CS33 DISCUSSION 6

CACHES

A Memory Problem

- Many instructions reference Memory
  - Instruction fetch, Load/Store
- Typical Access Time for DRAM ~ 60ns
- For 3 GHz Processor → ~ 200 cycles
  - Should we wait that long per access?

Memory Hierarchy

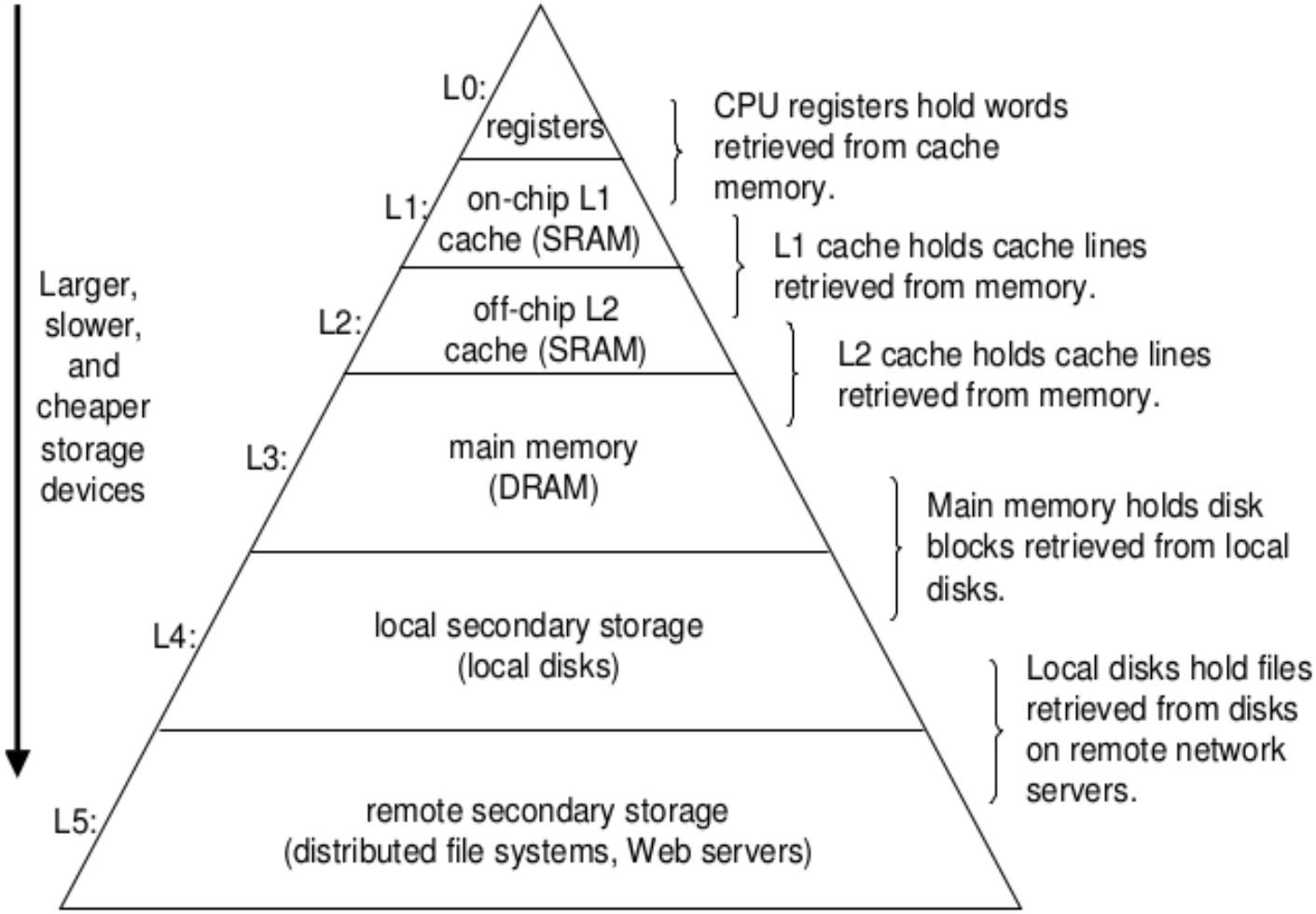


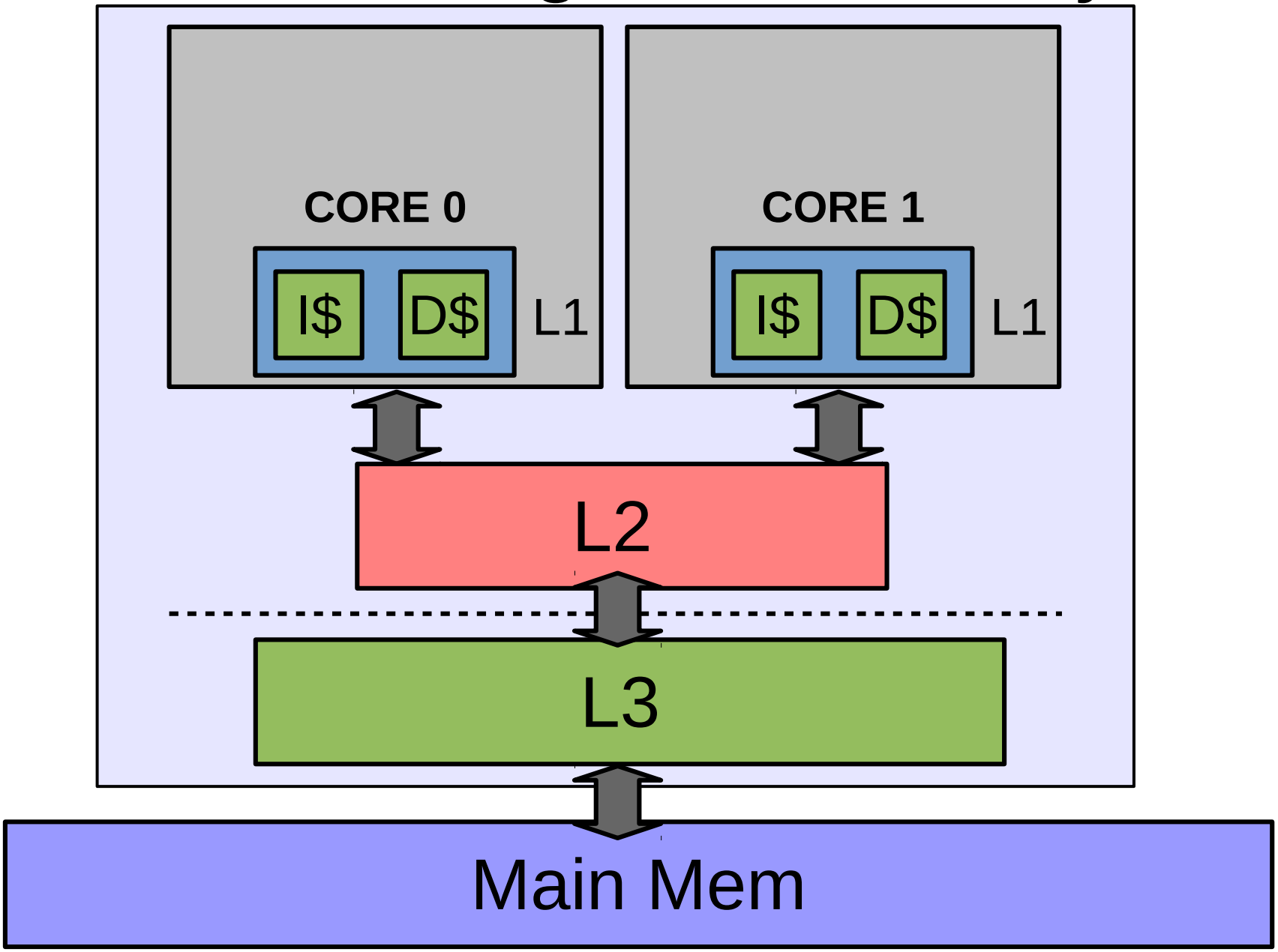
Figure 1.9: **The memory hierarchy.**

LET'S USE LOCALITY!

Two Types of Locality

- **Temporal**: Subsequent access to data are nearby in time
  - Loop code, accumulators, iterators
- **Spatial**: Data accesses occur at nearby locations
  - Aggregates (arrays, structs, etc), sequential code

Caching in on Locality



General Idea

- Idea: lets maintain a working subset of memory in $
- Divide memory into blocks

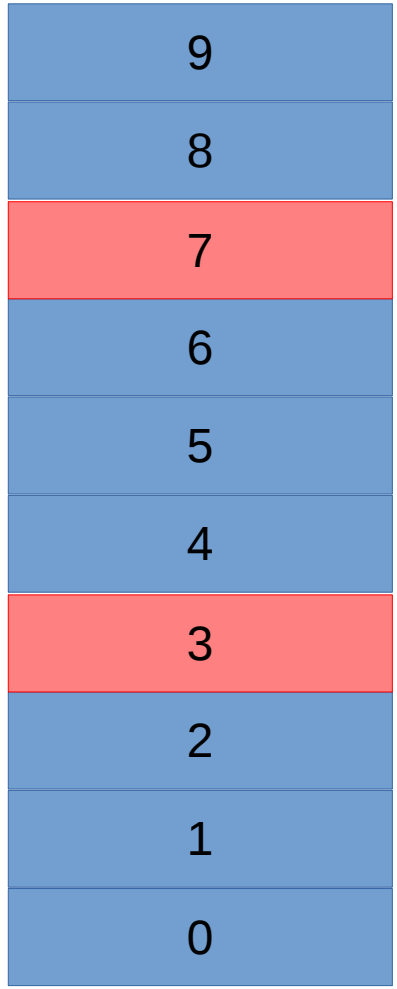
CACHE

MAIN MEMORY

General Idea

- Keep track of current memory blocks in use

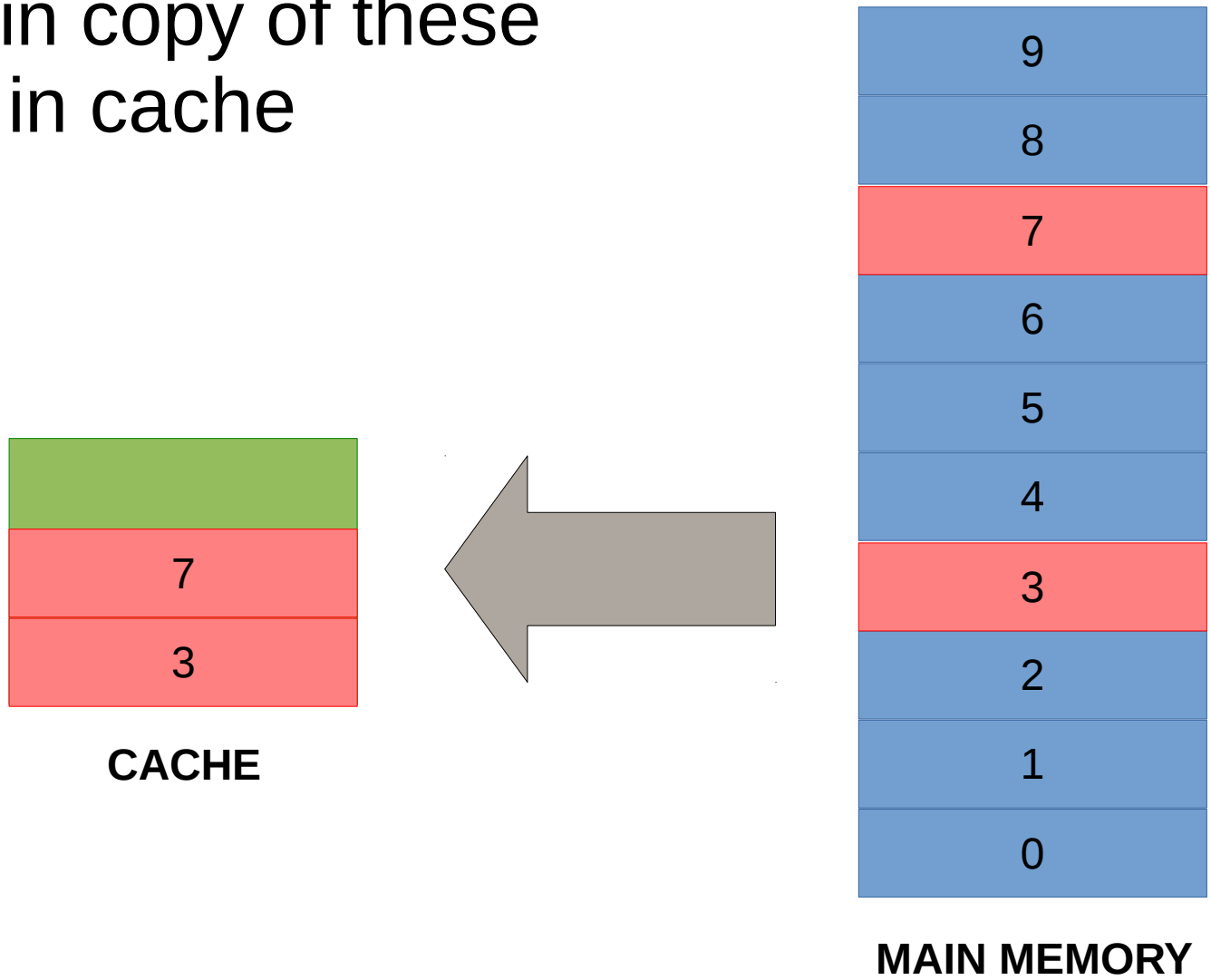
CACHE



MAIN MEMORY

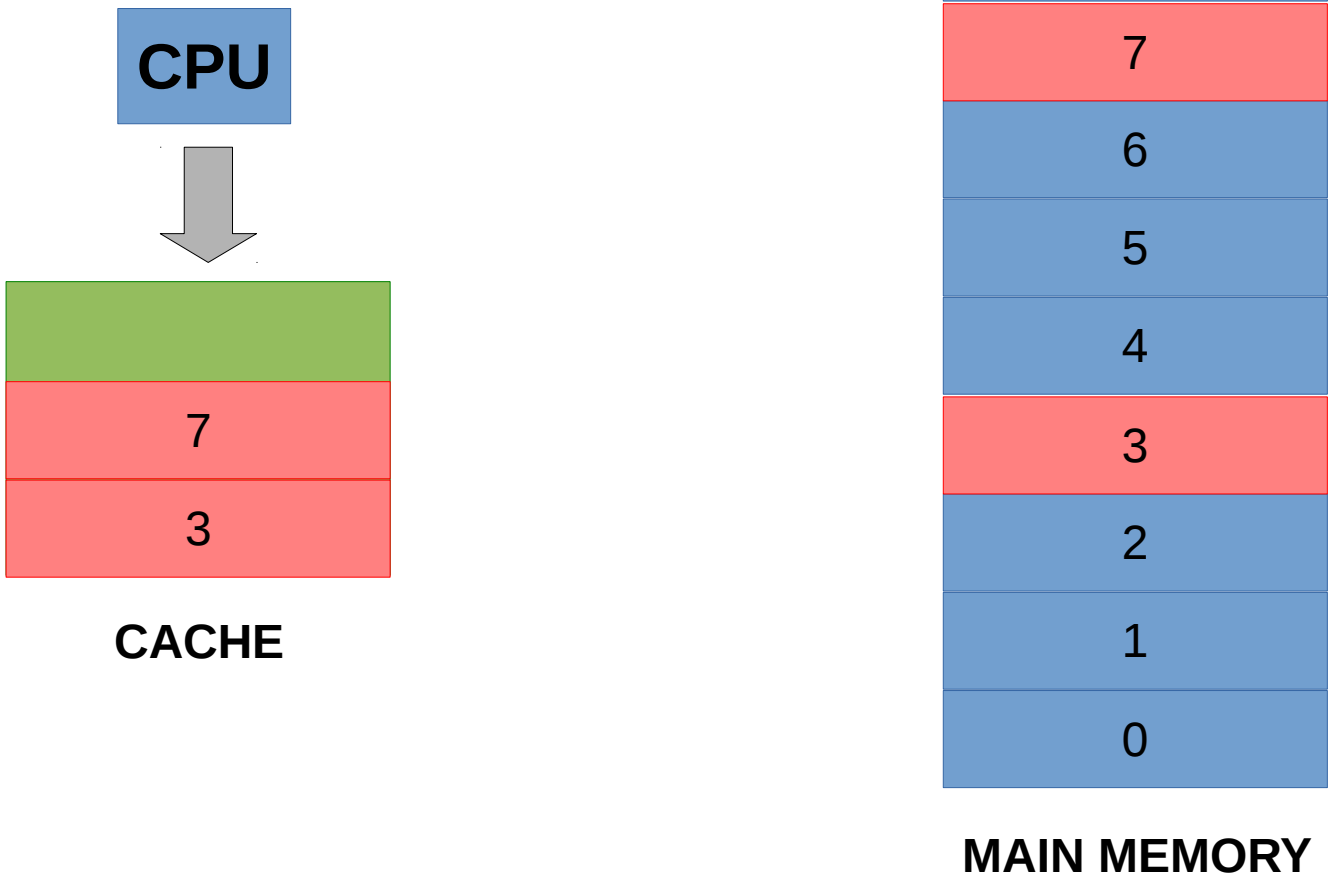
General Idea

- Maintain copy of these blocks in cache



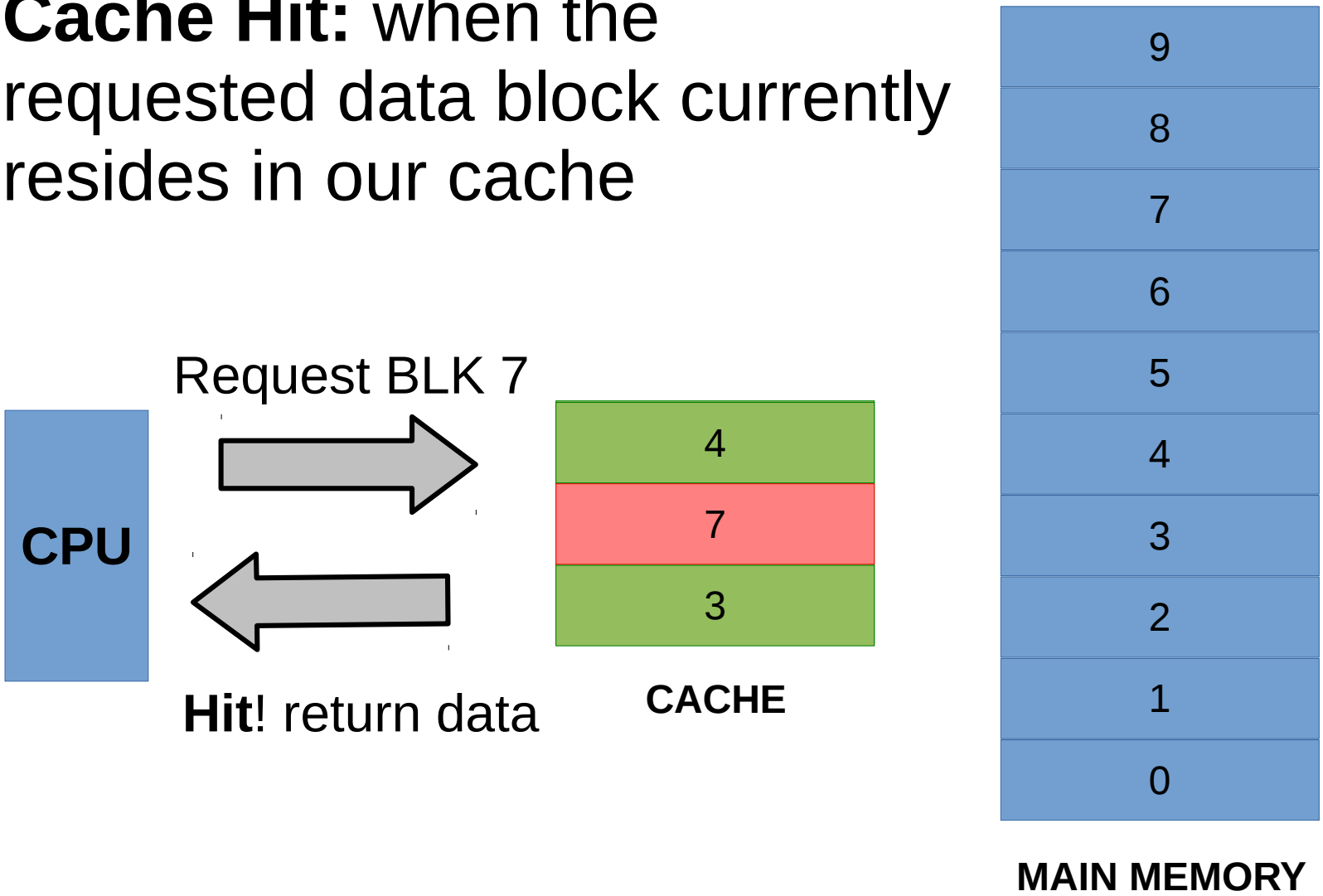
General Idea

- Now, CPU can access data from cache much faster



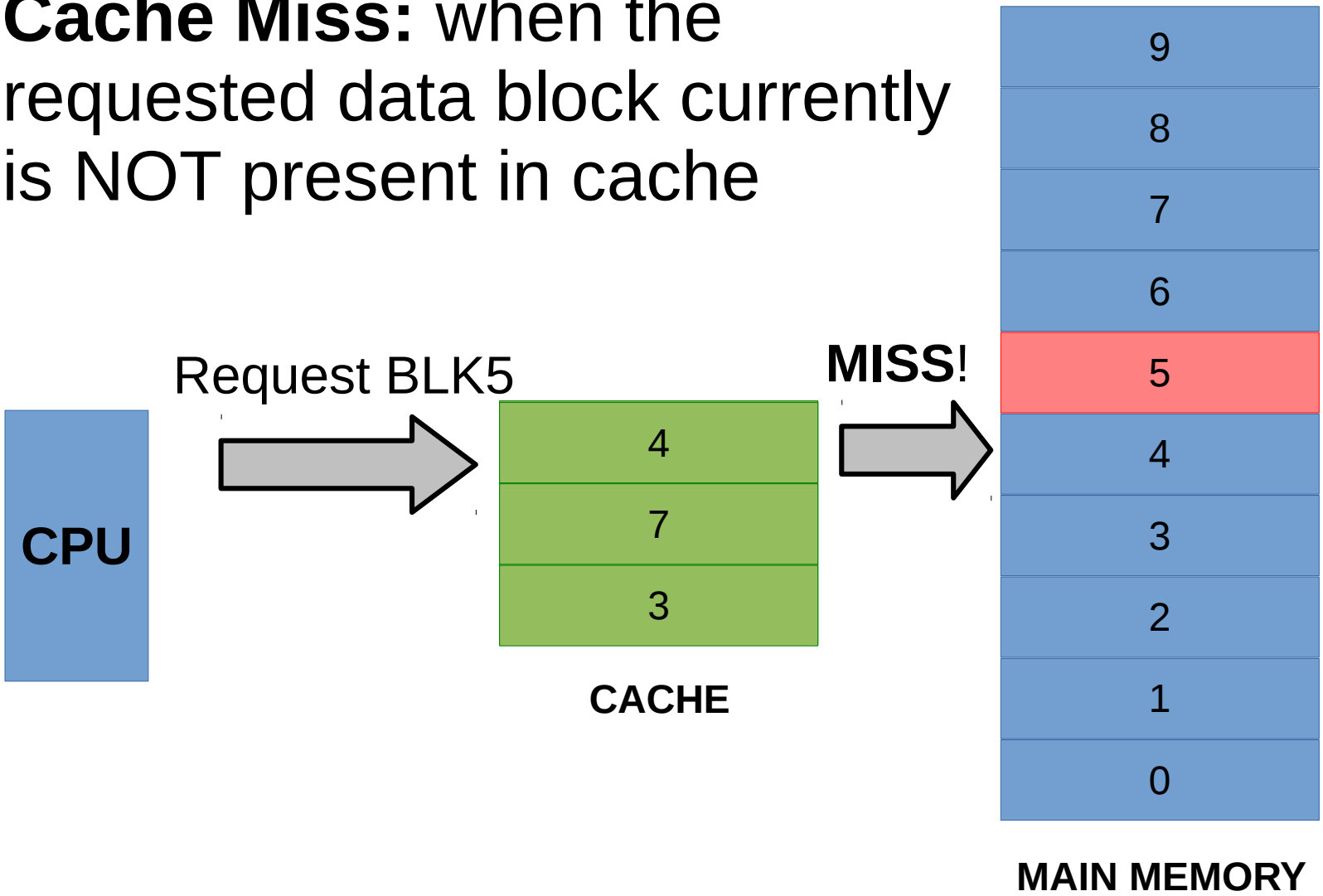
Some Terminology

- **Cache Hit:** when the requested data block currently resides in our cache



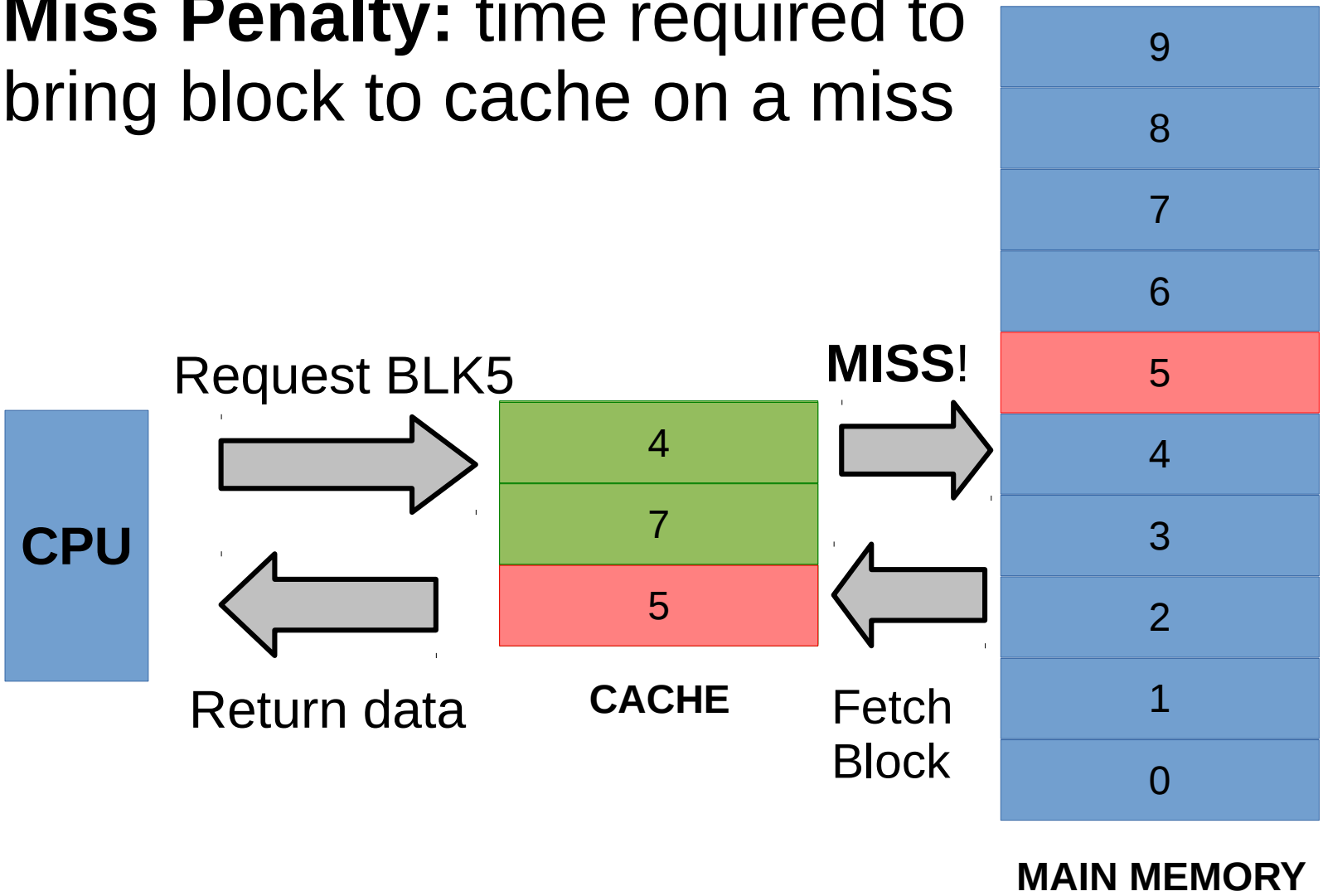
Some Terminology

- **Cache Miss:** when the requested data block currently is NOT present in cache



Some Terminology

- **Miss Penalty:** time required to bring block to cache on a miss



Some Terminology

- **Miss Rate:** (# of cache misses) / (total # of memory accesses)
  - 0 < Miss Rate <= 1
- **Hit Rate**: (# of cache hits) / (total # of memory accesses)
  - Hit Rate = 1 – Miss Rate
- **Miss Penalty:** cost (in cycles) of going to the next level in hierarchy
- **Hit Time:** time (in cycles) of bringing in cache line to processor

A Quick Example

- How much better is a cache with 97% Hit Rate versus 99% Hit Rate in Average Cycles per access?
  - Given **Hit time** = 1 cycle, **Miss Penalty** = 100 cycles?

A Quick Example

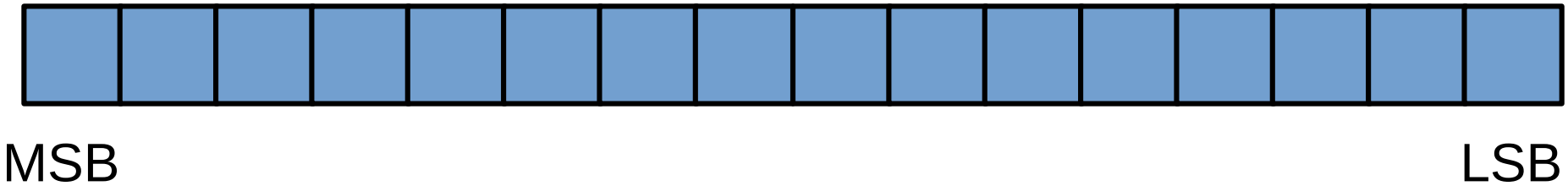
- How much better is a cache with 97% Hit Rate versus 99% Hit Rate in Average Cycles per access?
  - Given **Hit time** = 1 cycle, **Miss Penalty** = 100 cycles?
  - Avg CPA 1 = 1 + 0.03*100 = 1 + 3 = 4
  - Avg CPA 2 = 1 + 0.01*100 = 1 + 1 = 2
- So 99% Hit rate is **2X** as good as 97% Hit rate

So How Does it Work?

- Partition memory into “**blocks**”
  - Each Byte in memory belongs to a block
  - When we need something from Main Memory, we move its entire block
  - When we no longer need data, we evict on granularity of a block
    - i.e. all transactions across Memory bus occur in blocks

Example 1

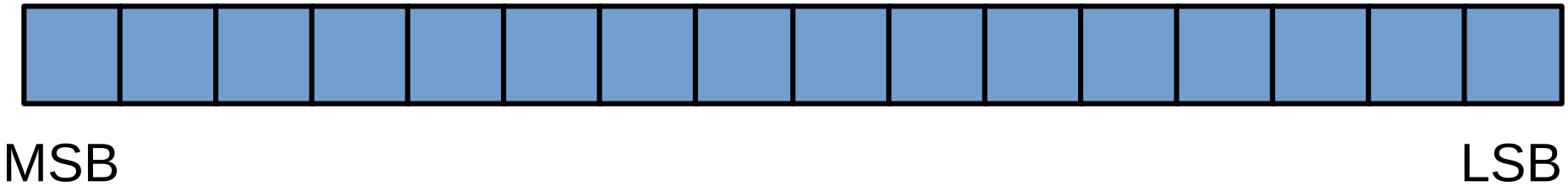
- Consider 16 bit address



- How many addressable bytes are in my address space?

Example 1

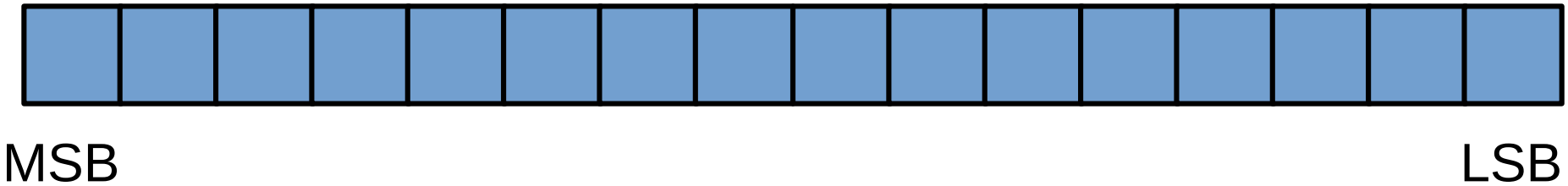
- Consider 16 bit address



- How many addressable bytes are in my address space?
  - **2^16 Bytes = 2^6 * 2^10 B = 64 KB**

Example 1

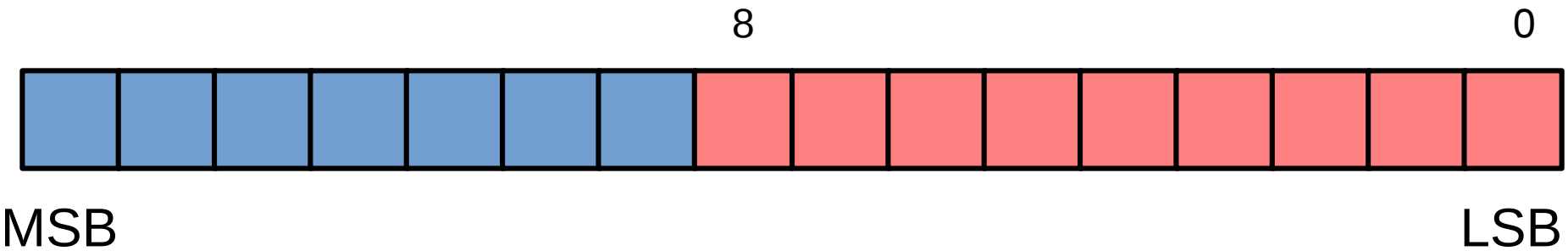
- Consider 16 bit address



- Choose block size B = 512B
- How many bits do we need to refer to a specific Byte in a block?

Example 1

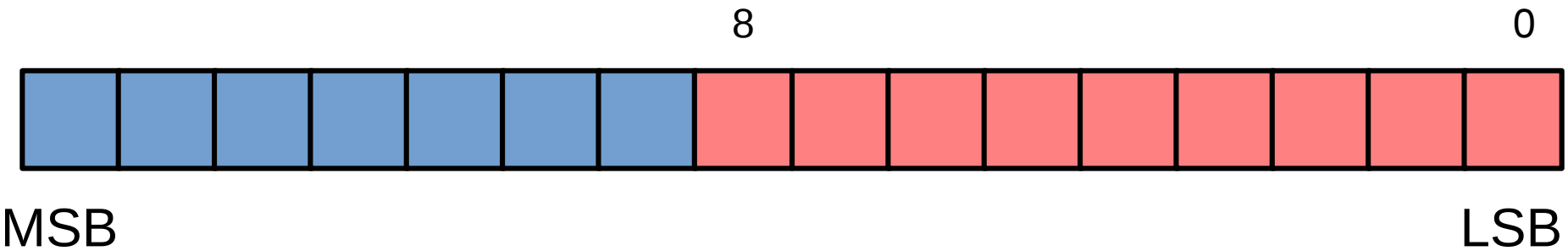
- Consider 16 bit address



- Choose block size B = 512B
- How many bits do we need to refer to a specific Byte in a block?
  - **log(B) = 9 bits**

Example 1

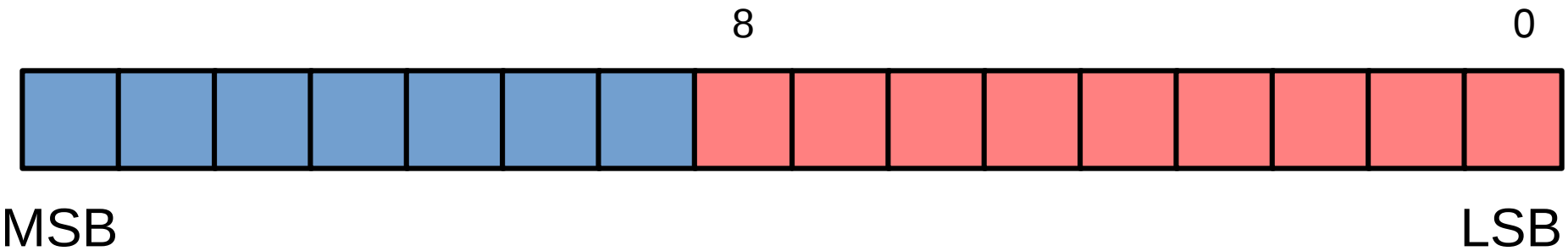
- Consider 16 bit address



- Choose block size B = 512B
- How many bits do we need to refer to a specific Byte in a block?
  - **log(B) = 9 bits**
- How many blocks do we have?

Example 1

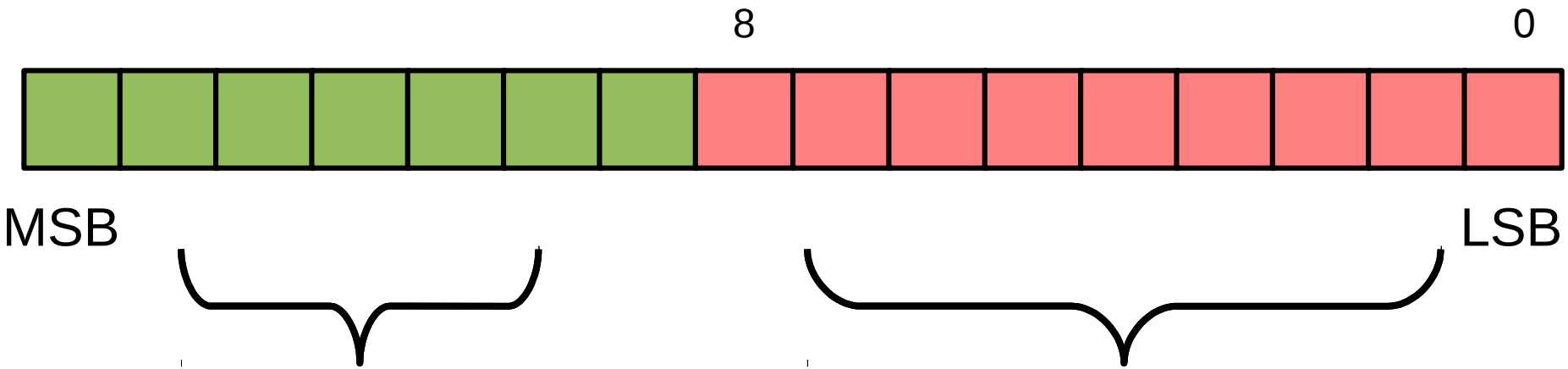
- Consider 16 bit address



- Choose block size **B = 512B**
- Offset = **log(B) = 9 bits**
- How many blocks do we have?
  - # blocks = (Total Memory Space) / (block size)
  - # blocks = (2^16) / (2^9) = 2^7 = **128 blocks**

Example 1

- So we can dissect an address into 2 parts



Block Number
7 bits

Block Offset
9 bits

Example 1B

- Now consider the following instruction (assume 16-bit machine)

```
movl (%bp), $eax
```

- In gdb:

```
(gdb) p $ebp
$1 =  0x3fab
```

Example 1B

```
movl (%bp), $eax
```

- What is the block number?
- Block offset?
- %bp = 0x3fab = 0011 1111 1010 1011

Example 1B

```
movl (%bp), $eax
```

- What is the block number?
- Block offset?
- %bp = 0x3fab = **0011 1111 1010 1011**

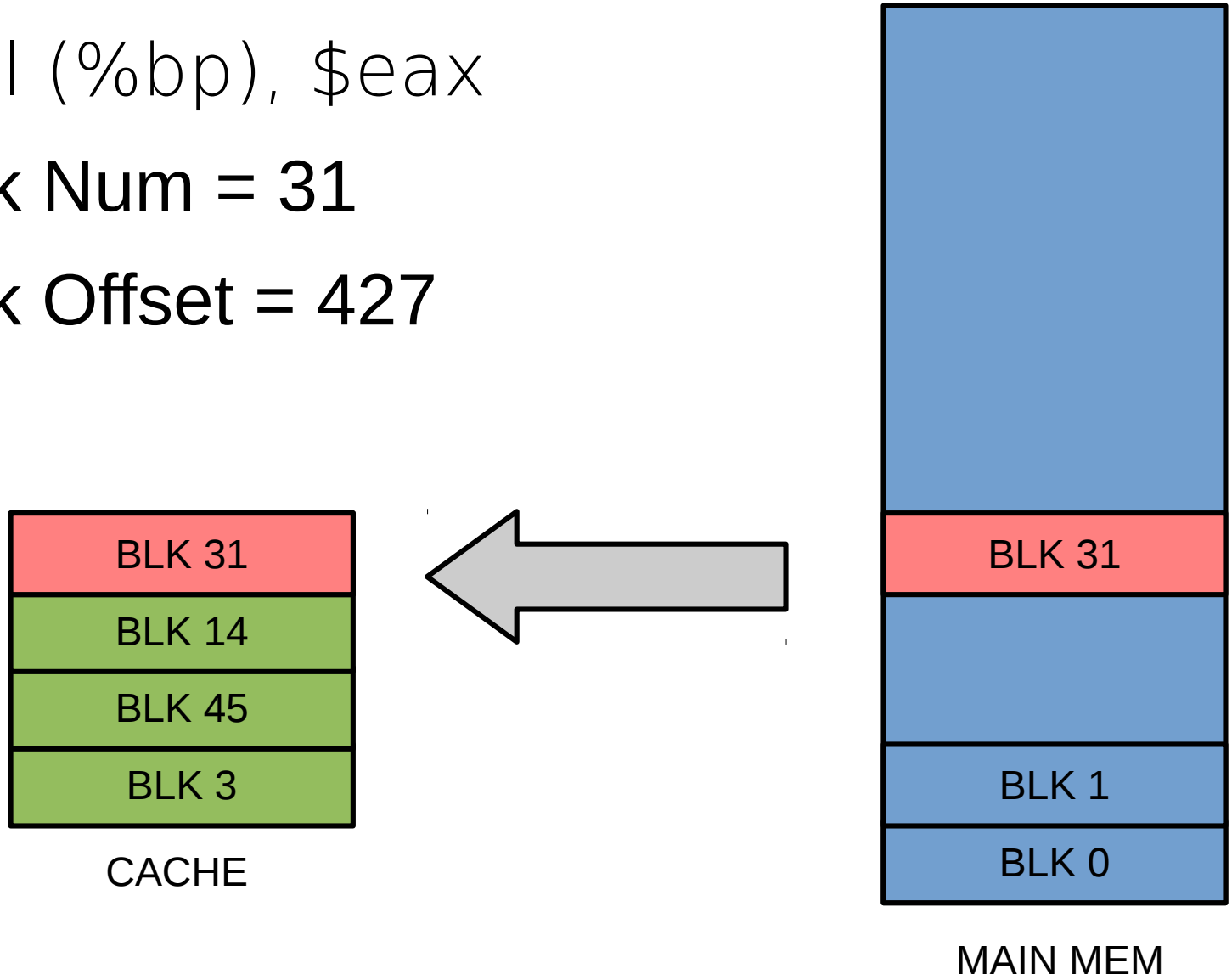
Block Num	**Block Offset**

- Block Num = 0011111 = 0x1f = 31
- Block Offset = 110101011 = 0x1ab = 427

Example 1B

```
movl (%bp), $eax
```

- Block Num = 31
- Block Offset = 427



A Simple Cache

- Keep a copy of data blocks that are in use from Main Memory (DRAM) in Cache (SRAM)
- When we need a data block that is not in Cache (a Cache miss), move block into Cache
- When we want to move a Block into Cache when the Cache is full, **evict** an old block

A Simple Cache

i.e. the **Block number**

one **Cache Line**

V	tag	data
0		
0		
0		
0		
0		
0		
0		

Valid bit

A Simple Cache: Example

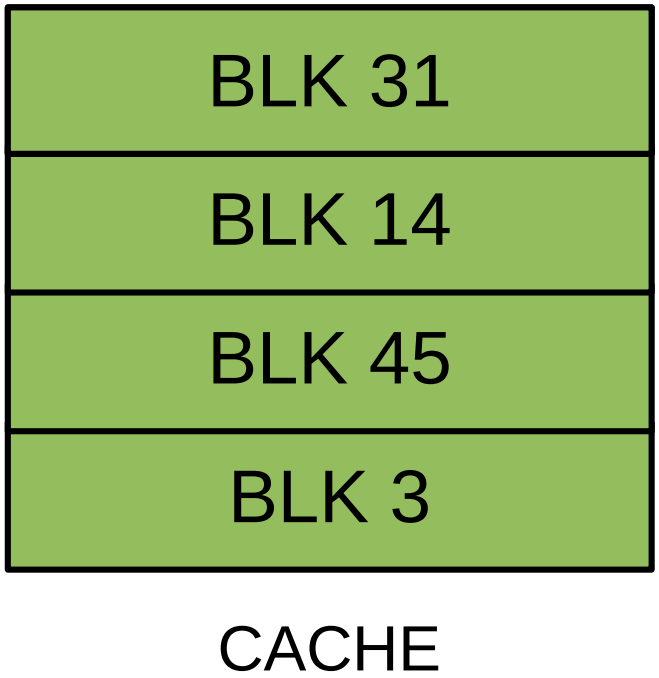
- For each cache line we must store:
- Block data (B bytes) + Tag ($\log_2 B$ bits) + 1 valid bit
- Valid bit = 1 indicates that cache line contains a valid block
- **Cache replacement policy:** When cache is full, need to decide who to evict
  - LRU, MRU, LFU, etc

Fully Associative Cache

- What we've defined is called a **Fully Associative Cache**
- Def: Any memory **Block** can map to any **Cache Line**
  - Minimizes **thrashing**
  - High hardware complexity of eviction policy

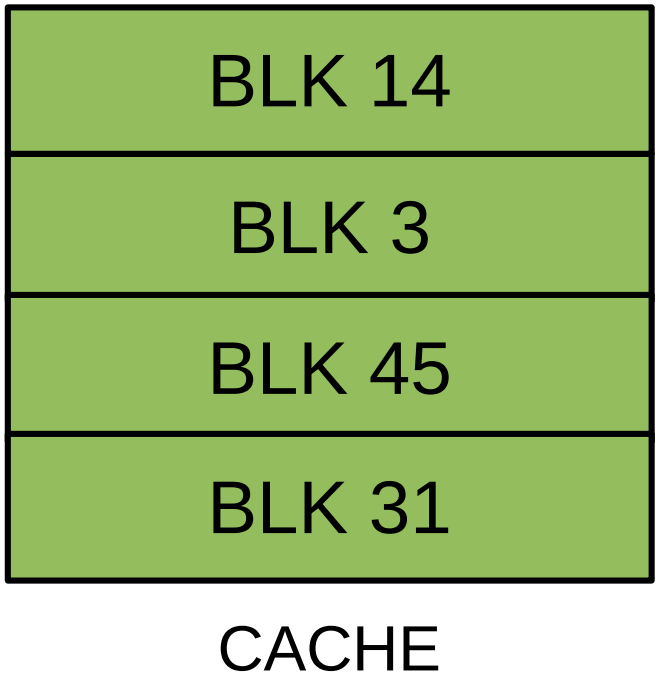
Fully Associative Cache

Fully Associative means this:



Fully Associative Cache

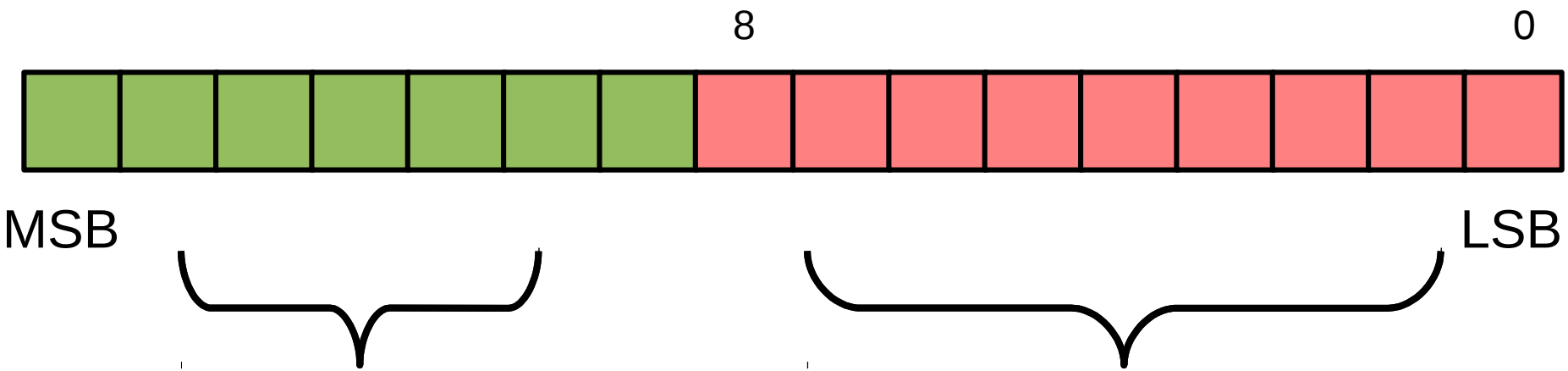
Is the same as this:



Direct Mapped Cache

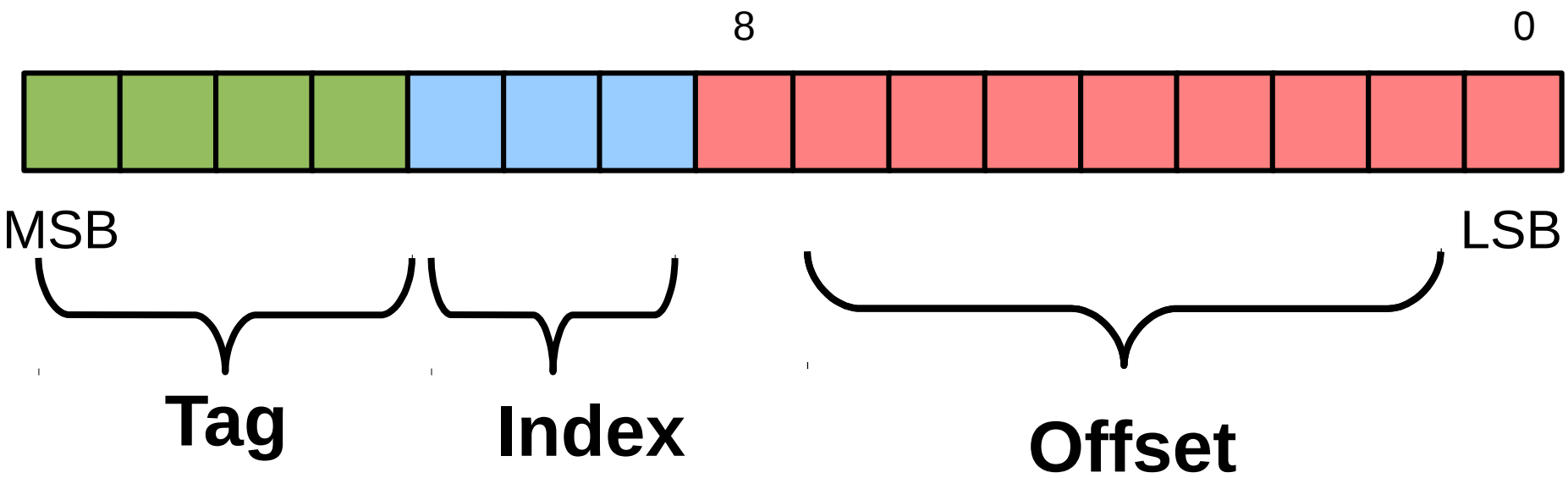
- **Direct Mapped Cache**: every memory Block is mapped to one **cache line** where it is allowed to reside
- Possible for uneccessary **thrashing** => under-utilization of cache => degrades performance
- Cache lookup/eviction is simple

Recall: Address Disection for Fully Associative Cache



(Tag) Block Number Block **Offset**

Address Dissection for Direct Mapped Caches



- **Index**: indicates what cache line each block maps to
- sizeof(Index) = **$\log_2$(# of sets)**

Direct Mapped Cache

INDEX

| index | = 3 bits

	V	tag	data
000			
001	0		
010	0		
011	0		
100	0		
101	0		
110	0		
111	0		

Cache Me If You Can

- Following our example, addresses 1011**001**0000010101 and 0110**001**001110001 map to the same index
- Only one such block can exist in cache at one time
  - What if we frequently alternate access to the two blocks?
  - Approaches **Miss Rate = 1**, so no point of caching at all
  - This is called **thrashing**

So is there a compromise?

Two-Way Set Associative Cache

- For each set, 2 blocks can reside in memory
- Within a set, blocks can map anywhere (associative)
- From previous example, block # 10110**01** and 01100**01** can coexist in cache

Two-Way Set Associative Cache

INDEX

| index | = 2 bits

	v	tag	data
00	0		
01	1	10110	1010100.....00
	1	01100	001001111.....10
10	0		
	0		
11	0		
	0		

one Set

Example 2: Calculating Miss Rate

- Given: block size B = 64B, direct mapped cache (E = 1) with 4 sets (S = 4) **[16 bit addr]**
- How many offset bits?
- How many bits for tag?
- How large is the cache (neglect valid bits)?

Example 2: Calculating Miss Rate

- Given: block size B = 64B, direct mapped cache (E = 1) with 4 sets (S = 4) **[16 bit addr]**
- How many offset bits?

  $= \log_2(B) = \log_2(64) = 6$ bits
- How many bits for tag?

- How large is the cache (neglect valid bits)?

Example 2: Calculating Miss Rate

- Given: block size B = 64B, direct mapped cache (E = 1) with 4 sets (S = 4) **[16 bit addr]**
- How many offset bits?

  = $\log_2$(B) = $\log_2$(64) = 6 bits
- How many bits for tag?

  = 16 - (# offset bits) - (# index bits) = 16 – 6 – 2

  = 8 bits
- How large is the cache (neglect valid bits)?

Example 2: Calculating Miss Rate

- Given: block size B = 64B, direct mapped cache (E = 1) with 4 sets (S = 4) **[16 bit addr]**
- How many offset bits?

  = $\log_2$(B) = $\log_2$(64) = 6 bits
- How many bits for tag?

  = 16 - (# offset bits) - (# index bits) = 16 – 6 – 2

  = 8 bits
- How large is the cache (neglect valid bits)?

  = (# sets) * (associativity) * sizeof(cache line)

  = 4 * 1 * (8 + 64B) = 2080 bits = 260B

Example 2: Calculating Miss Rate

Misses = 0 # Hits = 0

- Access address: 0xffa0

	v	tag	data
00	0	1e	[64B block]
01	1	ab	...
10	0	c0	...
11	0	32	...

Example 2: Calculating Miss Rate

Misses = 0 # Hits = 0

- Access address: 0xffa0

	v	tag	data
00	0	1e	[64B block]
01	1	ab	...
10	0	c0	...
11	0	32	...

TAG = 0xff
INDEX = 10
OS = 100000

Example 2: Calculating Miss Rate

Misses = **1** # Hits = 0

- Access address: 0xffa0

	v	tag	data
00	0	1e	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

TAG = 0xff
INDEX = 10
OS = 100000

Example 2: Calculating Miss Rate

Misses = 1 # Hits = 0

- Access address: 0x1e33

	v	tag	data
00	0	1e	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

TAG = ?
INDEX = ?
OS = ?

Example 2: Calculating Miss Rate

Misses = 1 # Hits = 0

- Access address: 0x1e33

	v	tag	data
00	0	1e	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

TAG = 0x1e
INDEX = 00
OS = 110011

Tags match
but valid = 0!

Example 2: Calculating Miss Rate

Misses = **2** # Hits = 0

- Access address: 0x1e33

	v	tag	data
00	1	1e	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

TAG = 0x1e
INDEX = 00
OS = 110011

Example 2: Calculating Miss Rate

Misses = 2 # Hits = 0

- Access address: 0xff8c

	v	tag	data
00	1	1e	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

TAG = ?
INDEX = ?
OS = ?

Example 2: Calculating Miss Rate

Misses = 2 # Hits = **1**

- Access address: 0xff8c

	v	tag	data
00	1	1e	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

TAG = 0xff
INDEX = 10
OS = 001100

HIT! Tags match AND valid = 1

Example 2: Calculating Miss Rate

Misses = 2 # Hits = 1

- Access address: 0x880f

	v	tag	data
00	1	1e	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

TAG = ?
INDEX = ?
OS = ?

Example 2: Calculating Miss Rate

Misses = 2 # Hits = 1

- Access address: 0x880f

	v	tag	data
00	1	1e	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

TAG = 0x88
INDEX = 00
OS = 001111

Example 2: Calculating Miss Rate

Misses = **3** # Hits = 1

- Access address: 0x880f

	v	tag	data
00	1	88	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

EVICT

TAG = 0x88
INDEX = 00
OS = 001111

Example 2: Calculating Miss Rate

Misses = 3 # Hits = 1

- So our Miss Rate = 3/(3+1) = 0.75 (pretty bad!)

	v	tag	data
00	1	88	[64B block]
01	1	ab	...
10	1	ff	...
11	0	32	...

Modifying Caches

- What happens when we modify the data stored on stack
  - Need to update “stale” blocks in memory
- Two policies
- **Write through:** when change made in cache, immediately issue request to next stage
- **Write back:** when change is made in cache, delay updating next stage until block is <u>evicted</u>

Cache is King

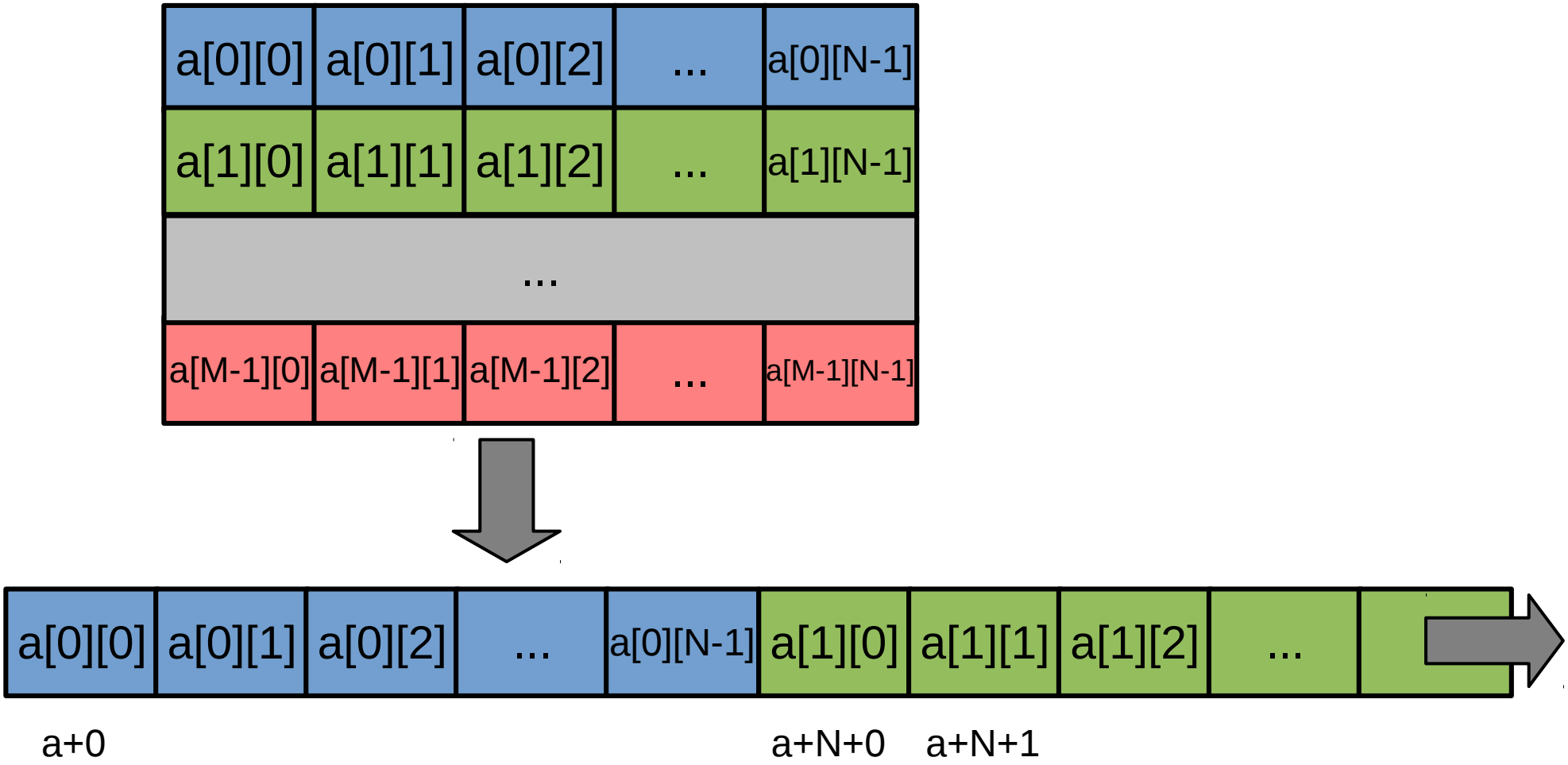
Conclusions

- Latency of Cache read/write << latency of going to Main Memory
- Cache maintenance and integrity is handled by hardware – completely invisible to programmer
- **Then why should we care???**

An example in code

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

Recall Matrix Representation



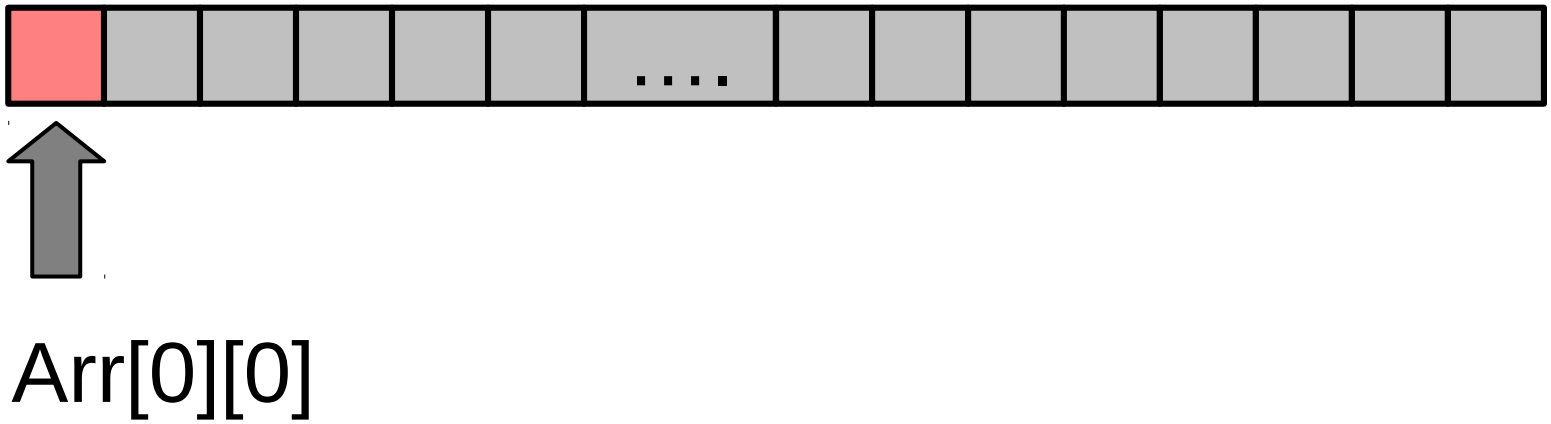
- In general, &(a[i][j]) = a + N*i+j

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

- What is the access pattern for this code?

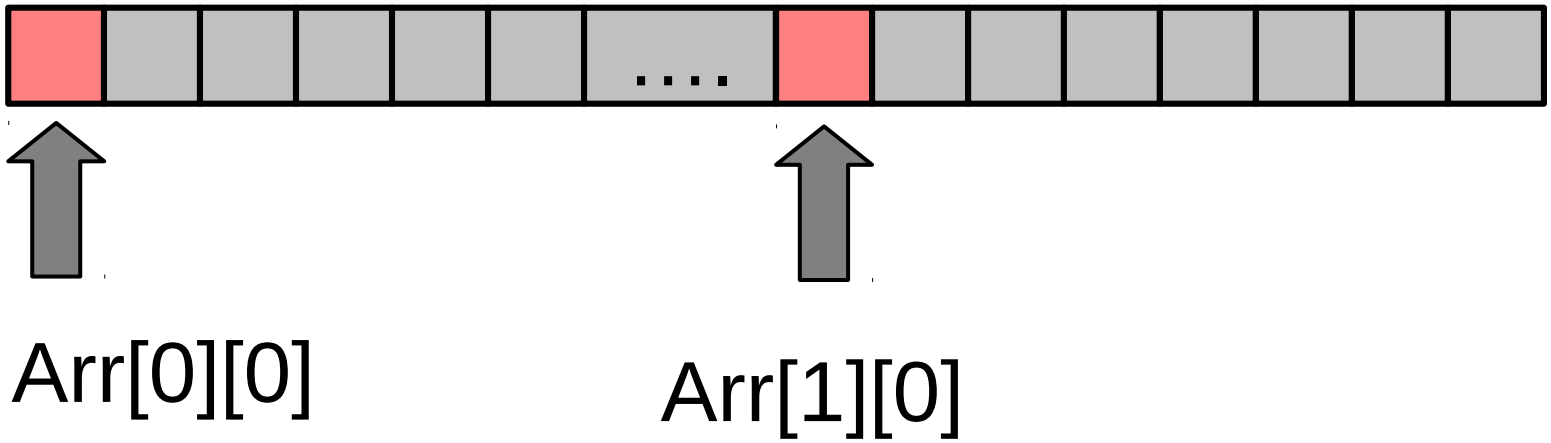
```
for (int j=0; j < N; j++) {
   for (int i=0; i < M; i++) {
      sum += arr[i][j]
   }
}
```

- What is the access pattern for this code?



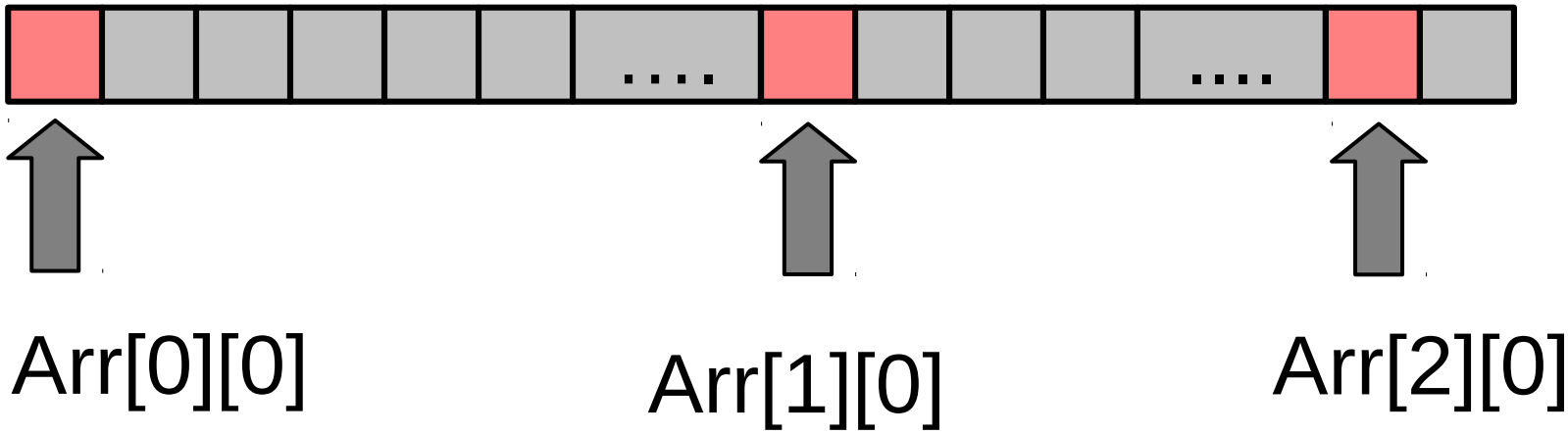
```
for (int j=0; j < N; j++) {
   for (int i=0; i < M; i++) {
      sum += arr[i][j]
   }
}
```

- What is the access pattern for this code?



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

- What is the access pattern for this code?



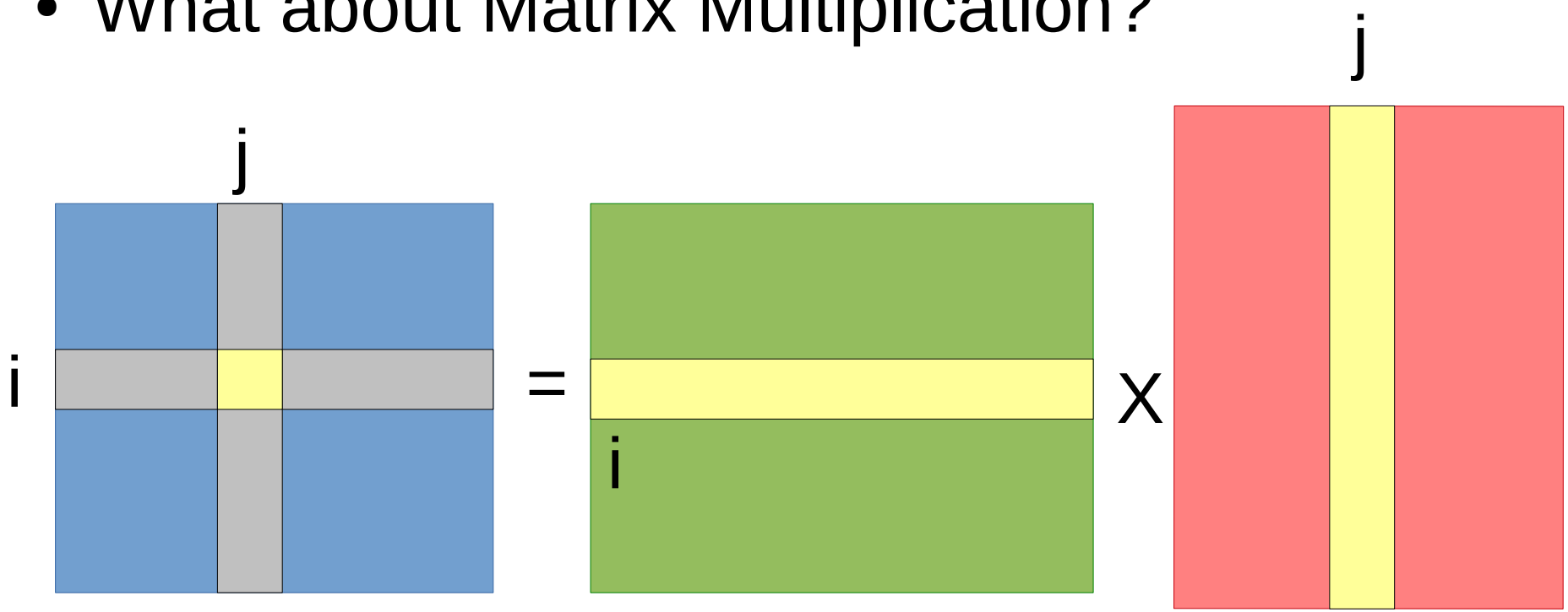
Cache Friendly Version

- If N is super large, access pattern ~ random access
  - i.e. we lose spatial locality
- Re-order loop iterators

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

Cache Blocking

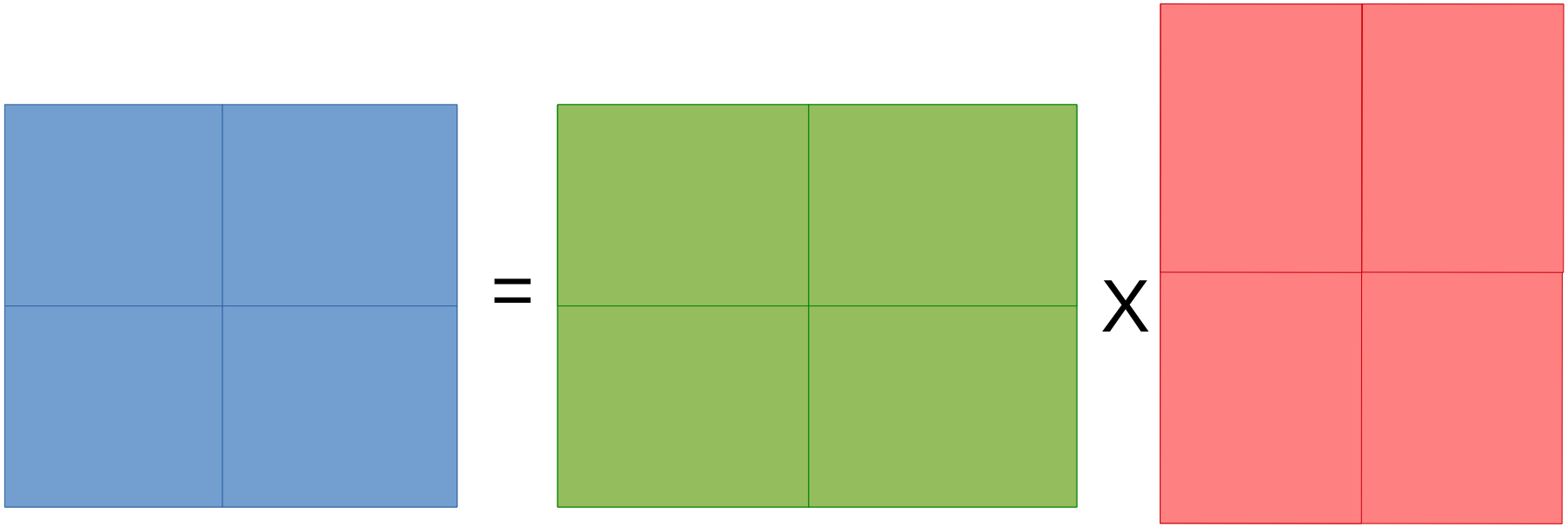
- What about Matrix Multiplication?



- C = A*B
- Alg calls for iterating B in colum major order = **bad** locality

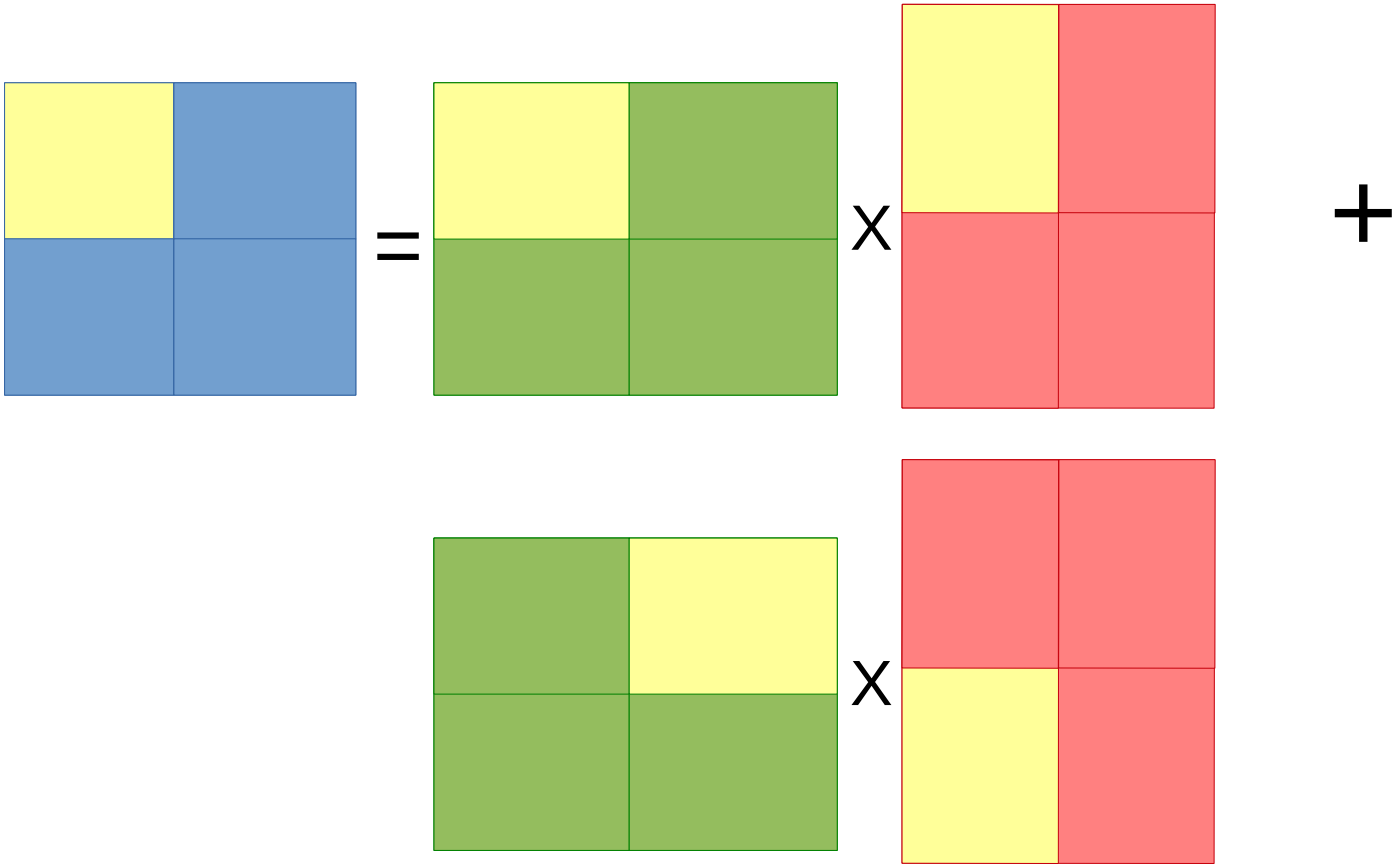
Cache Blocking

Only have to maintain 3 small blocks in Cache



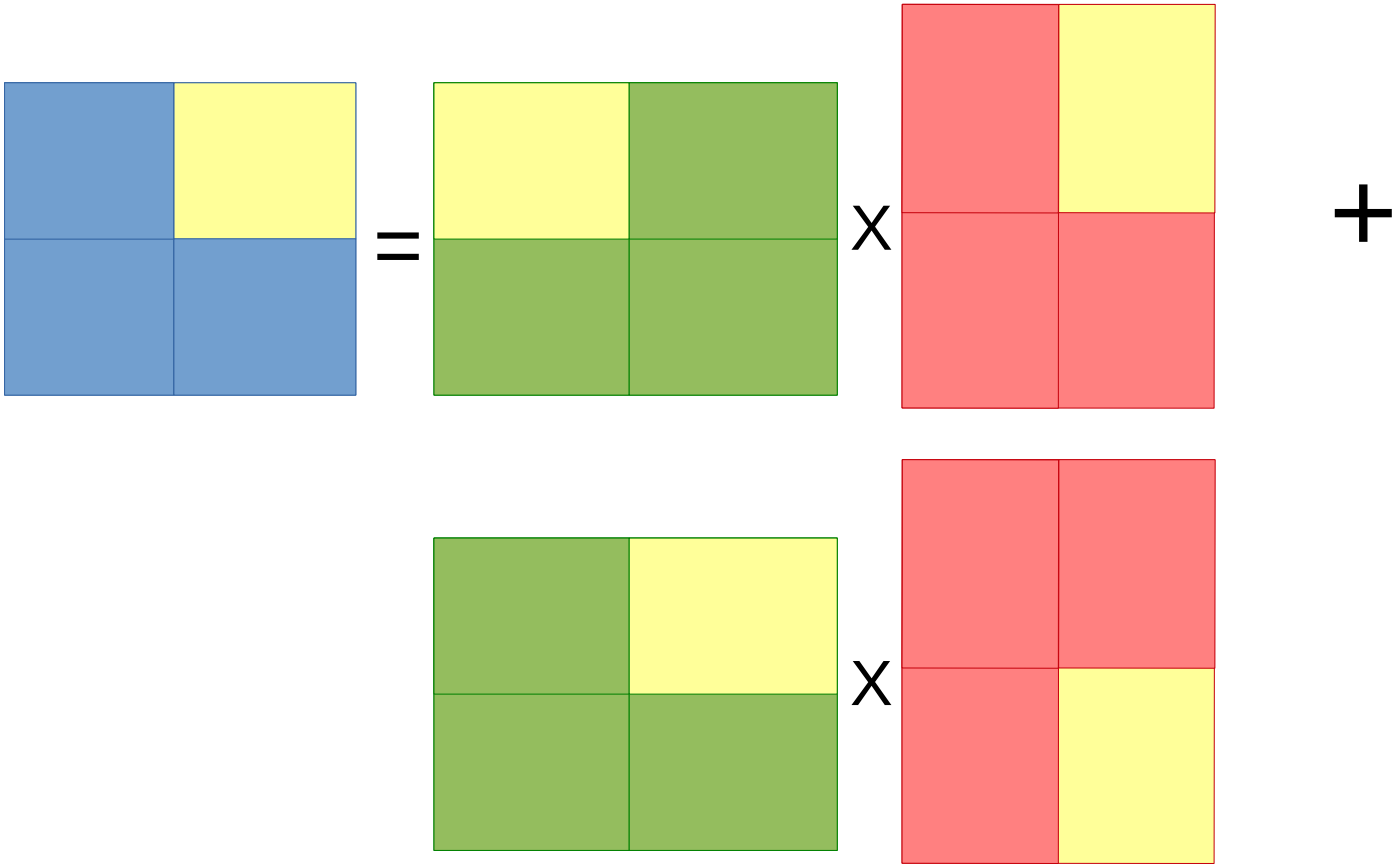
Cache Blocking

Only have to maintain 3 small blocks in Cache



Cache Blocking

Only have to maintain 3 small blocks in Cache



DATA-FLOW GRAPHS

```
leal (%edx, %eax), %ebx
addl %edx, %ebx
addl 1, %edx
cmpl %edx, %edi
jge .L33
```

- What registers do we read from?

```
leal (%edx, %eax), %ebx
addl %edx, %ebx
addl 1, %edx
cmpl %edx, %edi
jge .L33
```

Why not %ebx?

%edx %eax %edi

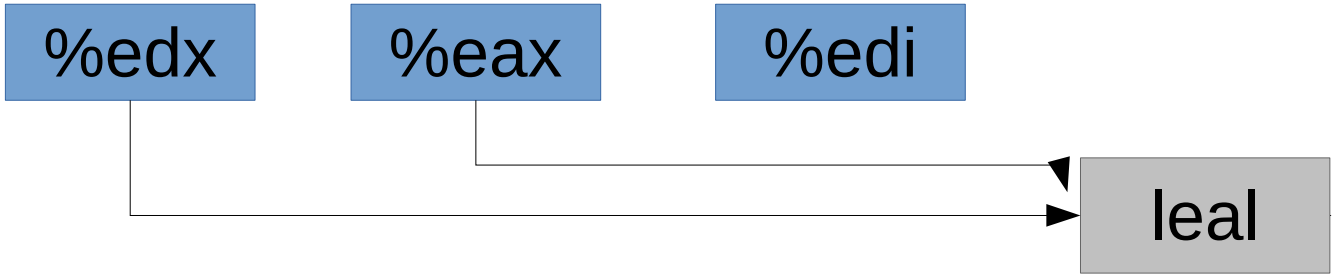
- Which do we write to?

```
leal (%edx, %eax), %ebx
addl %edx, %ebx
addl 1, %edx
cmpl %edx, %edi
jge .L33
```

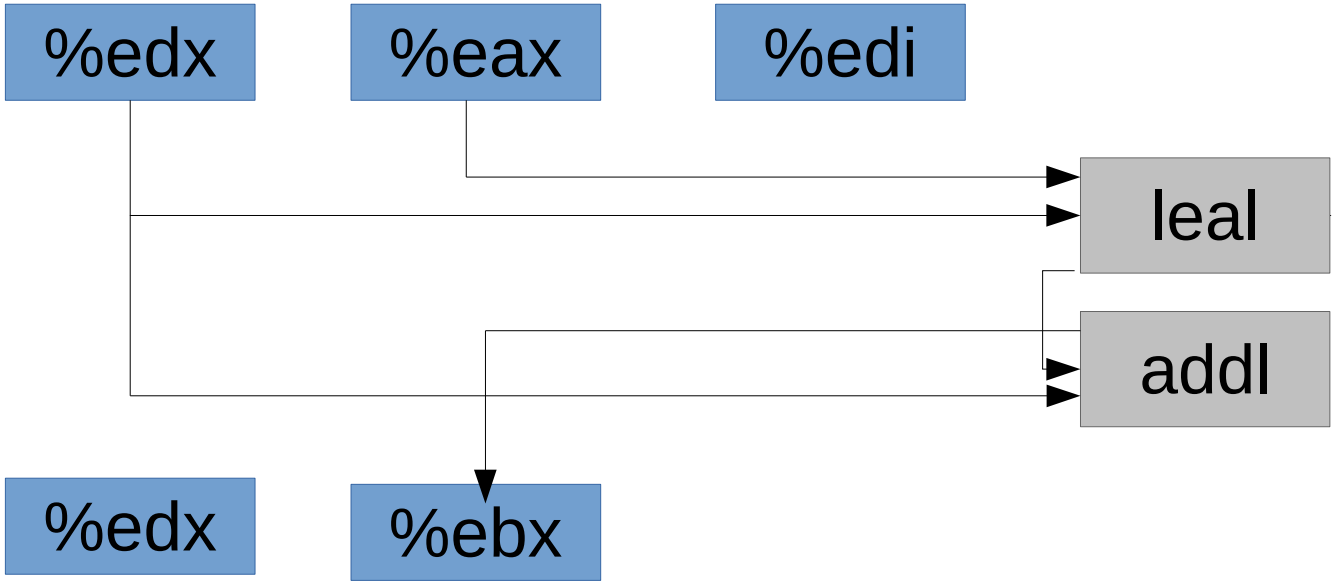
%edx %eax %edi

%edx %ebx

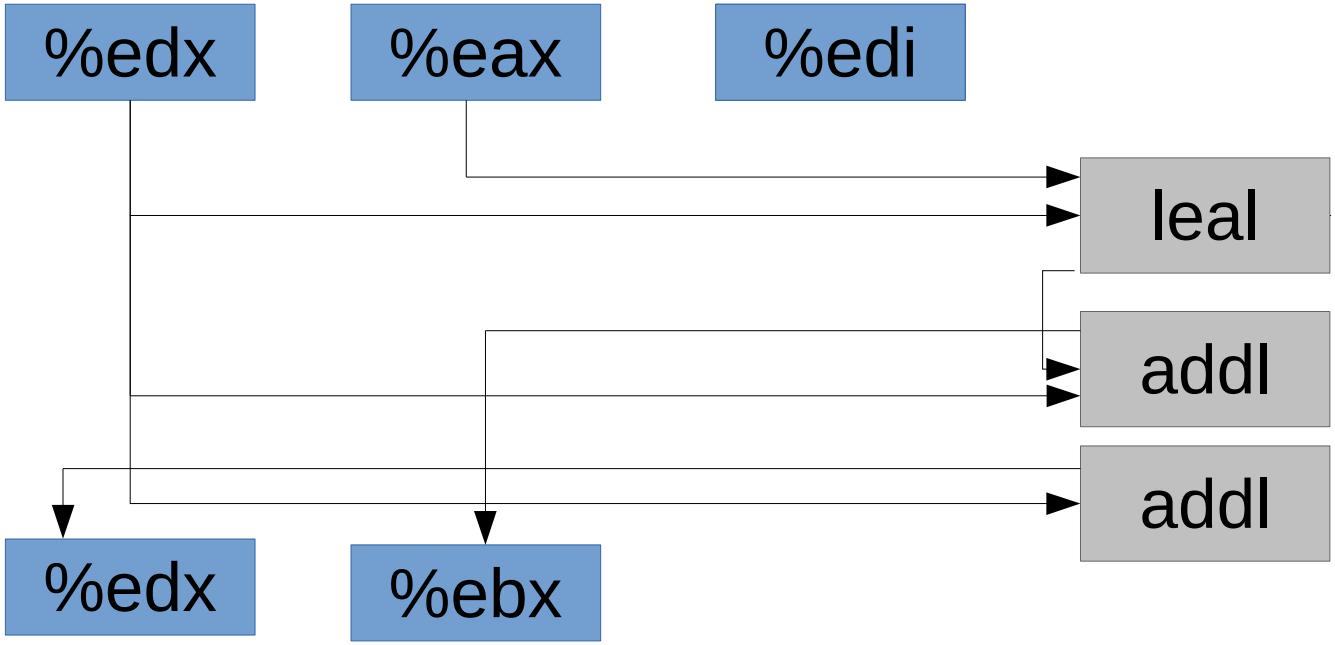
```
leal (%edx, %eax), %ebx
addl %edx, %ebx
addl 1, %edx
cmpl %edx, %edi
jge .L33
```



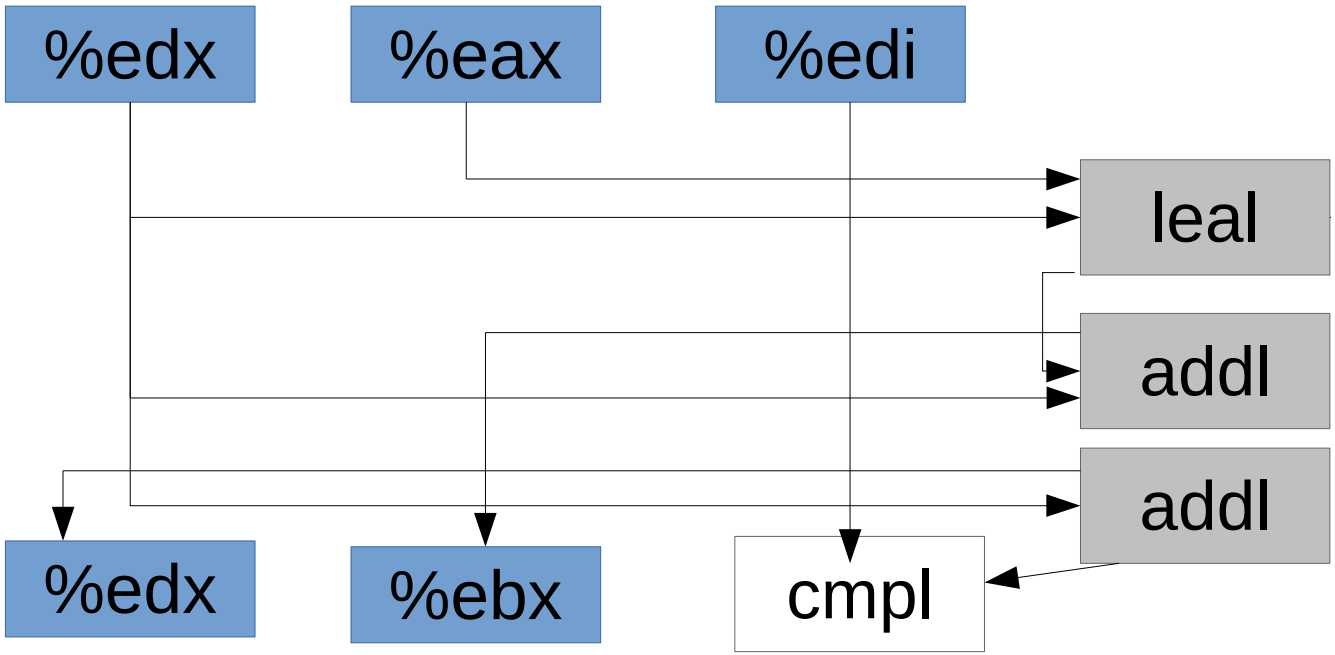
```
leal (%edx, %eax), %ebx
addl %edx, %ebx
addl 1, %edx
cmpl %edx, %edi
jge .L33
```



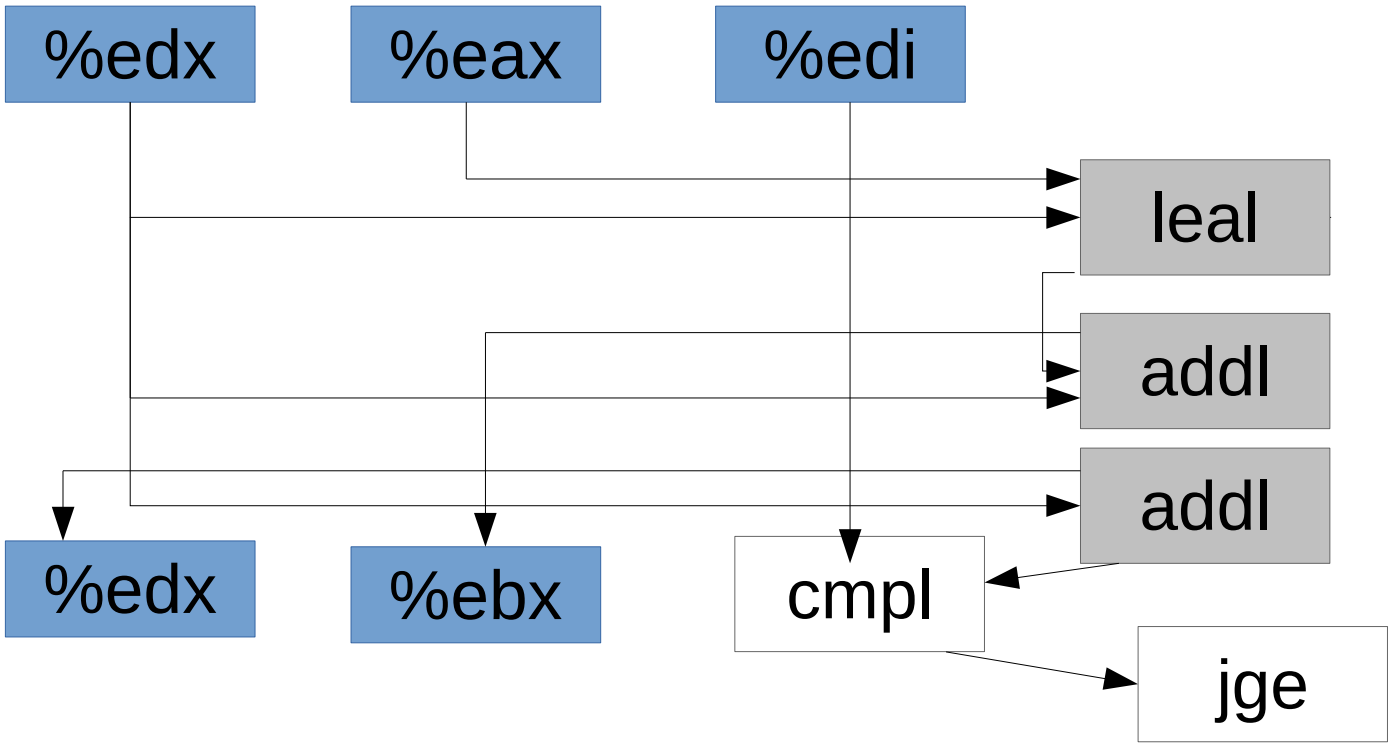
```
leal (%edx, %eax), %ebx
addl %edx, %ebx
addl 1, %edx
cmpl %edx, %edi
jge .L33
```

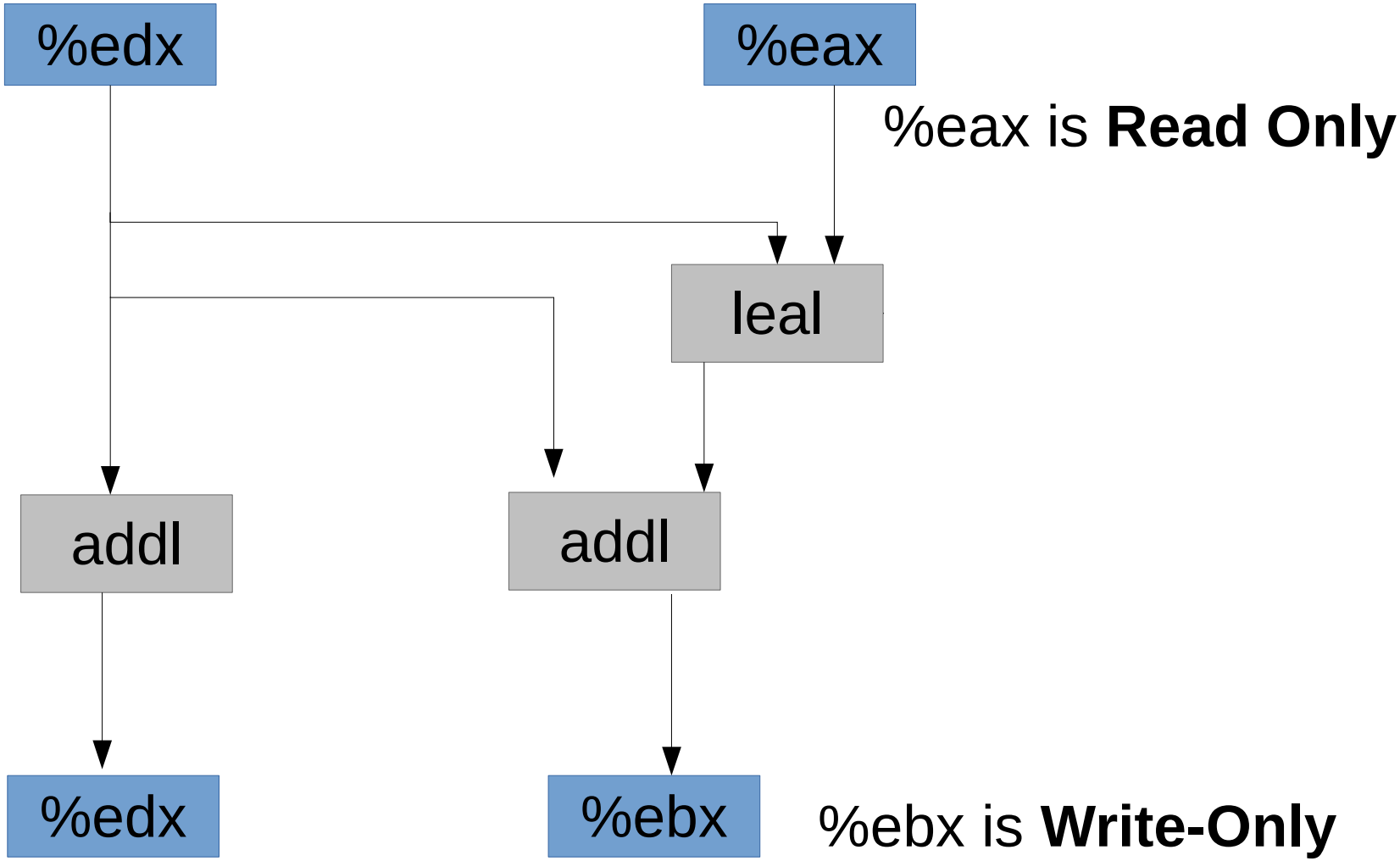


```
leal (%edx, %eax), %ebx
addl %edx, %ebx
addl 1, %edx
cmpl %edx, %edi
jge .L33
```



```
leal (%edx, %eax), %ebx
addl %edx, %ebx
addl 1, %edx
cmpl %edx, %edi
jge .L33
```




%edx
%eax
%eax is **Read Only**
leal
addl
addl
%edx
%ebx
%ebx is **Write-Only**