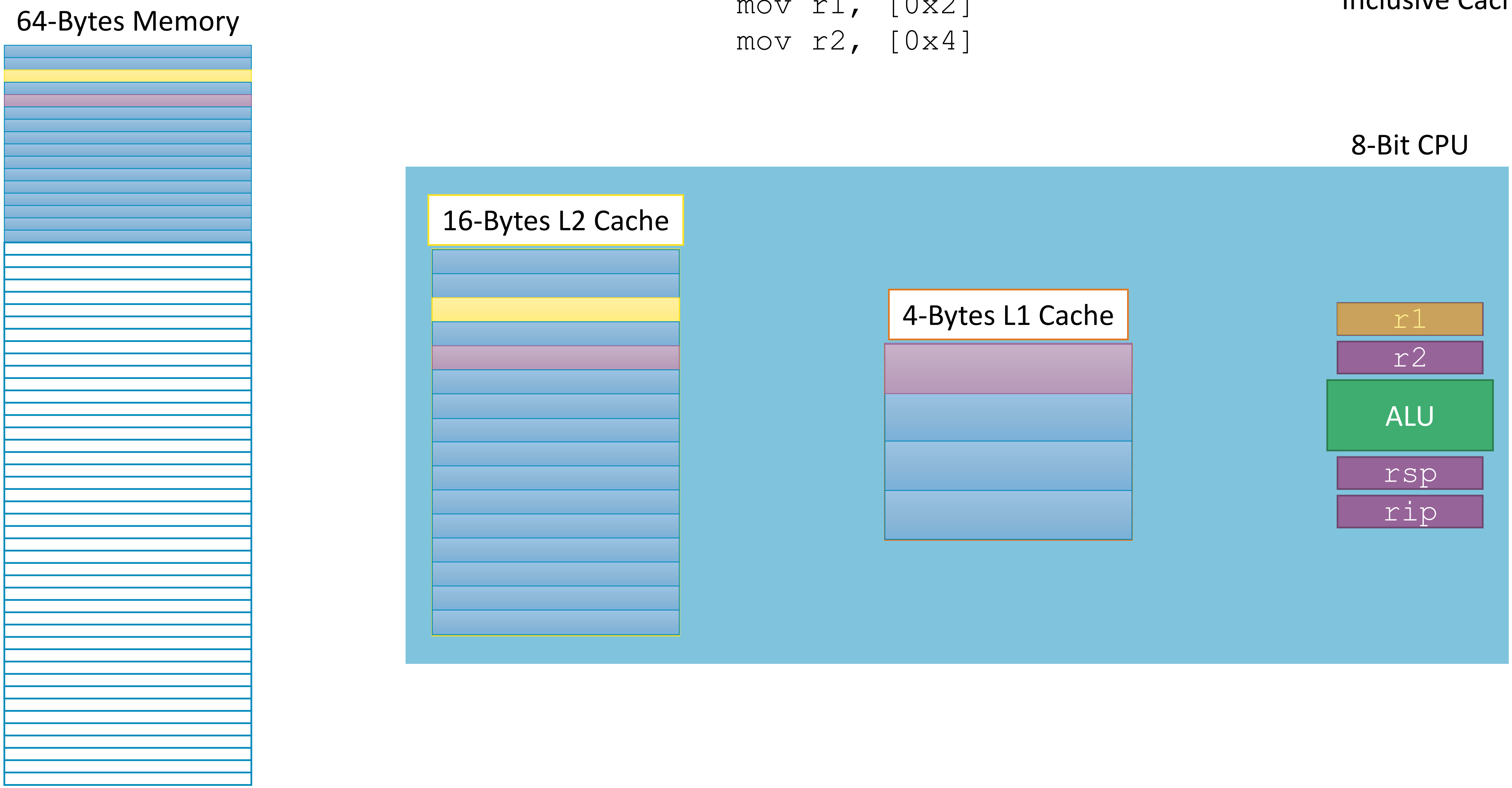
Direct-Mapped, Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
```



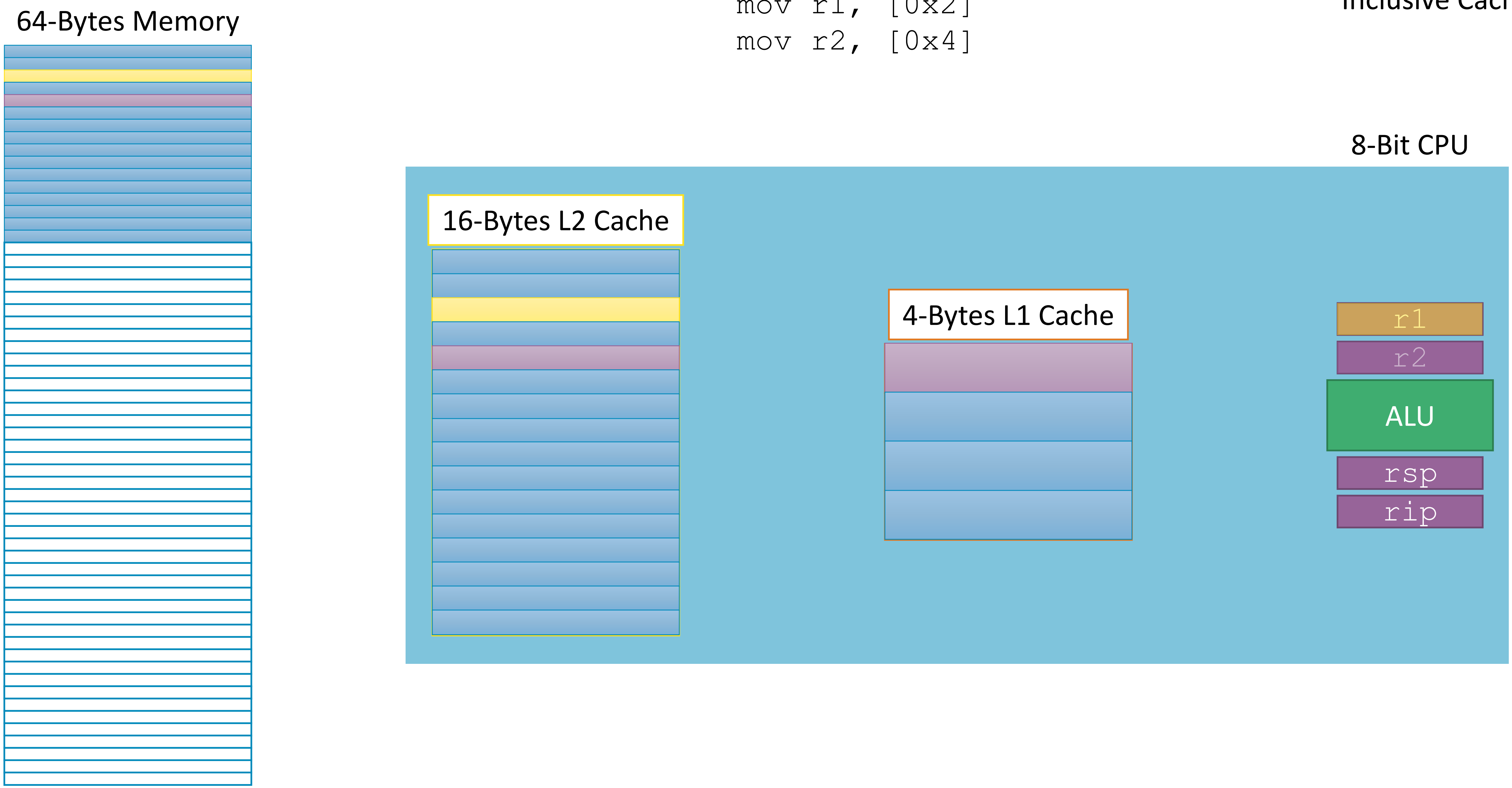
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
```



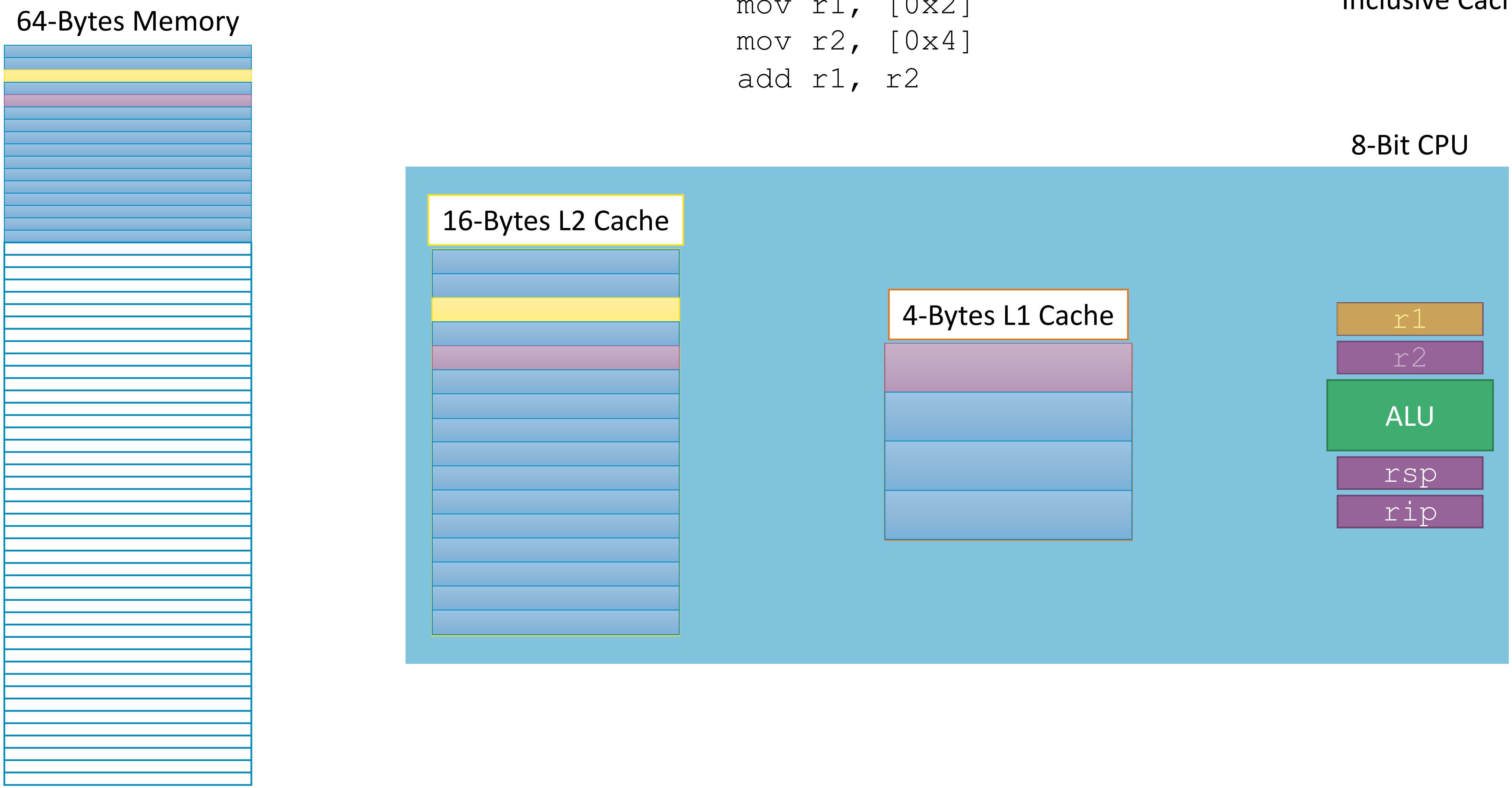
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
```



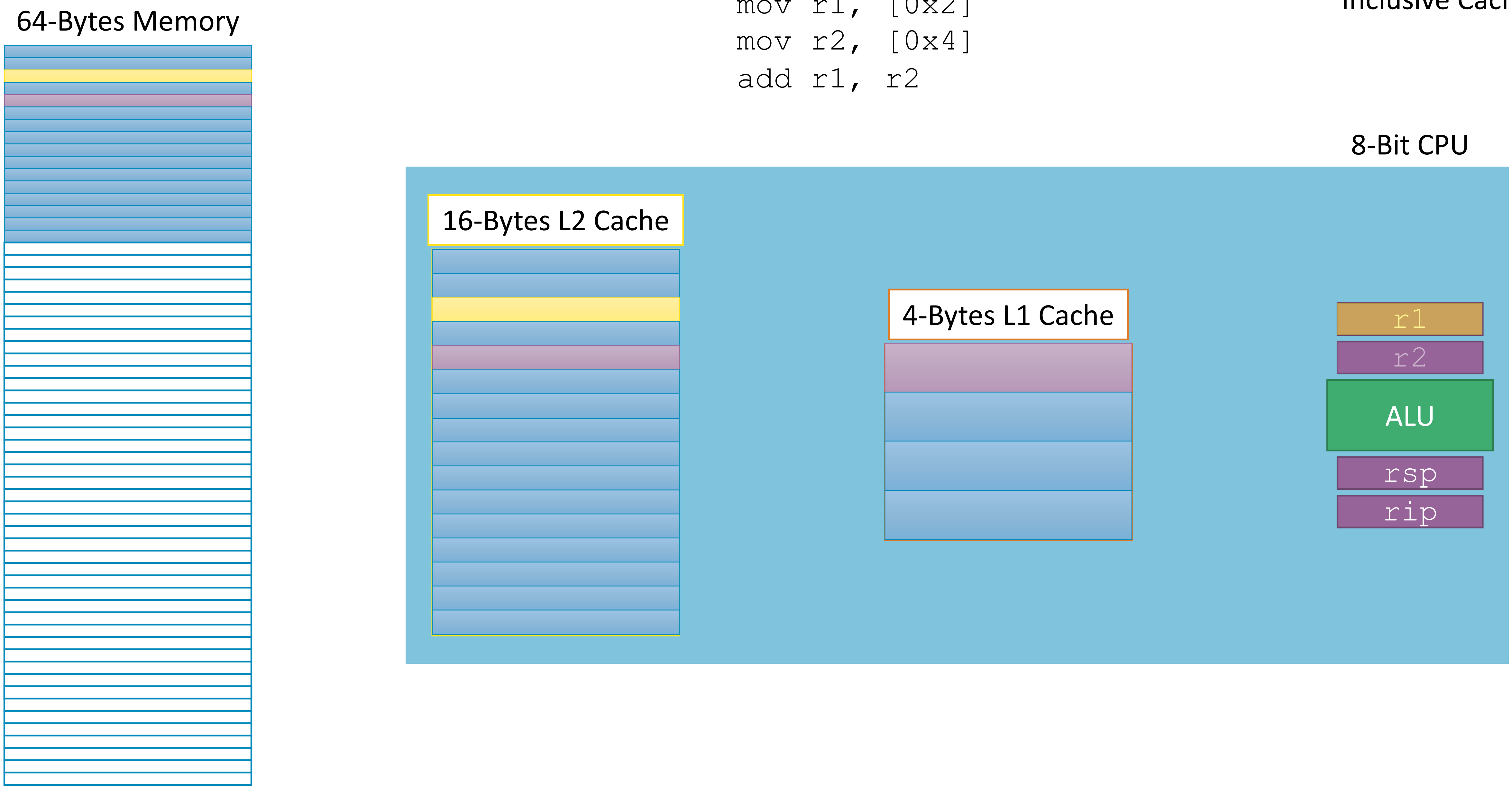
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
add r1, r2
```



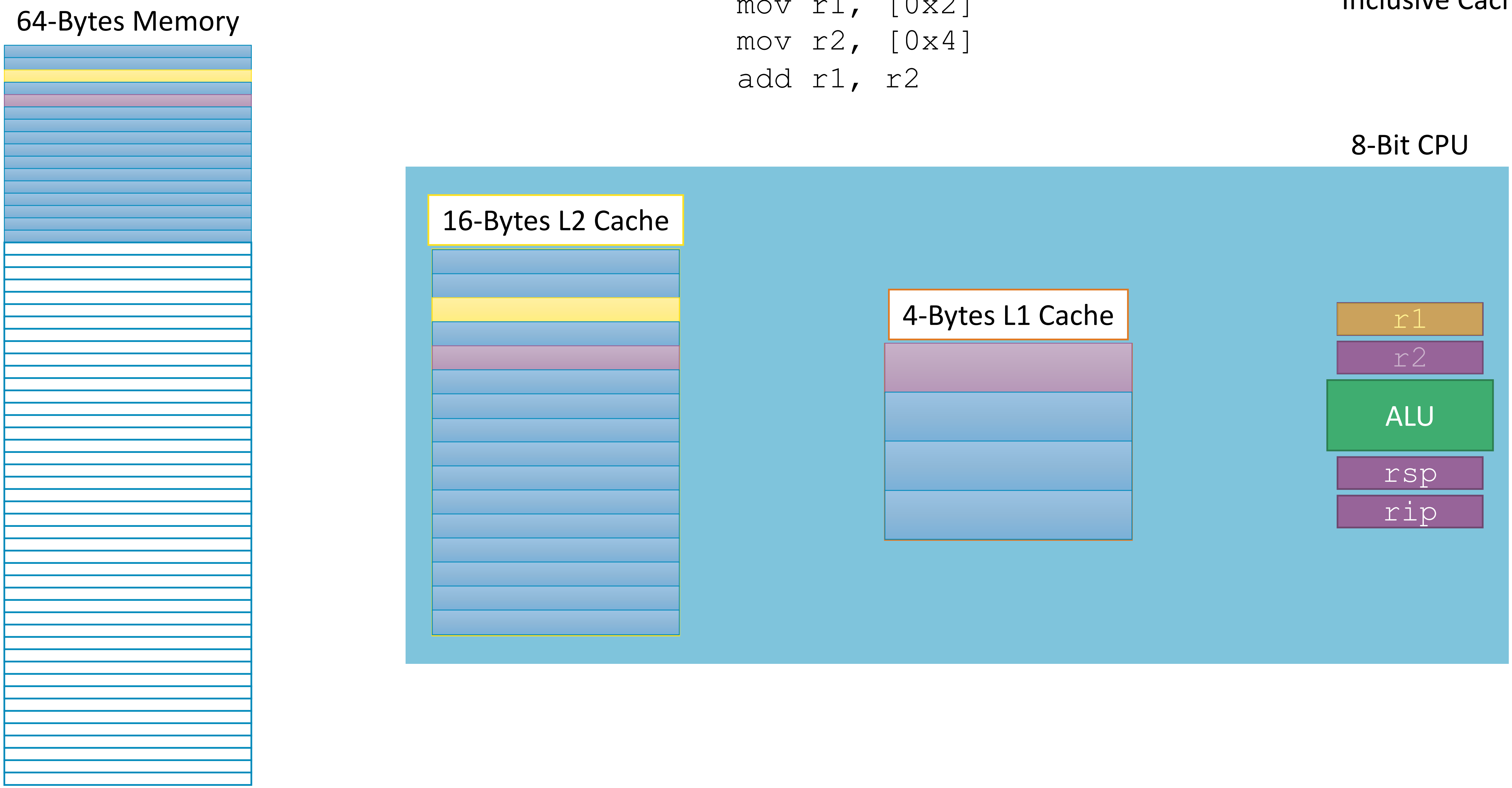
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
add r1, r2
```



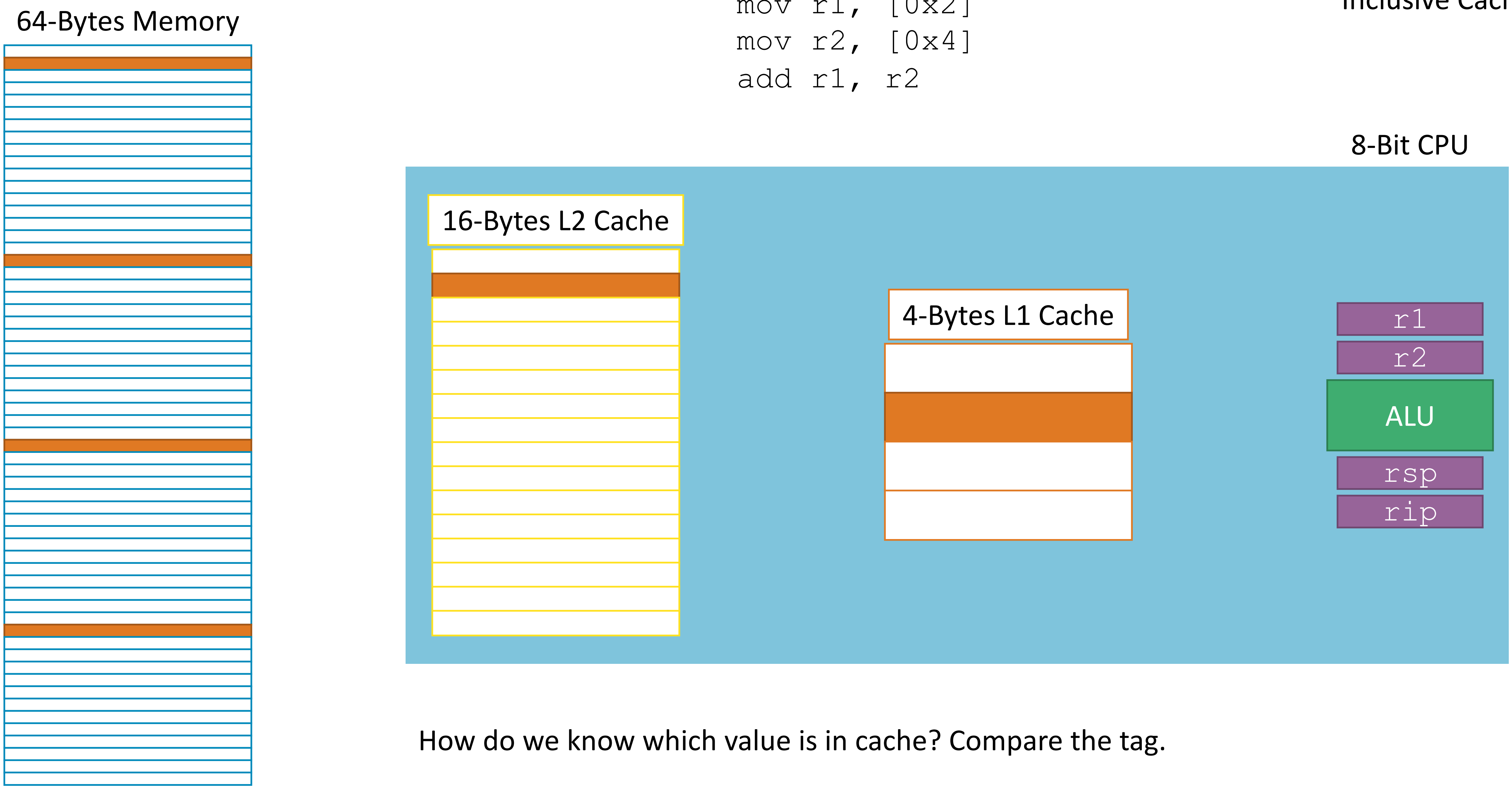
Direct-Mapped,
Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
add r1, r2
```



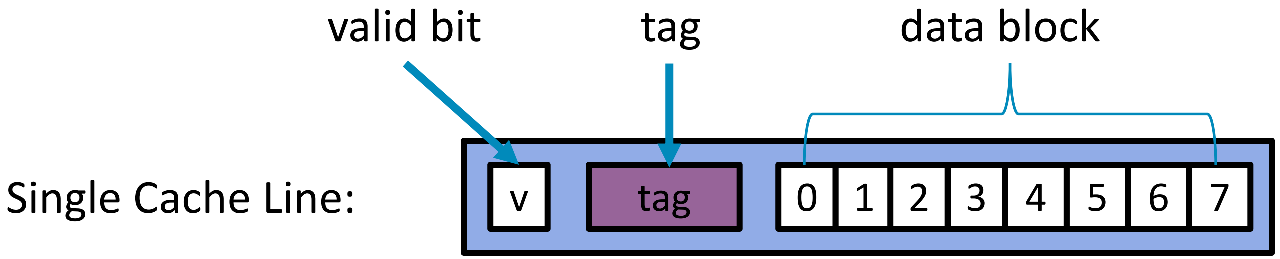
Direct-Mapped, Inclusive Cache

```
mov r1, [0x2]
mov r2, [0x4]
add r1, r2
```



How do we know which value is in cache? Compare the tag.

Cache Lines

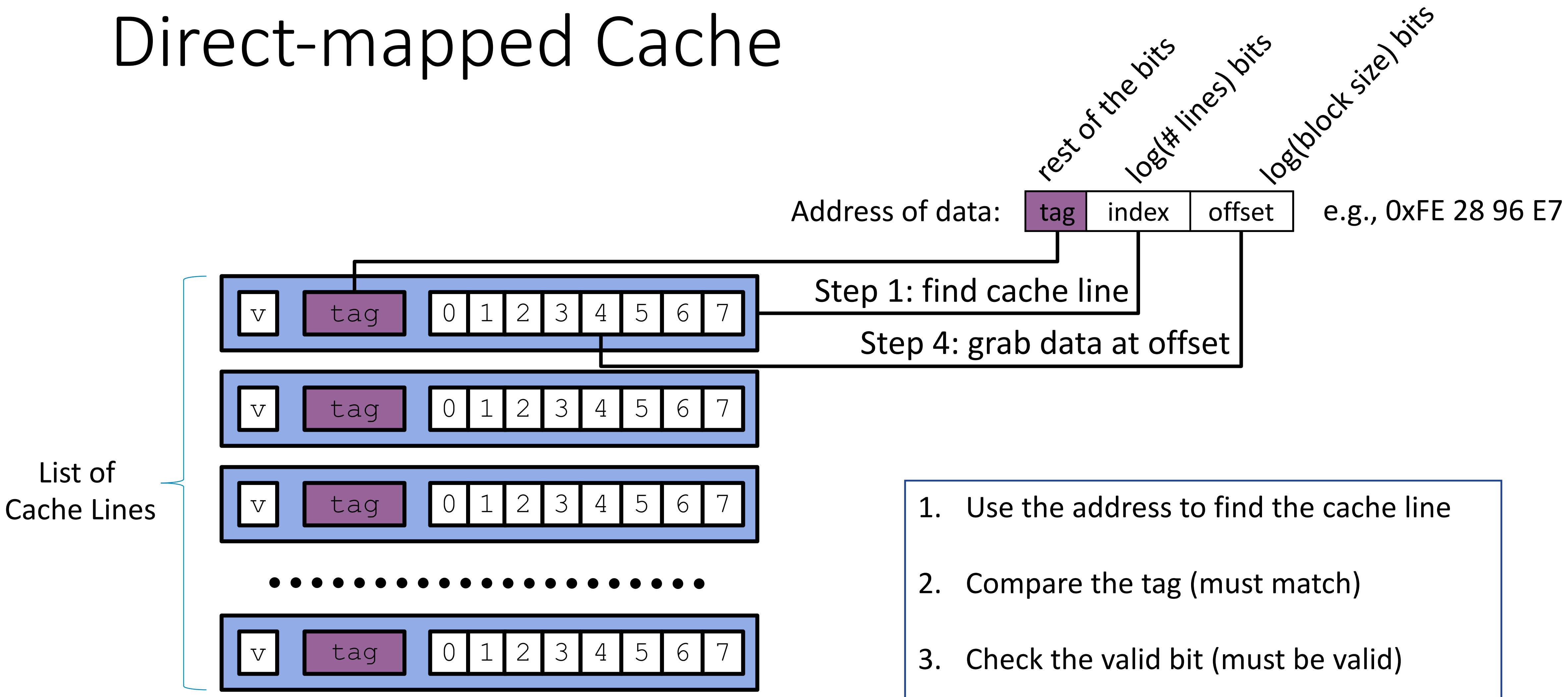


Data block: cached data (i.e., copy of bytes from memory)

Tag: uniquely identifies the data is stored in the cache line

Valid bit: indicates whether the line contains meaningful information

Direct-mapped Cache



1. Use the address to find the cache line
2. Compare the tag (must match)
3. Check the valid bit (must be valid)
4. Grab data at offset into cache line

Do the first two steps sound familiar?

Address of data:

tag	index	offset

Example: Direct-mapped Cache

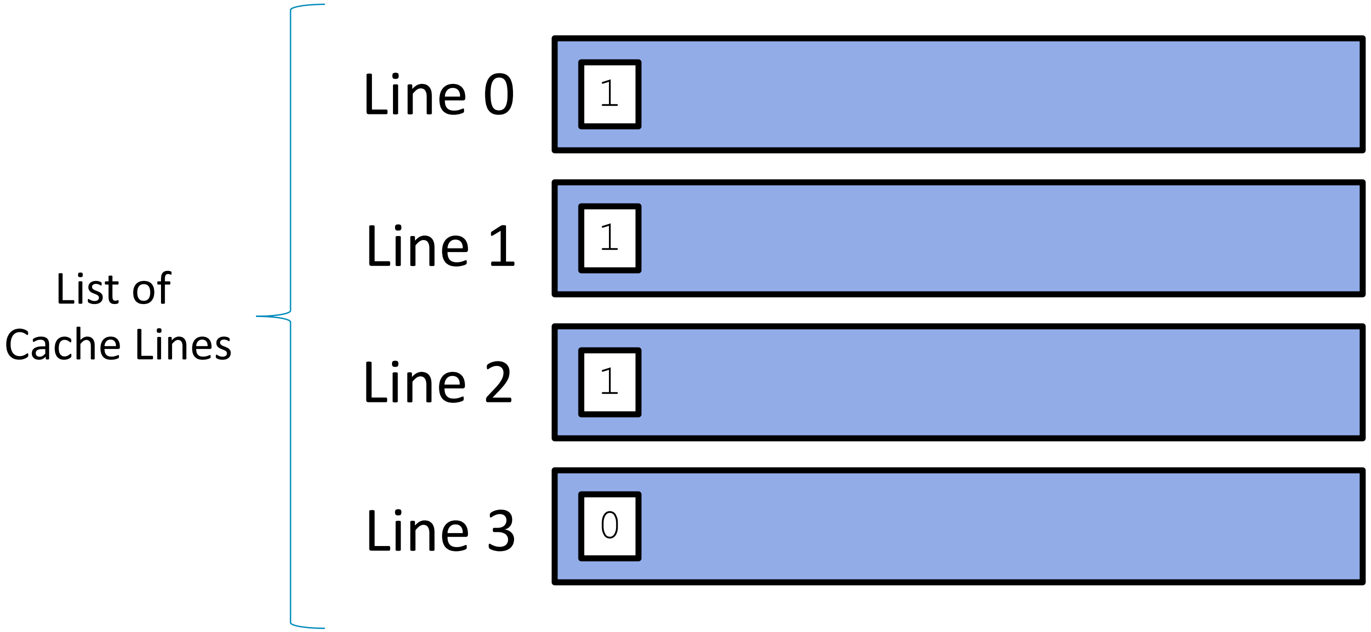
Assume: cache block size 8 bytes

Assume: assume 8-bit machine

How many bits in address?

Address of data:

?



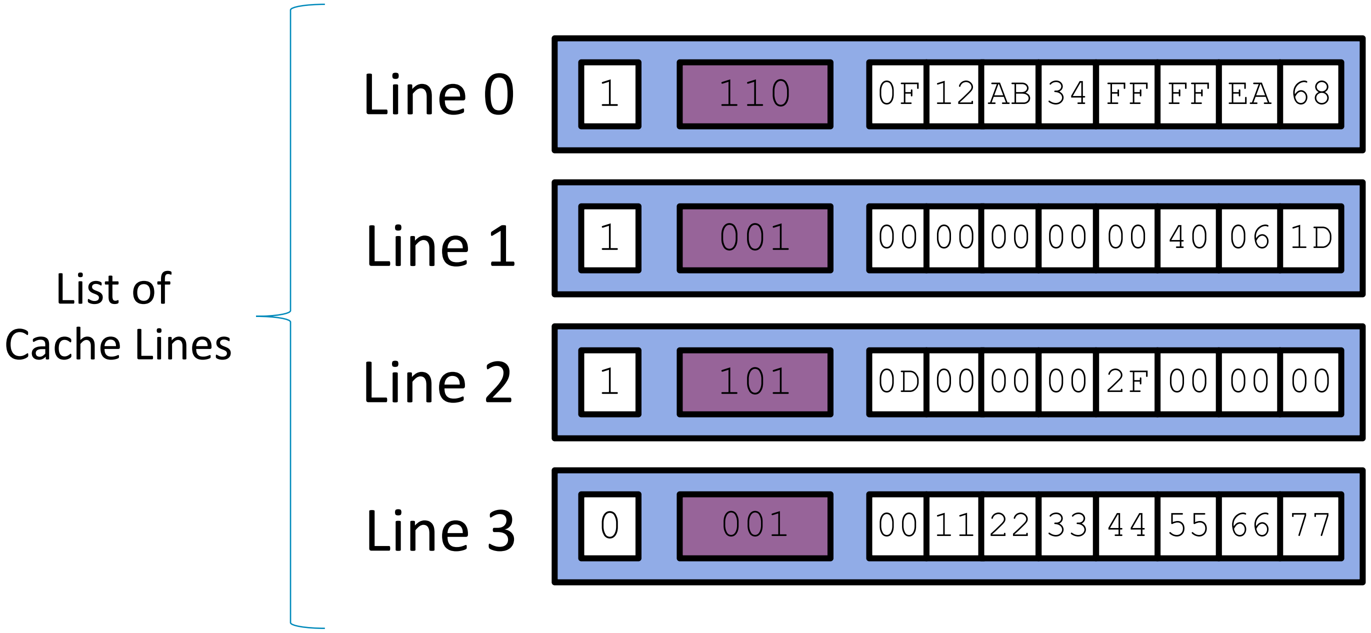
Address of data:

tag	index	offset

Example: Direct-mapped Cache

Assume: cache block size 8 bytes

Assume: assume 8-bit machine



List of Cache Lines

Line 0	1	110	0F	12	AB	34	FF	FF	EA	68
Line 1	1	001	00	00	00	00	00	40	06	1D
Line 2	1	101	0D	00	00	00	2F	00	00	00
Line 3	0	001	00	11	22	33	44	55	66	77

How many bits in address?

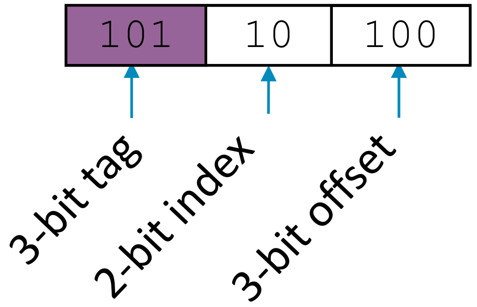
Address of data: 0xB4

1011 0100

How many bits for the index?

How many bits for the offset?

How many bits for the tag?



Practice Interpreting Addresses

Consider the hex address `0xA59`. What are the tag, index, and offset for this address with each of the following cache configurations?

1. A direct-mapped cache with 8 cache lines and 8-byte data blocks

2. A direct-mapped cache with 16 cache lines and 4-byte data blocks

3. A direct-mapped cache with 16 cache lines and 8-byte data blocks

Practice Interpreting Addresses

`1010 0101 1001`

Consider the hex address `0xA59`. What are the tag, index, and offset for this address with each of the following cache configurations?

1. A direct-mapped cache with 8 cache lines and 8-byte data blocks

Tag	Index	Offset
101001	011	001

2. A direct-mapped cache with 16 cache lines and 4-byte data blocks

Tag	Index	Off
101001	0110	01

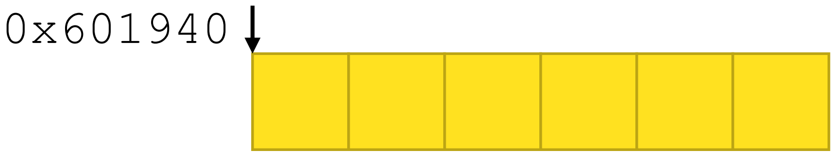
3. A direct-mapped cache with 16 cache lines and 8-byte data blocks

Tag	Index	Off
10100	1011	001

Practice with Cache Indices

You have an array of 6 `ints` (4-bytes) at address `0x601940`. Direct-mapped cache with 8 cache lines and 8-byte data blocks.

In which cache line would you find each of the 6 integers?



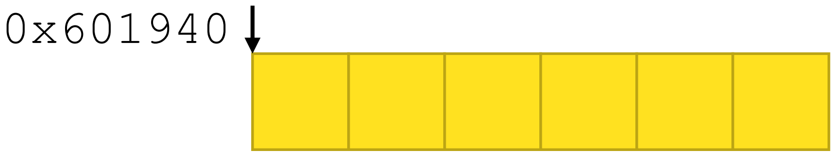
Element	Address	Binary Address	Index	Offset
`a[0]`				
`a[1]`				
`a[2]`				
`a[3]`				
`a[4]`				
`a[5]`				

How many bits for the offset?
How many bits for the index?

Practice with Cache Indices

You have an array of 6 `ints` (4-bytes) at address `0x601940`. Direct-mapped cache with 8 cache lines and 8-byte data blocks.

<u>In which cache line would you find each of the 6 integers?</u>



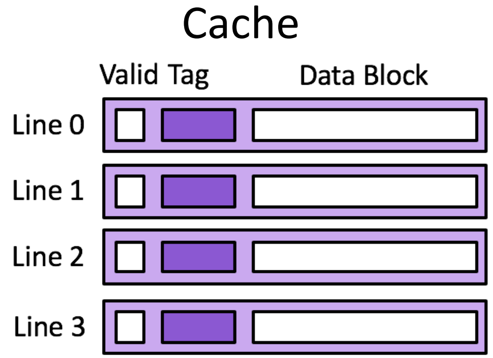
Element	Address	Binary Address	Index	Offset
`a[0]`	`0x601940`	… `0100 0000`	`000`	`000`
`a[1]`	`0x601944`	… `0100 0100`	`000`	`100`
`a[2]`	`0x601948`	… `0100 1000`	`001`	`000`
`a[3]`	`0x60194c`	… `0100 1100`	`001`	`100`
`a[4]`	`0x601950`	… `0101 0000`	`010`	`000`
`a[5]`	`0x601954`	… `0101 0100`	`010`	`100`

Practice with Direct-mapped Cache

Memory

0x14	18
0x10	17
0x0c	16
0x08	15
0x04	14
0x00	13

How many bits for the offset?
How many bits for the index?



Assume 4-byte data blocks

Binary	Access	tag	idx	off	h/m
0000 0000	rd 0x00	0000	00	00	m
0000 0100	rd 0x04				
0001 0100	rd 0x14				
0000 0000	rd 0x00				
0000 0100	rd 0x04				
0001 0000	rd 0x14				

Line 0			Line 1			Line 2			Line 3		
0	0000	47	0	0000	47	0	0000	47	0	0000	47

Time ↓

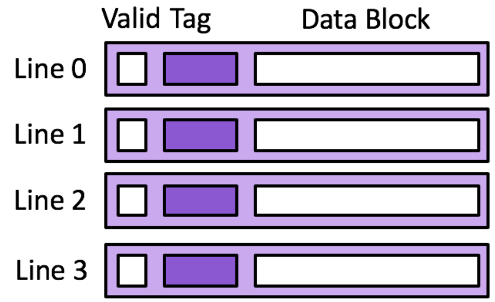
Only showing updates to the cache.

Practice with Direct-mapped Cache

Memory

0x14	18
0x10	17
0x0c	16
0x08	15
0x04	14
0x00	13

Cache



Assume 4-byte data blocks

						Line 0			Line 1			Line 2			Line 3		
Binary	Access	tag	idx	off	h/m	0	0000	47	0	0000	47	0	0000	47	0	0000	47
0000 0000	rd 0x00	0000	00	00	m	1	0000	13									
0000 0100	rd 0x04																
0001 0100	rd 0x14																
0000 0000	rd 0x00																
0000 0100	rd 0x04																
0001 0000	rd 0x14																

Time

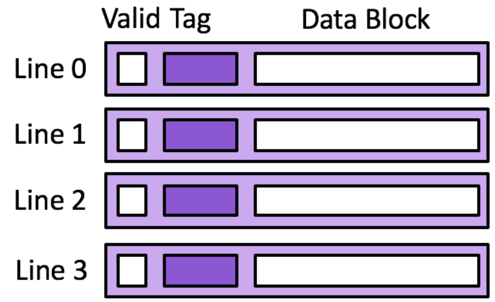
Only showing updates to the cache.

Practice with Direct-mapped Cache

Memory

0x14	18
0x10	17
0x0c	16
0x08	15
0x04	14
0x00	13

Cache



Assume 4-byte data blocks

						Line 0			Line 1			Line 2			Line 3		
Binary	Access	tag	idx	off	h/m	0	0000	47	0	0000	47	0	0000	47	0	0000	47
0000 0000	rd 0x00	0000	00	00	m	1	0000	13									
0000 0100	rd 0x04	0000	01	00	m				1	0000	14						
0001 0100	rd 0x14	0001	01	00	m				1	0001	18						
0000 0000	rd 0x00	0000	00	00	h												
0000 0100	rd 0x04	0000	01	00	m				1	0000	14						
0001 0000	rd 0x14	0001	01	00	m				1	0001	18						

Time

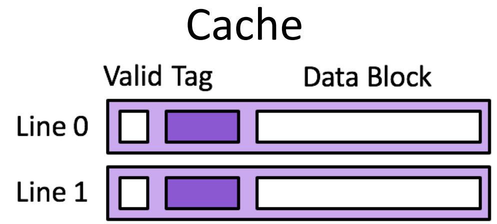
Only showing updates to the cache.

More Practice with Direct-mapped Cache

Same memory and same code

Memory

0x14	18
0x10	17
0x0c	16
0x08	15
0x04	14
0x00	13



Assume 8-byte data blocks

Access	tag	idx	off	h/m
rd 0x00				
rd 0x04				
rd 0x14				
rd 0x00				
rd 0x04				
rd 0x14				

Line 0			Line 1		
0	0000	47 48	0	0000	47 48

Time ↓

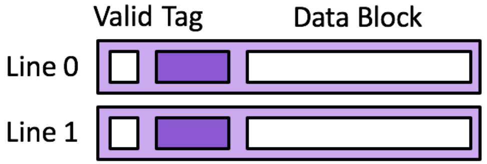
More Practice with Direct-mapped Cache

Same memory
and same code

Memory

0x14	18
0x10	17
0x0c	16
0x08	15
0x04	14
0x00	13

Cache



Assume 8-byte data blocks

Access	tag	idx	off	h/m
rd 0x00	0000	0	000	m
rd 0x04	0000	0	100	h
rd 0x14	0001	0	100	m
rd 0x00	0000	0	000	m
rd 0x04	0000	0	000	h
rd 0x14	0001	0	000	m

Line 0				Line 1			
0	0000	47	48	0	0000	47	48
1	0000	13	14				
1	0001	17	18				
1	0000	13	14				
1	0001	17	18				

Time

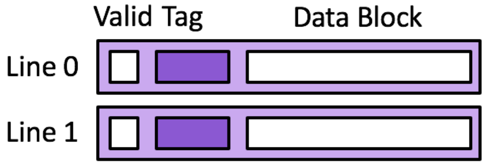
More Practice with Direct-mapped Cache

Same memory and same code

Memory

0x14	18
0x10	17
0x0c	16
0x08	15
0x04	
0x00	

Cache



How well does this take advantage of spacial locality?

How well does this take advantage of temporal locality?

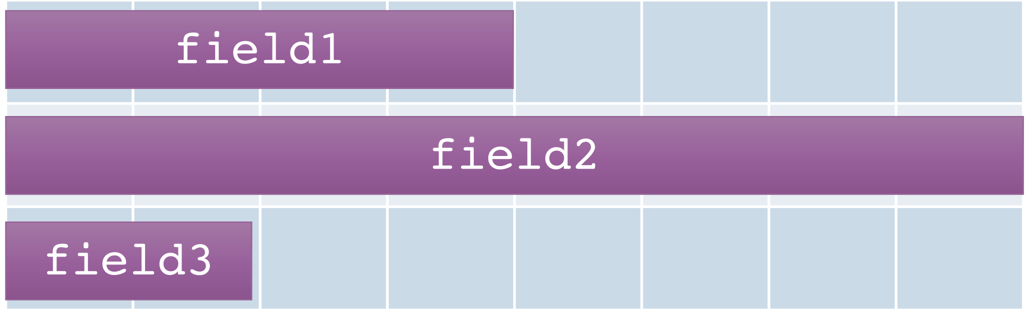
										1
Access										47 48
rd 0x00										
rd 0x04										
rd 0x14	0001	0	100	m	1	0001	17 18			
rd 0x00	0000	0	000	m	1	0000	13 14			
rd 0x04	0000	0	000	h						
rd 0x14	0001	0	000	m	1	0001	17 18			

Time

Alignment

- Modern process mostly allow *unaligned* data access
- Unaligned access: an n-byte piece of data with an address not divisible by n
- But most system programming languages still align all data for performance reasons (it matters less now than it used to)

```
struct data {
    u32 field1;
    u64 field2;
    u16 field3;
};
```

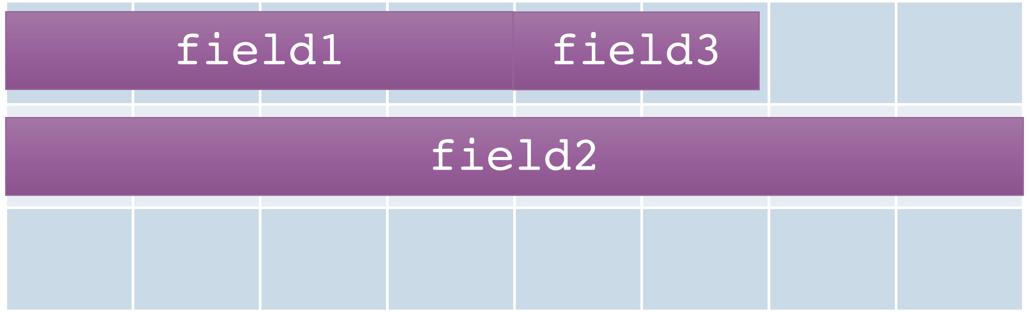


64-bits wide

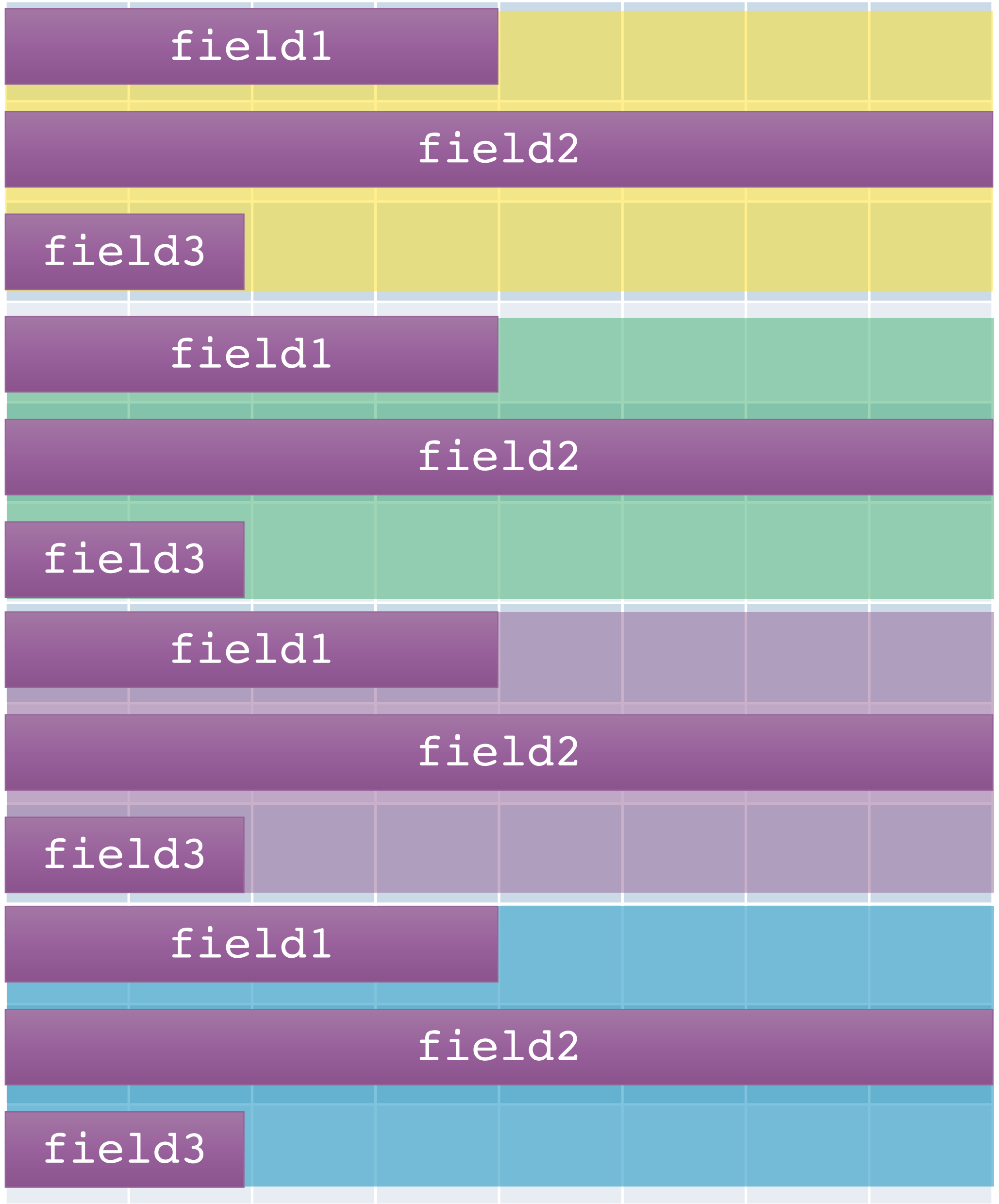
Alignment

- Modern process mostly allow *unaligned* data access
- Unaligned access: an n-byte piece of data with an address not divisible by n
- But most system programming languages still align all data for performance reasons (it matters less now than it used to)

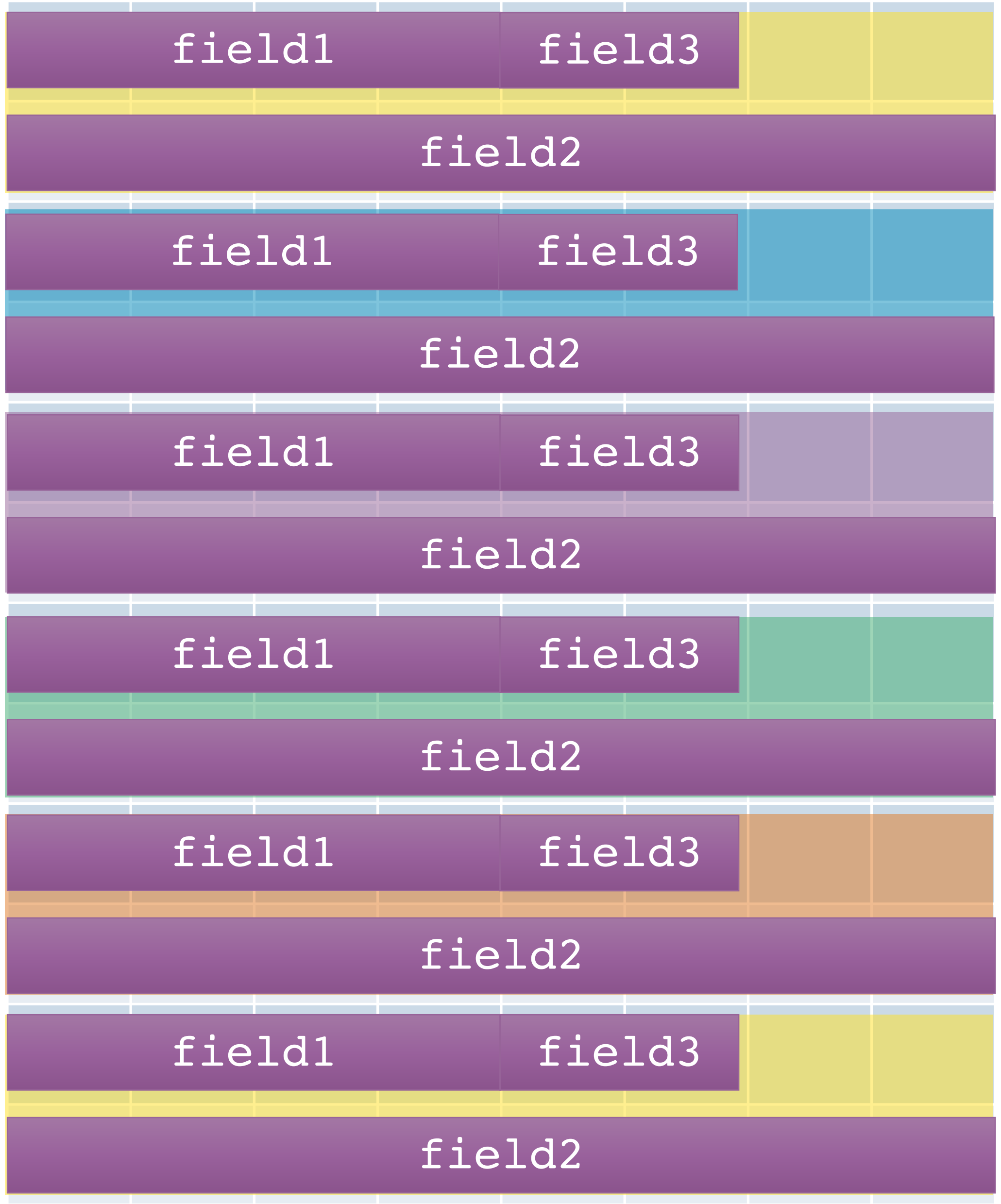
```
struct data {
    u32 field1;
    u16 field3;
    u64 field2;
};
```



64-bits wide



64-bits wide



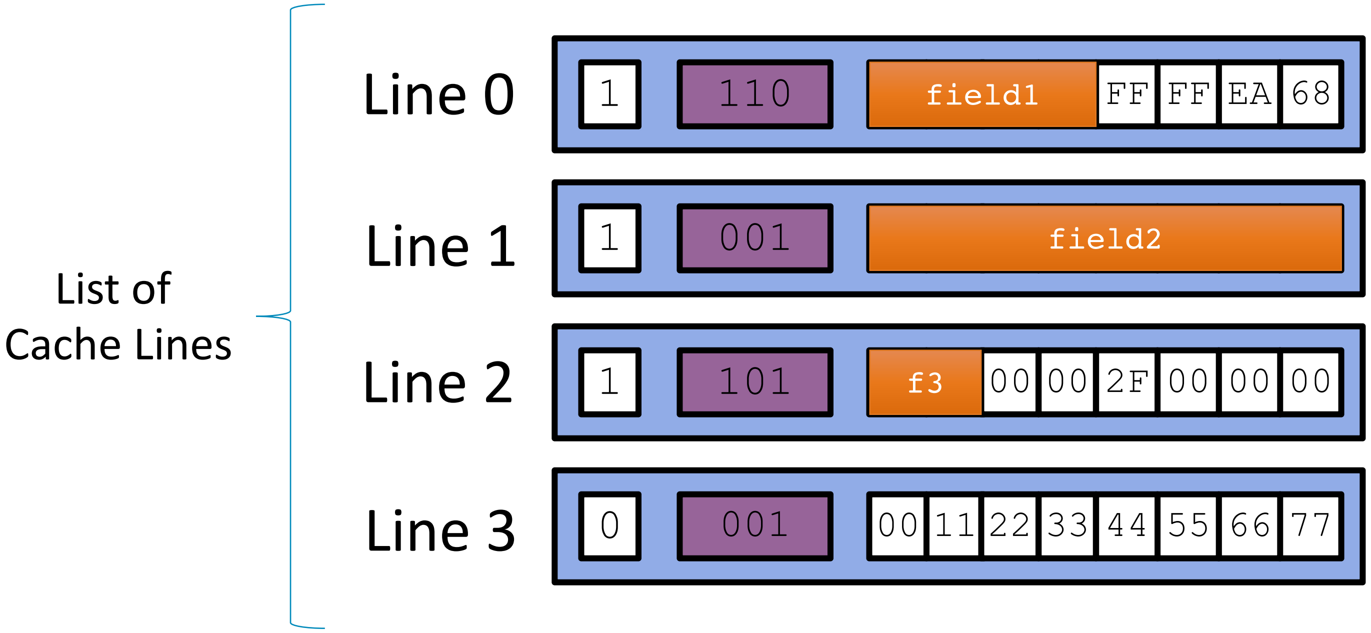
64-bits wide

Address of data:	tag	index	offset

Cache and Alignment

Assume: cache block size 8 bytes

Assume: assume 8-bit machine



List of Cache Lines

Line 0	1	110	field1 FF FF EA 68
Line 1	1	001	field2
Line 2	1	101	f3 00 00 2F 00 00 00
Line 3	0	001	00 11 22 33 44 55 66 77

How many bits in address?

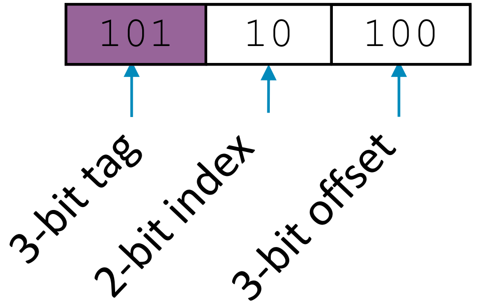
Address of data: 0xB4

1011 0100

How many bits for the index?

How many bits for the offset?

How many bits for the tag?

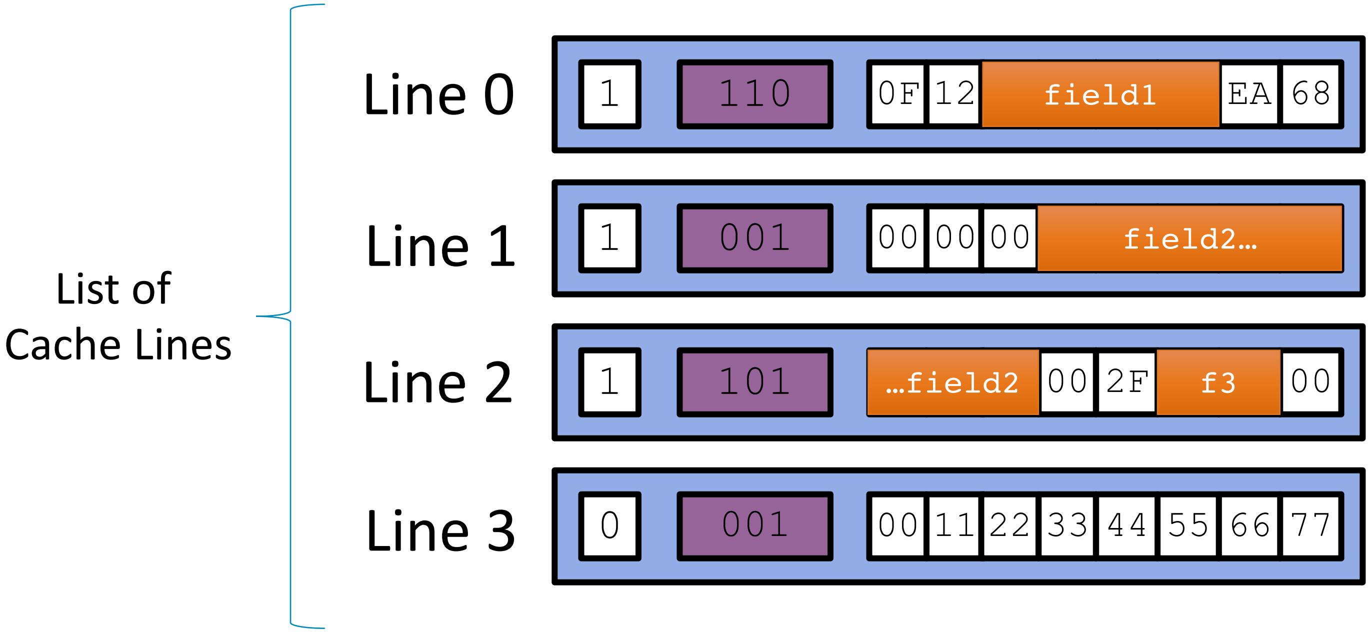


Address of data: | tag | index | offset |

Cache and Alignment

Assume: cache block size 8 bytes

Assume: assume 8-bit machine



List of Cache Lines

Line 0	1	110	0F 12 field1 EA 68
Line 1	1	001	00 00 00 field2…
Line 2	1	101	…field2 00 2F f3 00
Line 3	0	001	00 11 22 33 44 55 66 77

How many bits in address?

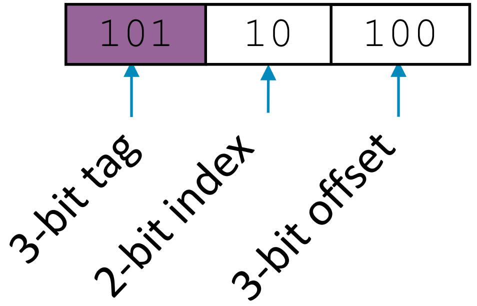
Address of data: 0xB4

1011 0100

How many bits for the index?

How many bits for the offset?

How many bits for the tag?


101
10
100
3-bit tag
2-bit index
3-bit offset