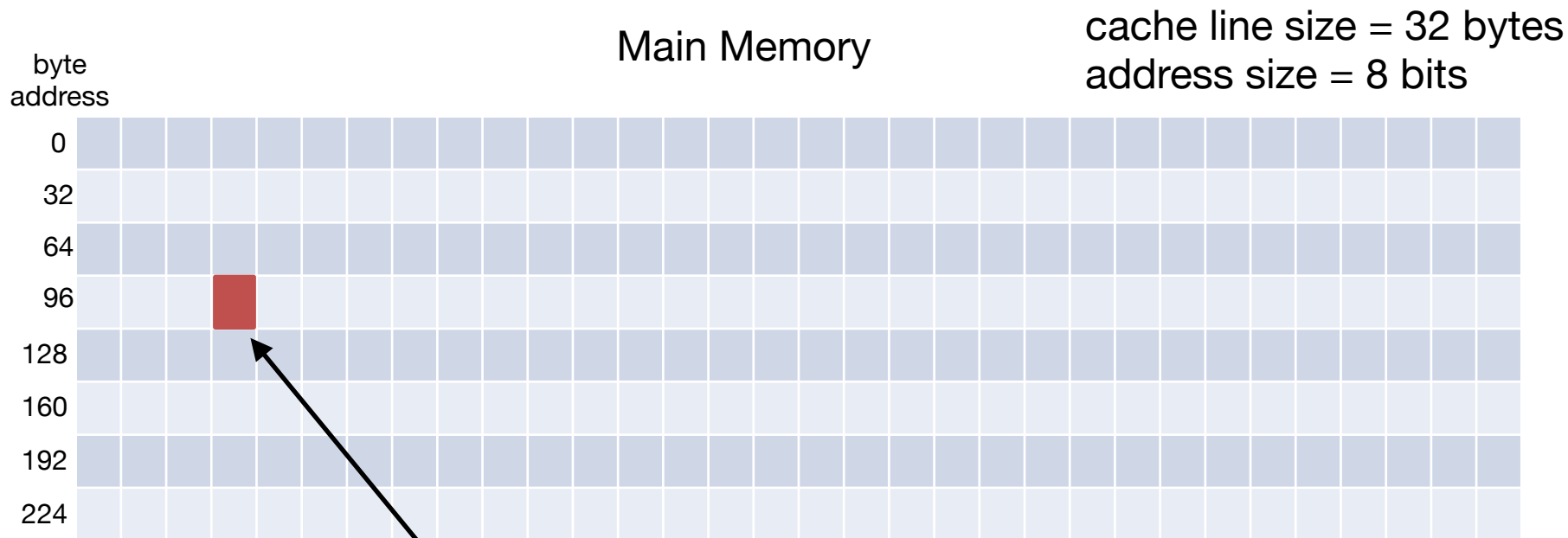


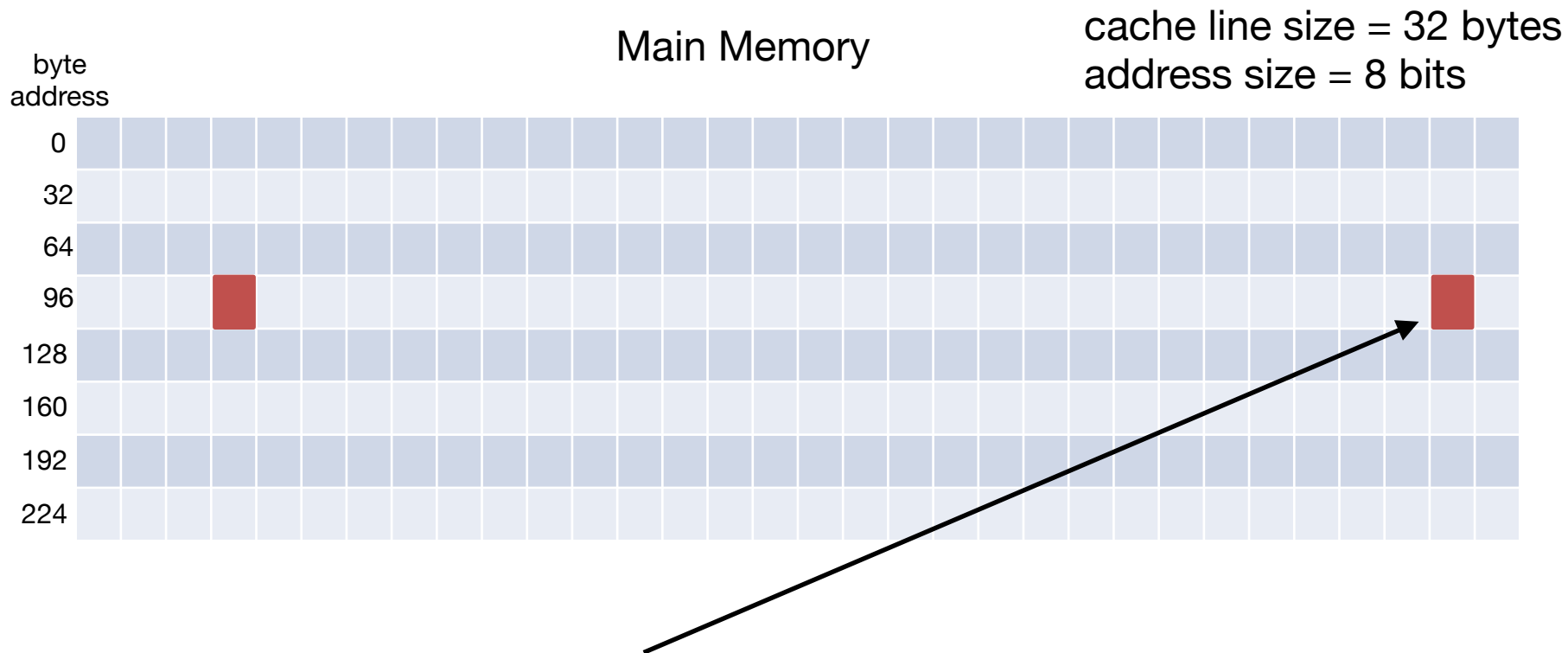
Caches

Review: Cache Lines



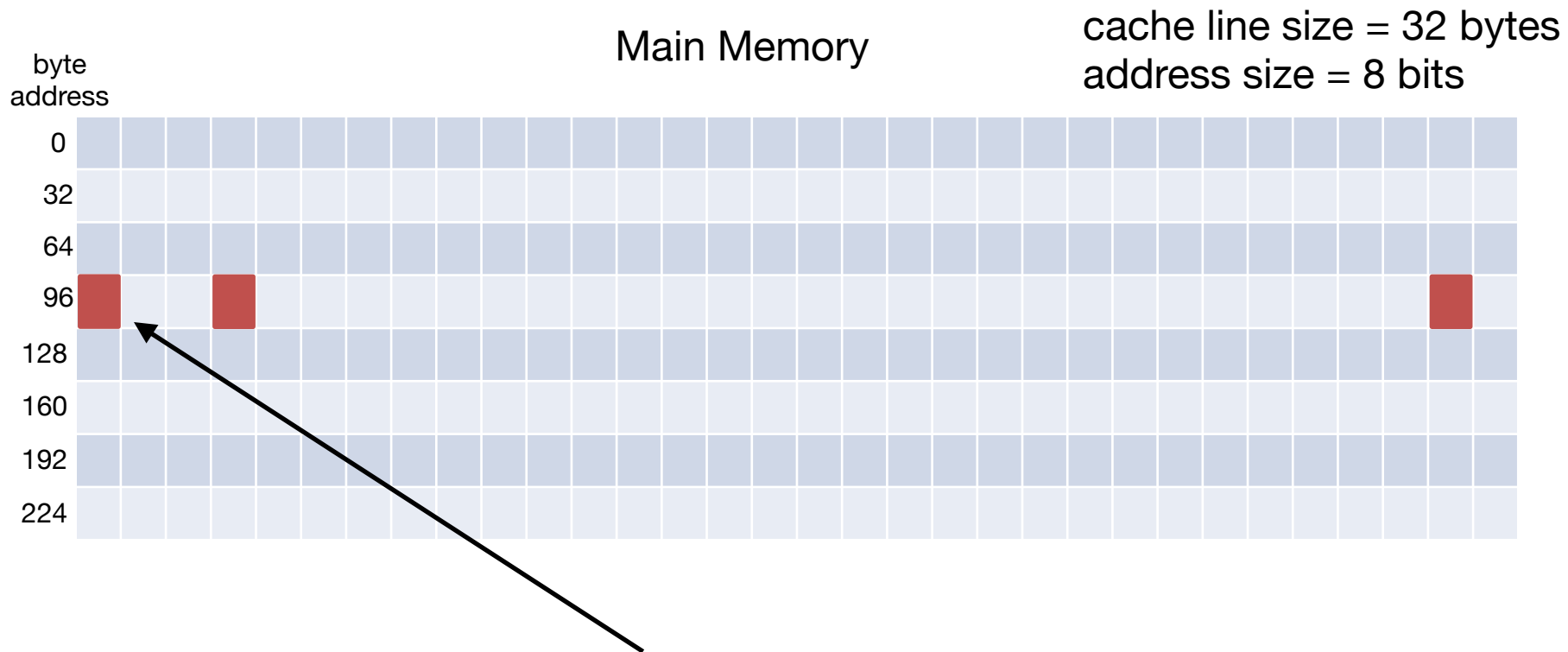
If we want to read byte 99_{10} and its not in the cache, we would bring in bytes 96_{10} - 127_{10} into the cache

Review: Cache Lines



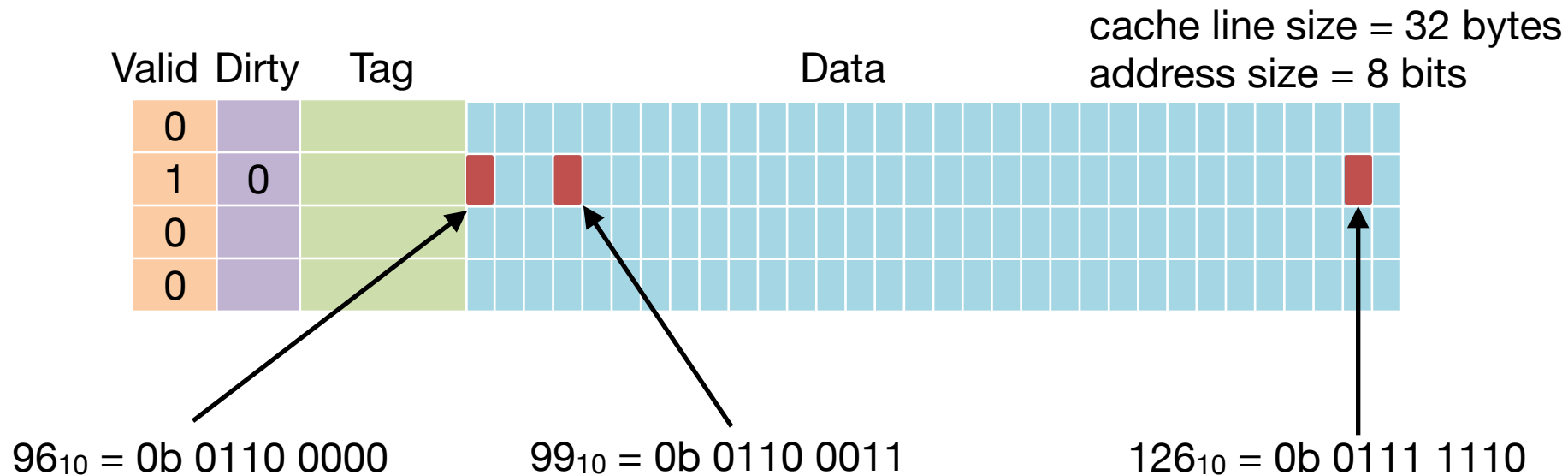
If we then wanted to read byte 126_{10} , we would get a cache hit because we just brought in the line that its in

Review: Cache Lines

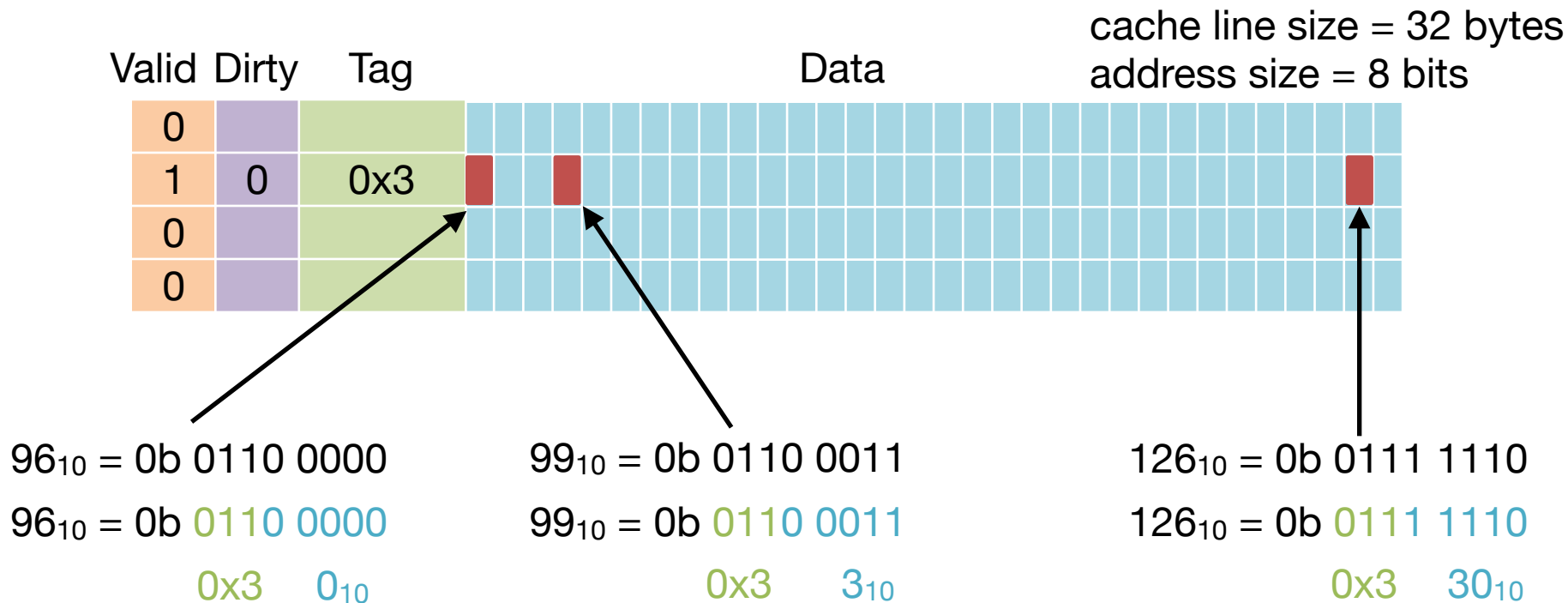


If we then wanted to read byte 96_{10} , we would get a cache hit because we just brought in the line that it's in

Fully Associative Cache Example



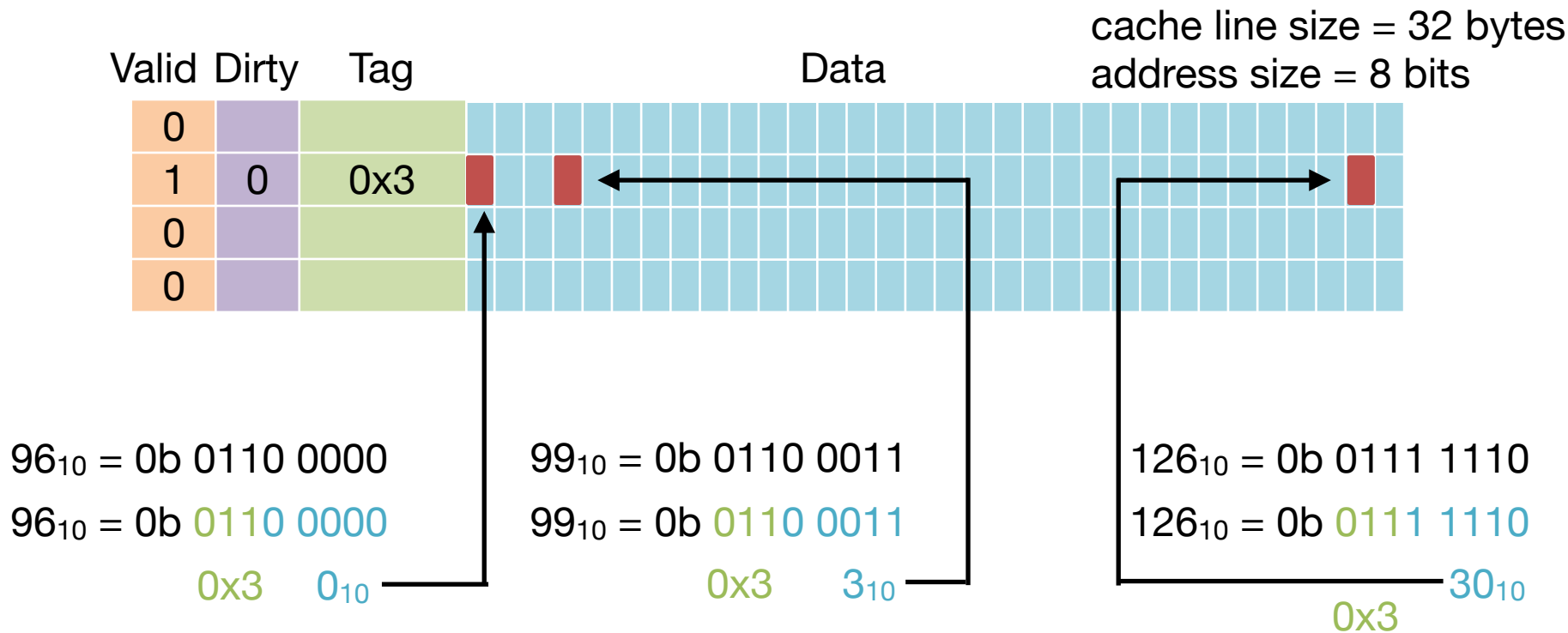
Fully Associative Cache Example



byte offset bits = $\log_2(\text{line size}) = \log_2(32) = 5$

tag bits = # address bits - # offset bits = 3

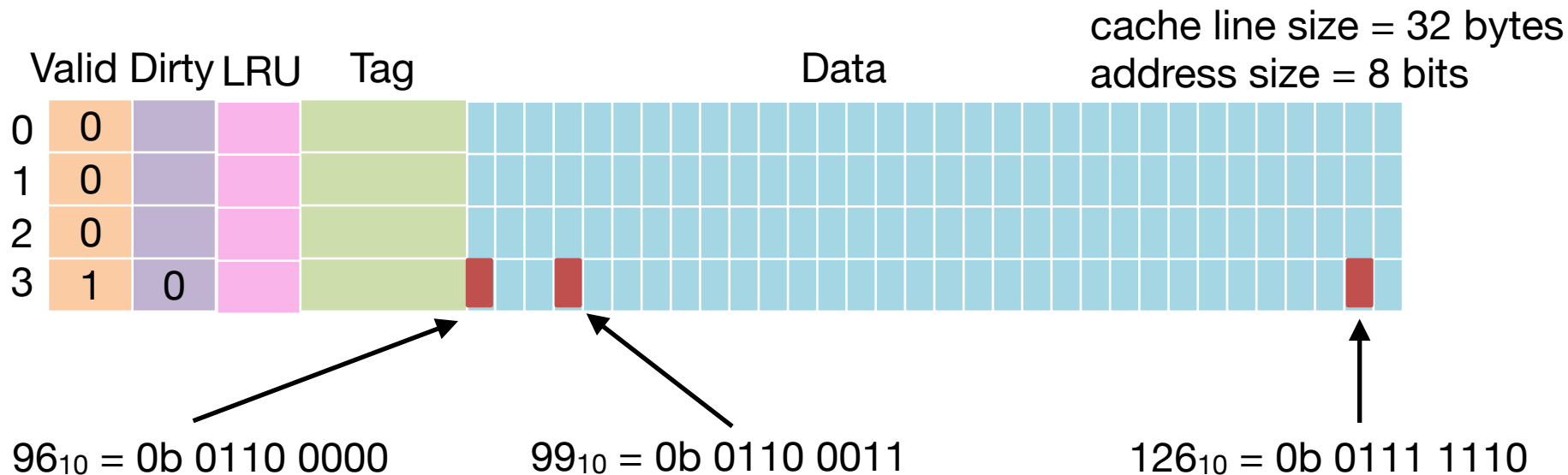
Fully Associative Cache Example



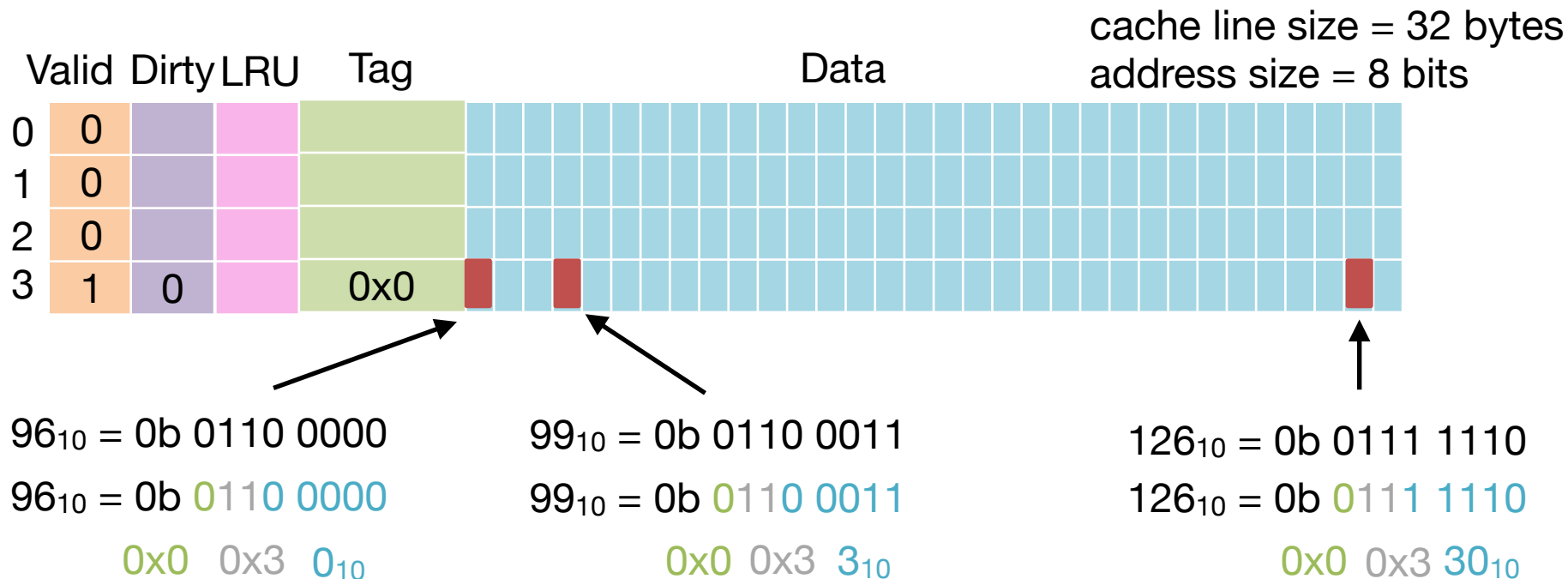
byte offset bits = $\log_2(\text{line size}) = \log_2(32) = 5$

tag bits = # address bits - # offset bits = 3

Direct Mapped Cache Example



Direct Mapped Cache Example

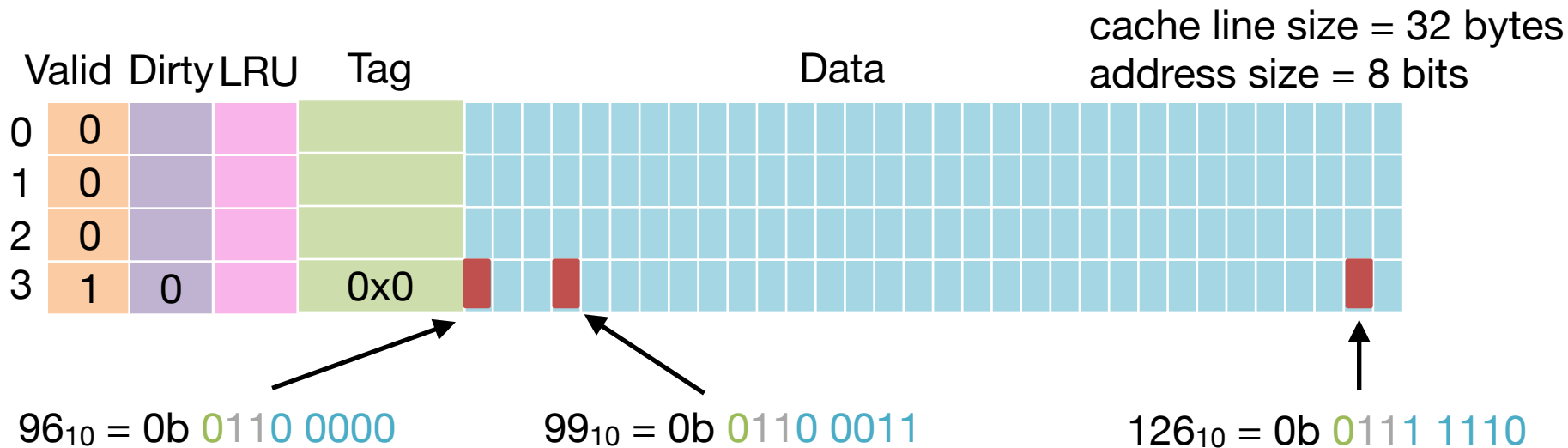


byte offset bits = $\log_2(\text{line size}) = \log_2(32) = 5$

index bits = $\log_2(\text{\# lines}) = \log_2(4) = 2$

tag bits = # address bits - # offset bits = 3

Direct Mapped Cache Example

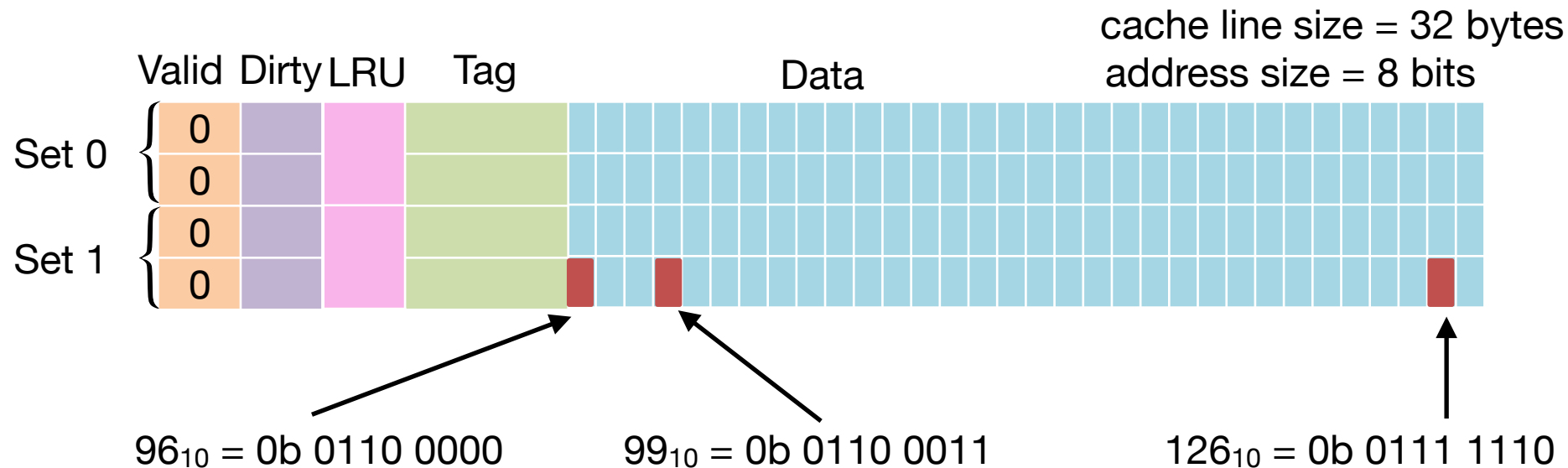


Addresses 0110 0000 through 0110 1111 are part of the same cache line and going to map to the same index in the cache

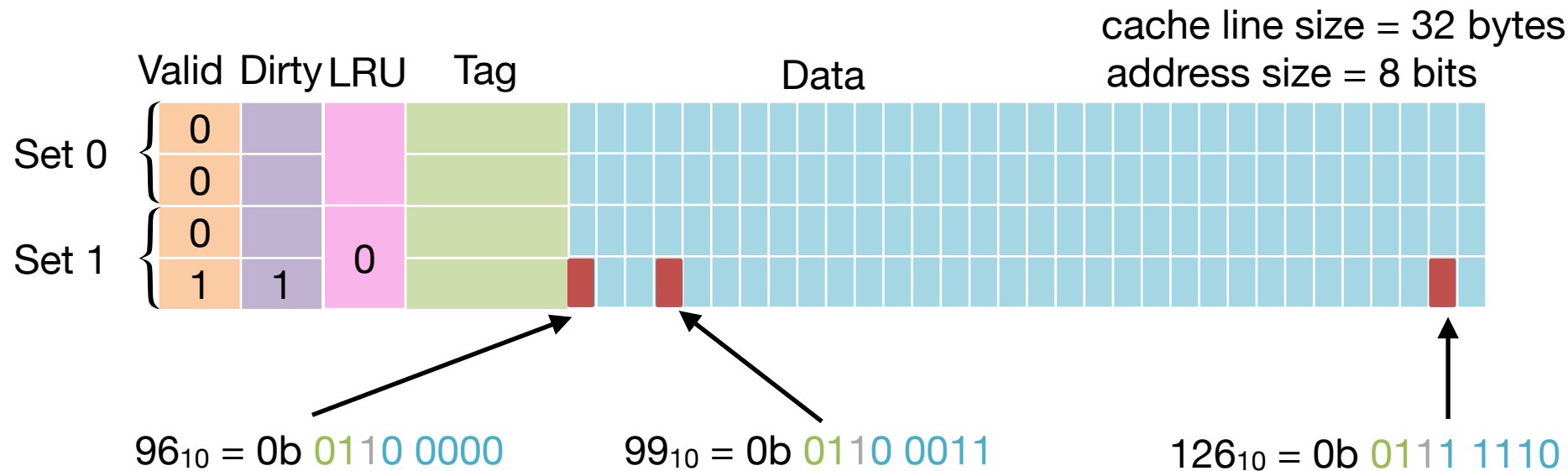
Addresses 1110 0000 through 1110 1111 are part of the same cache line and going to map to the same index in the cache

These two sets of addresses will conflict with each other

2-way Set-Associative Cache Example



2-way Set-Associative Cache Example



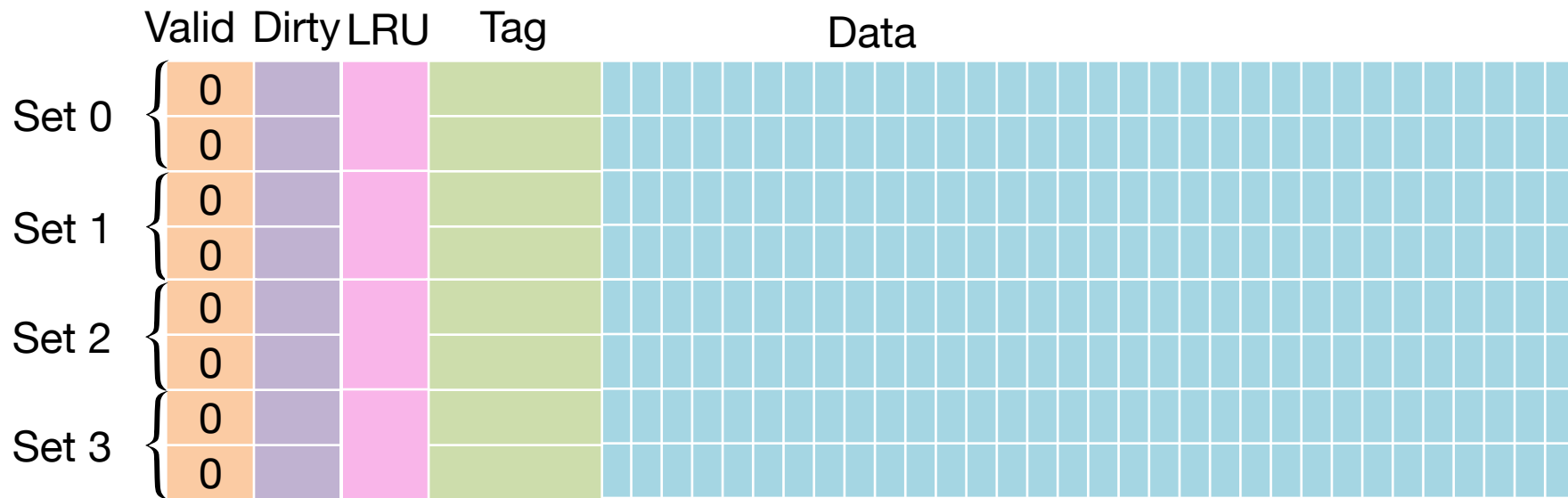
$$\# \text{ byte offset bits} = \log_2(\text{line size}) = \log_2(32) = 5$$

$$\# \text{ index bits} = \log_2(\# \text{ sets}) = \log_2(2) = 1$$

$$\# \text{ tag bits} = \# \text{ address bits} - \# \text{ offset bits} = 3$$

Larger 2-way Set-Associative Cache

cache line size = 32 bytes
cache size = 256 bytes

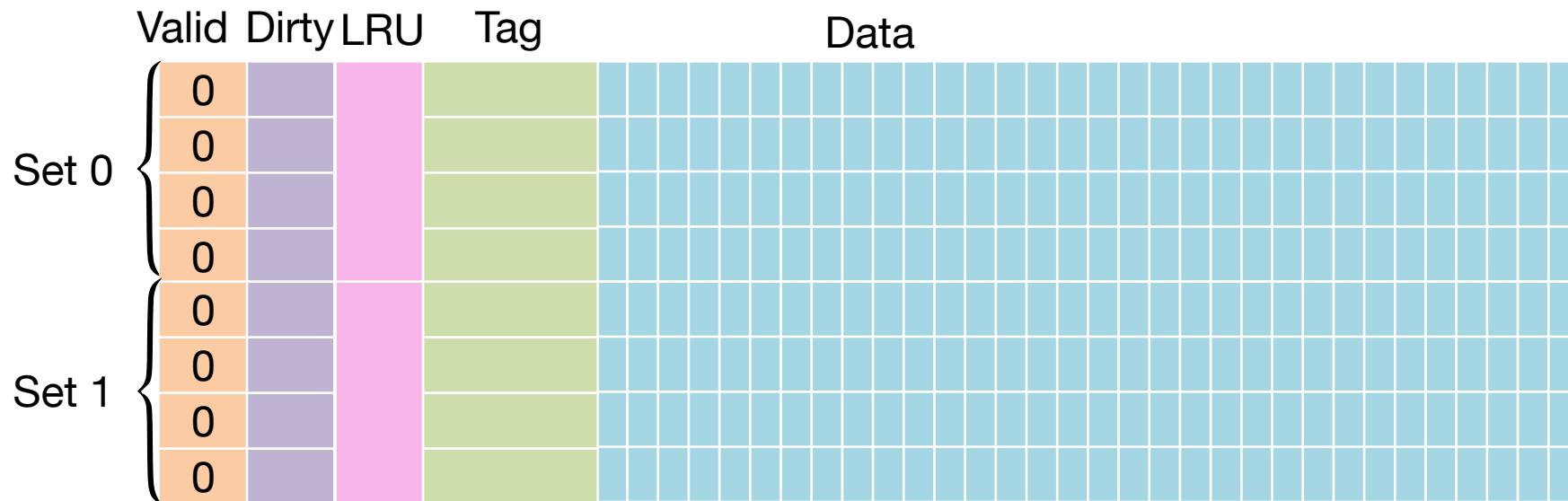


A **set** includes all of the **ways** of that index

The LRU bit tells us which **way** is the least recently used in the set

4-way Set-Associative Cache

cache line size = 32 bytes
cache size = 256 bytes



A **set** includes all of the **ways** of that index

The LRU bit tells us which **way** is the least recently used in the set

Comparing Layouts of an 8-Block Cache

**One-way set associative
(direct mapped)**

Block	Tag	Data
0		
1		
2		
3		
4		
5		
6		
7		

Two-way set associative

Set	Tag	Data	Tag	Data
0				
1				
2				
3				

Four-way set associative

Set	Tag	Data	Tag	Data	Tag	Data	Tag	Data
0								
1								

Eight-way set associative (fully associative)

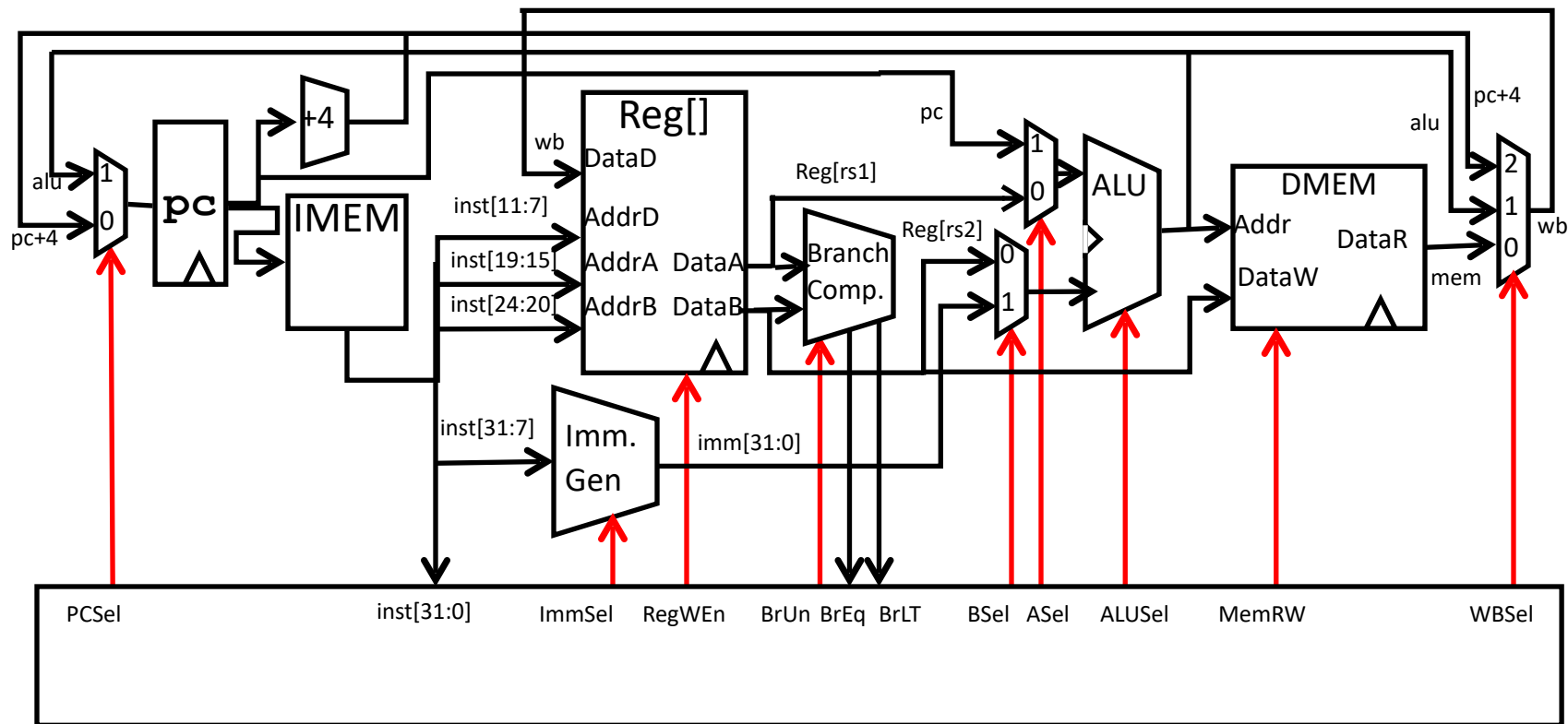
Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data	Tag	Data

With eight blocks, an 8-way set-associative cache is same as a fully associative cache

Pros and Cons

- Fully Associative
 - Pro: No conflicts (but you can still run out of room)
 - Con: Requires a lot of hardware to check for tag matches
- Direct Mapped
 - Pro: Only need to check one entry in the cache
 - Con: Lots of conflicts
- Set Associative
 - Pro: Less hardware than fully associative
 - Con: Still prone to conflicts (but less than direct mapped)

Recall: Single-Cycle RISC-V RV32I Datapath



Caches

- Our datapath has two memories: IMEM and DMEM
- Each of these memories have their own separate caches

Improving Cache Performance Through Programming Techniques

Array Stride

```
int sum_array(int *my_arr, int size, int stride) {  
    int sum = 0;  
    for (int i = 0; i < size; i += stride) {  
        sum += my_arr[i];  
    }  
    return sum;  
}
```

Array Strides

```
int sum_array(int *my_arr, int size, int stride) {  
    int sum = 0;  
    for (int i = 0; i < size; i += stride) {  
        sum += my_arr[i];  
    }  
    return sum;  
}
```

sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
lines = 16

stride = 1

my_arr 

 Miss
 Hit

Array Strides

```
int sum_array(int *my_arr, int size, int stride) {  
    int sum = 0;  
    for (int i = 0; i < size; i += stride) {  
        sum += my_arr[i];  
    }  
    return sum;  
}
```

sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
lines = 16

stride = 1

my_arr 

 Miss
 Hit

Array Strides

```
int sum_array(int *my_arr, int size, int stride) {  
    int sum = 0;  
    for (int i = 0; i < size; i += stride) {  
        sum += my_arr[i];  
    }  
    return sum;  
}
```


sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
lines = 16

stride = 2

my_arr 

 Miss

 Hit

 Brought in, but unused

Array Strides

```
int sum_array(int *my_arr, int size, int stride) {  
    int sum = 0;  
    for (int i = 0; i < size; i += stride) {  
        sum += my_arr[i];  
    }  
    return sum;  
}
```


sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
lines = 16

stride = 2

my_arr 

 Miss

 Hit

 Brought in, but unused

Array Strides

```
int sum_array(int *my_arr, int size, int stride) {  
    int sum = 0;  
    for (int i = 0; i < size; i += stride) {  
        sum += my_arr[i];  
    }  
    return sum;  
}
```


sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
lines = 16

stride = 4

my_arr 

 Miss

 Hit

 Brought in, but unused

Array Strides

```
int sum_array(int *my_arr, int size, int stride) {  
    int sum = 0;  
    for (int i = 0; i < size; i += stride) {  
        sum += my_arr[i];  
    }  
    return sum;  
}
```

sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
lines = 16

stride = 4



Miss

Hit

Brought in, but unused

Array Strides

```
int sum_array(int *my_arr, int size, int stride) {  
    int sum = 0;  
    for (int i = 0; i < size; i += stride) {  
        sum += my_arr[i];  
    }  
    return sum;  
}
```

sizeof(int) = 4 bytes
array size = 32 elements
Fully associative cache
line size = 16 bytes
lines = 16

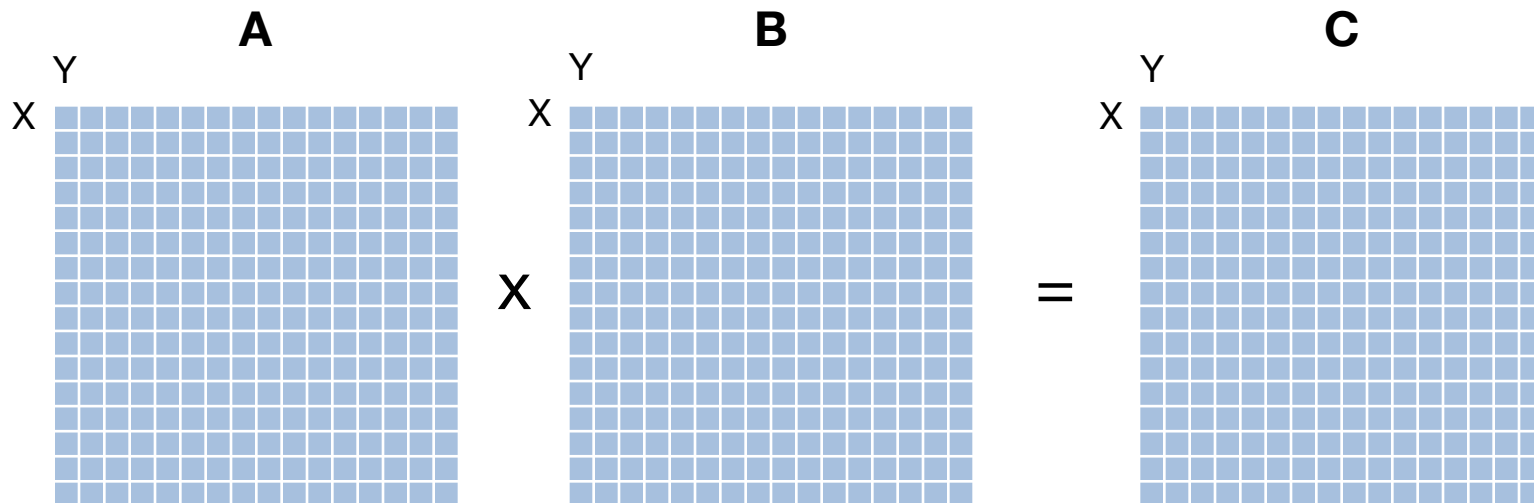
stride = 4



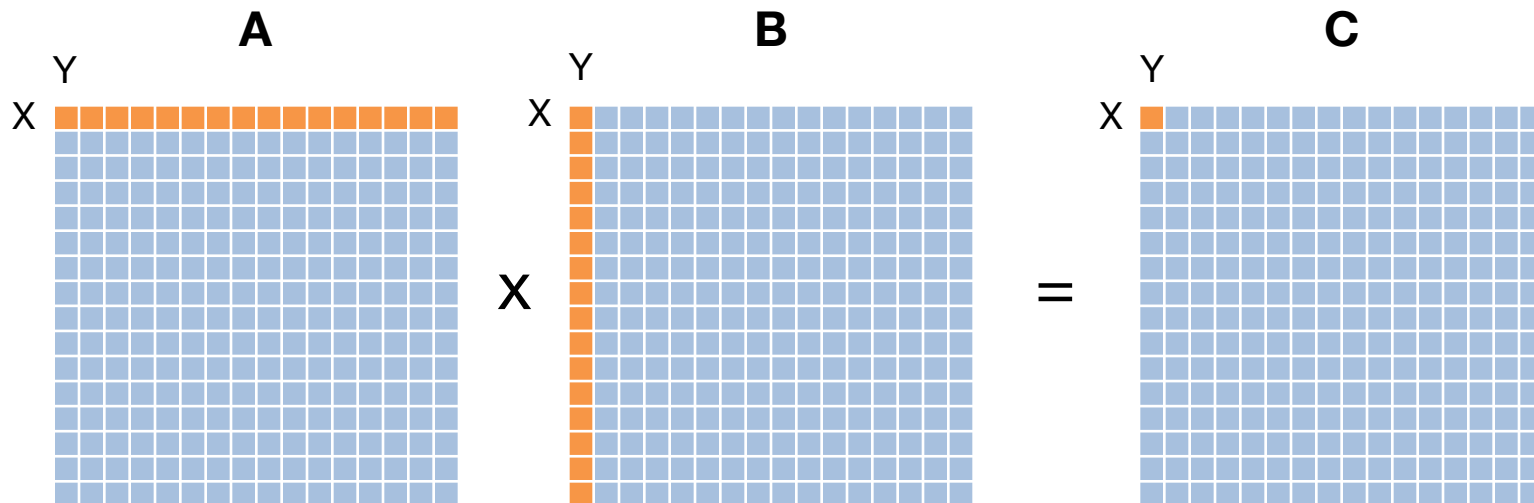
If the stride \geq block size, you
don't take advantage of
bringing in an entire line

- Miss
- Hit
- Brought in, but unused

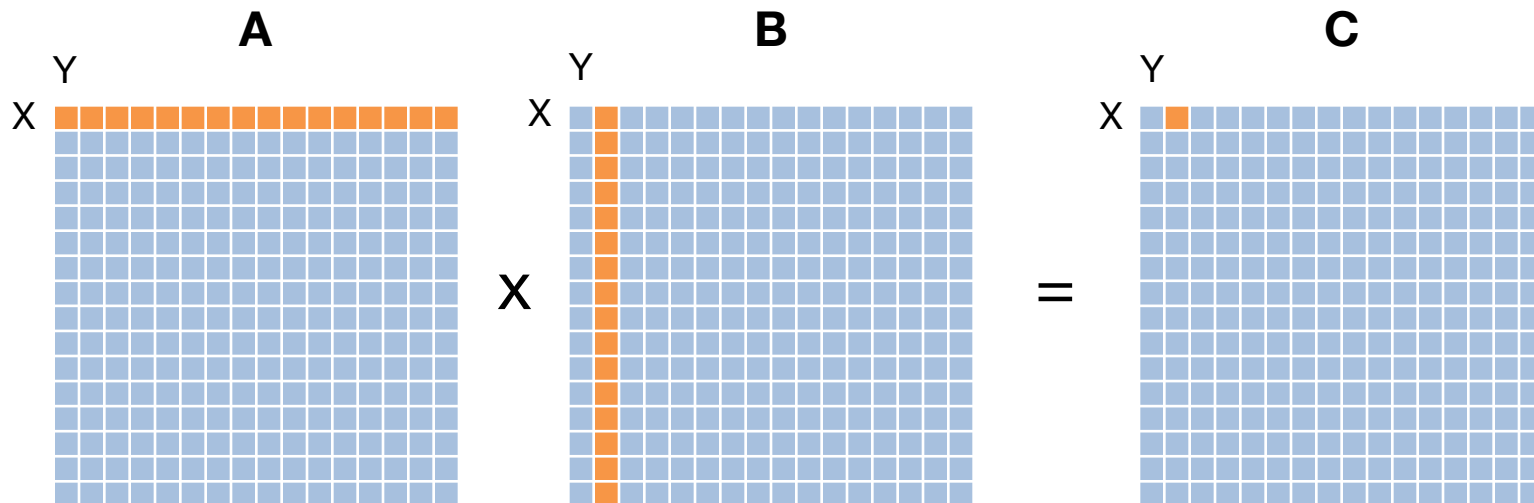
Matrix Multiply



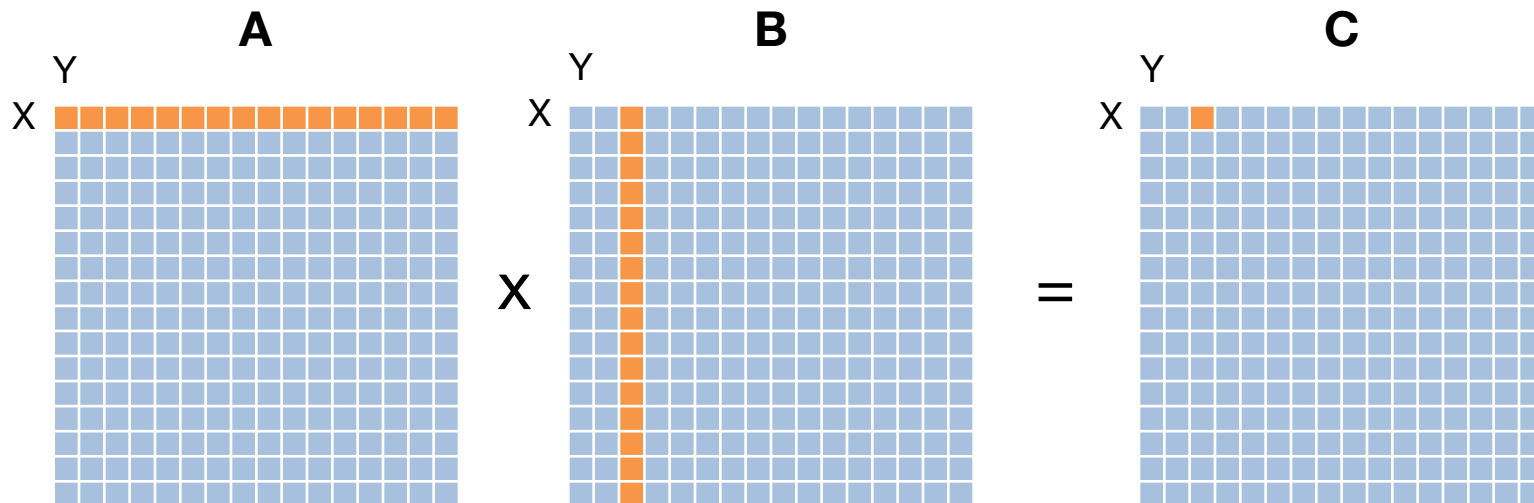
Matrix Multiply



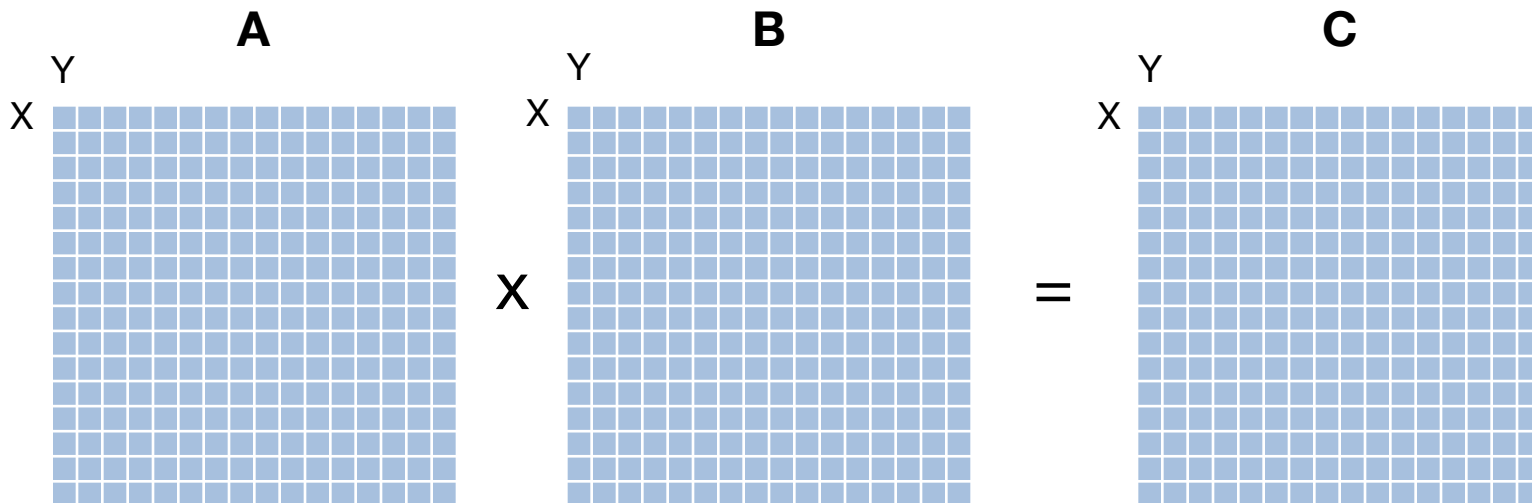
Matrix Multiply



Matrix Multiply



Matrix Multiply

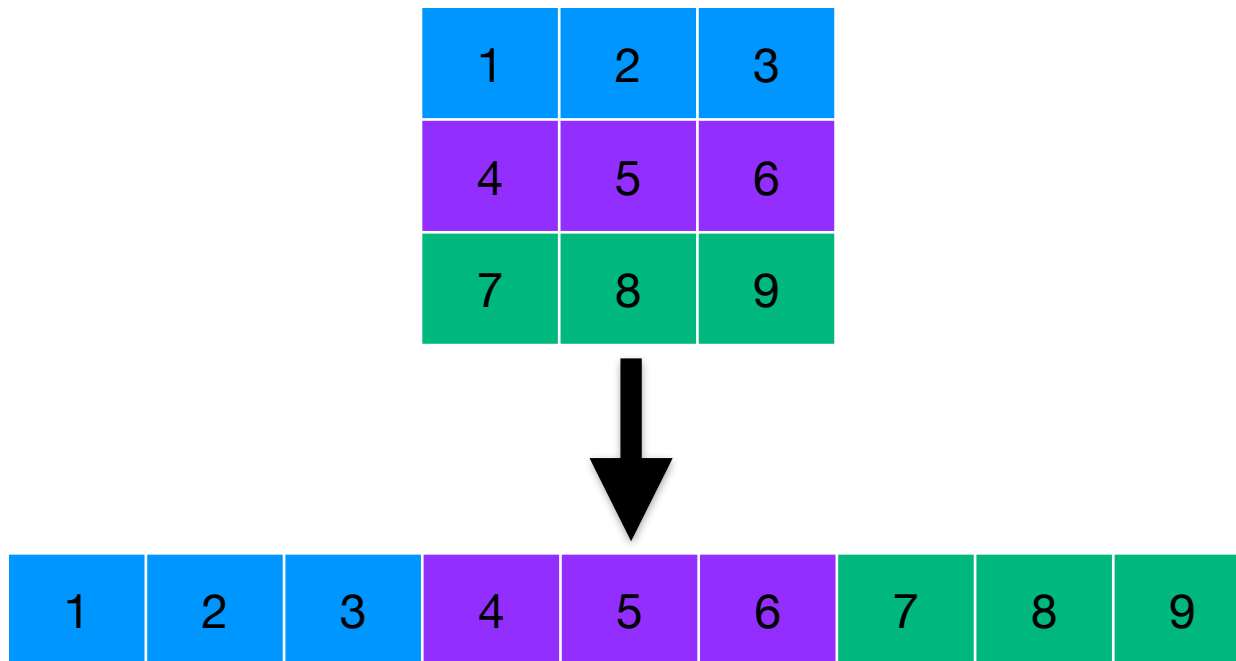


matrix of integers
`sizeof(int) = 4 bytes`

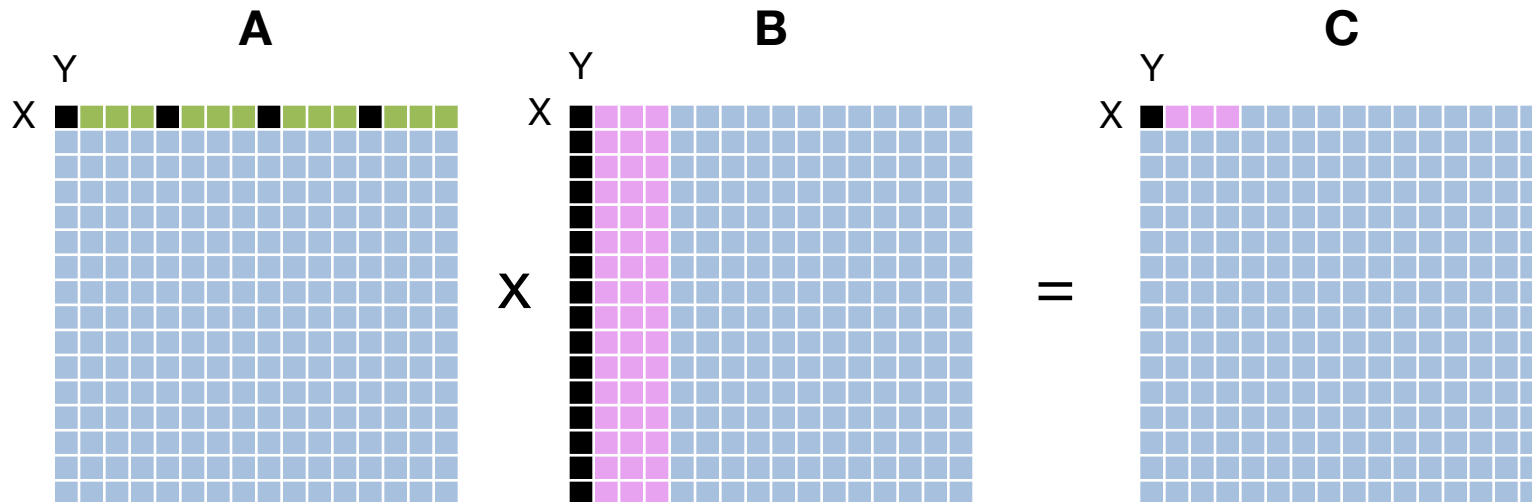
Fully associative cache
line size = 16 bytes
lines = 16

Matrix Multiply

- Arrays are stored in row-major order



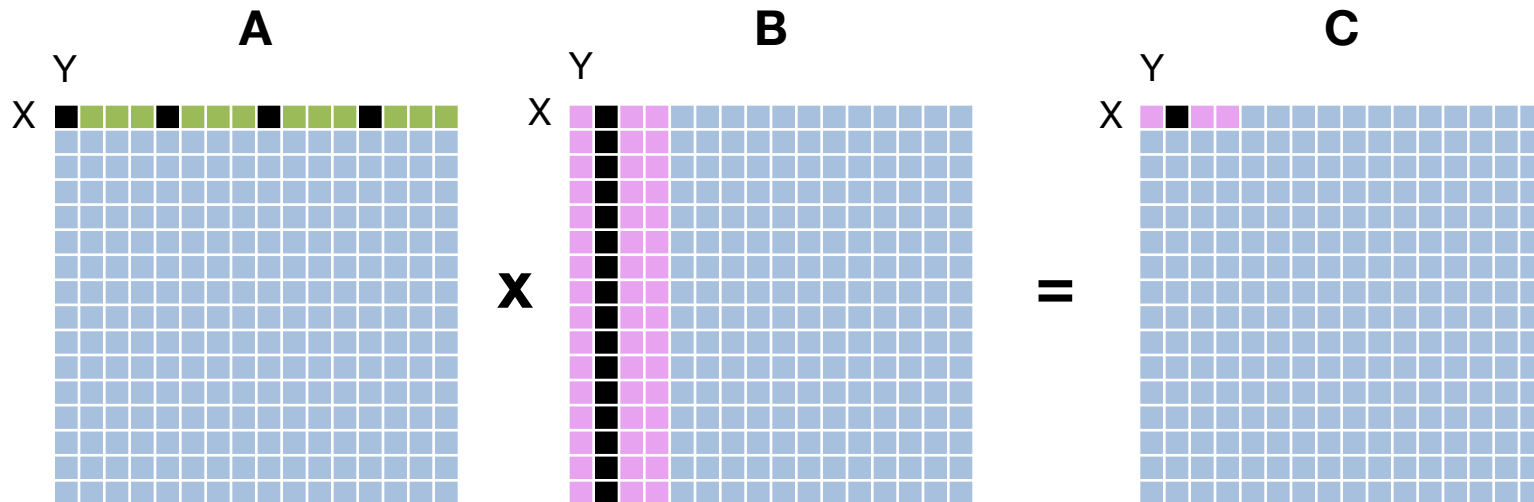
Matrix Multiply



- Miss
- Hit
- Brought in, but unused before eviction

Fully associative cache
line size = 16 bytes
lines = 16

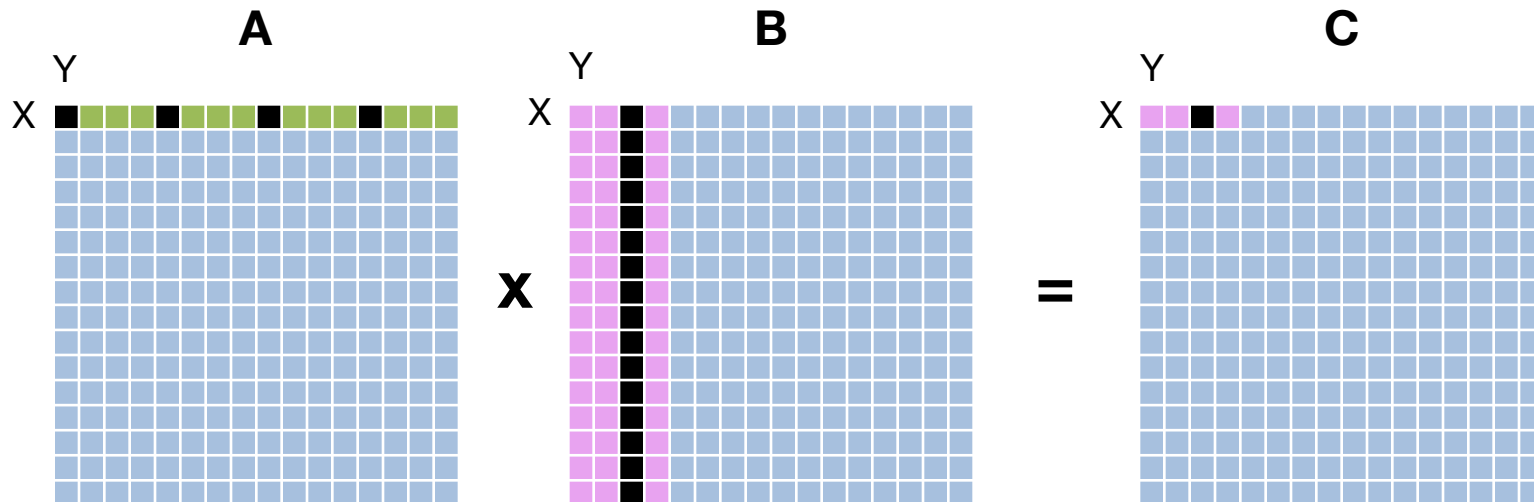
Matrix Multiply



- Miss
- Hit
- Brought in, but unused before eviction

Fully associative cache
line size = 16 bytes
lines = 16

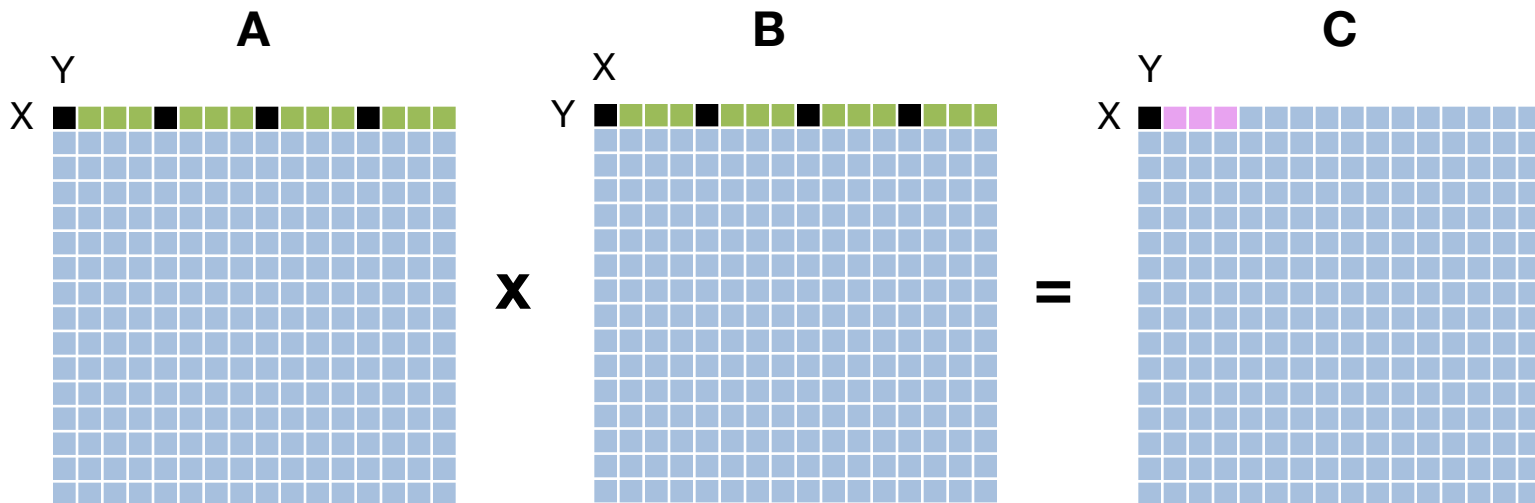
Matrix Multiply



- Miss
- Hit
- Brought in, but unused before eviction

Fully associative cache
line size = 16 bytes
lines = 16

Making better use of the cache by transposing the matrix



- Miss
- Hit
- Brought in, but unused before eviction

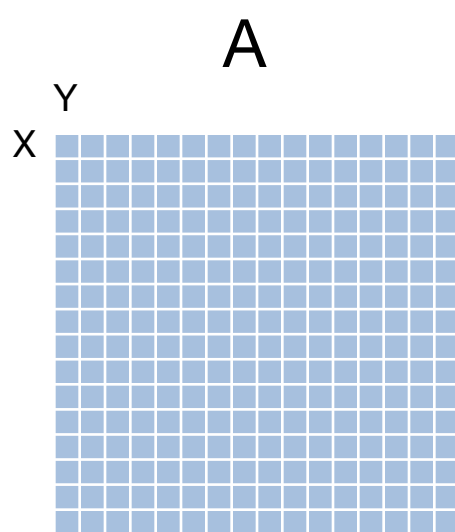
Fully associative cache
line size = 16 bytes
lines = 16

Cache Blocking

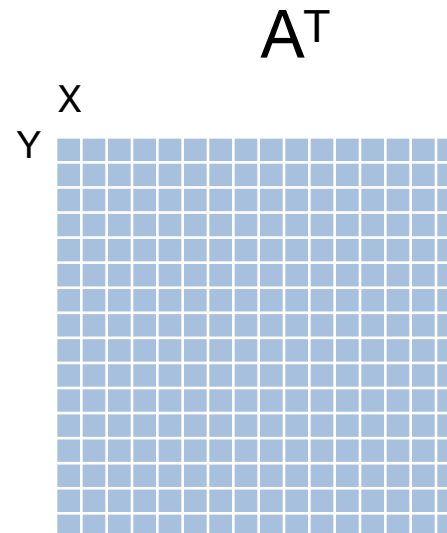
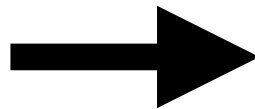
Cache Blocking

- A technique where data accesses are rearranged to make better use of the data that is brought into the cache
- Helps prevent repeatedly evicting and fetching the same data from the main memory

Matrix Transpose

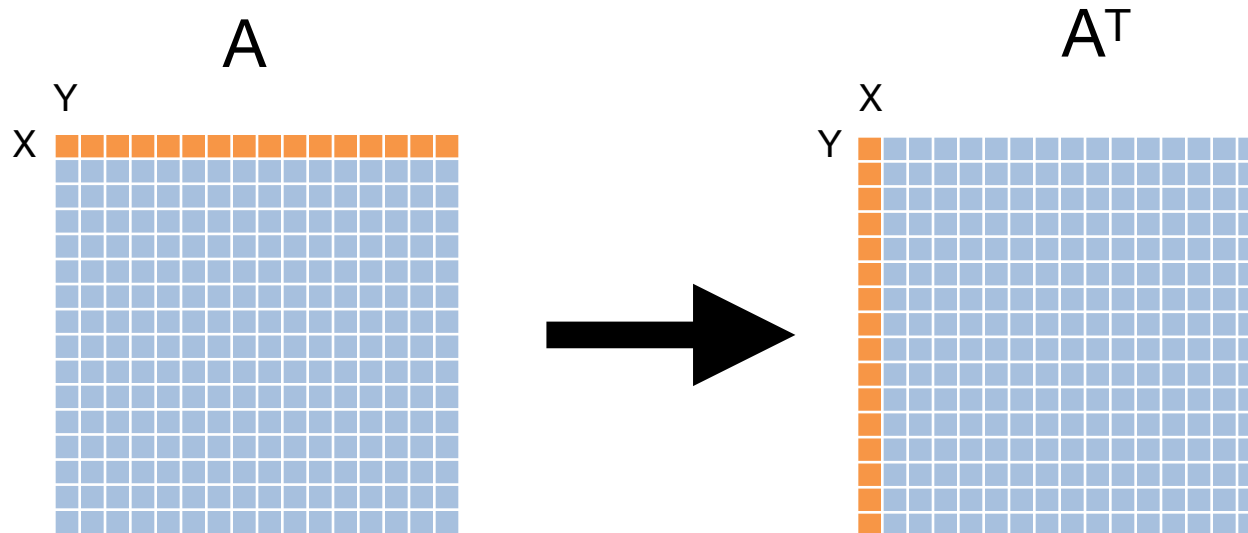


integer matrix
`sizeof(int) = 4 bytes`



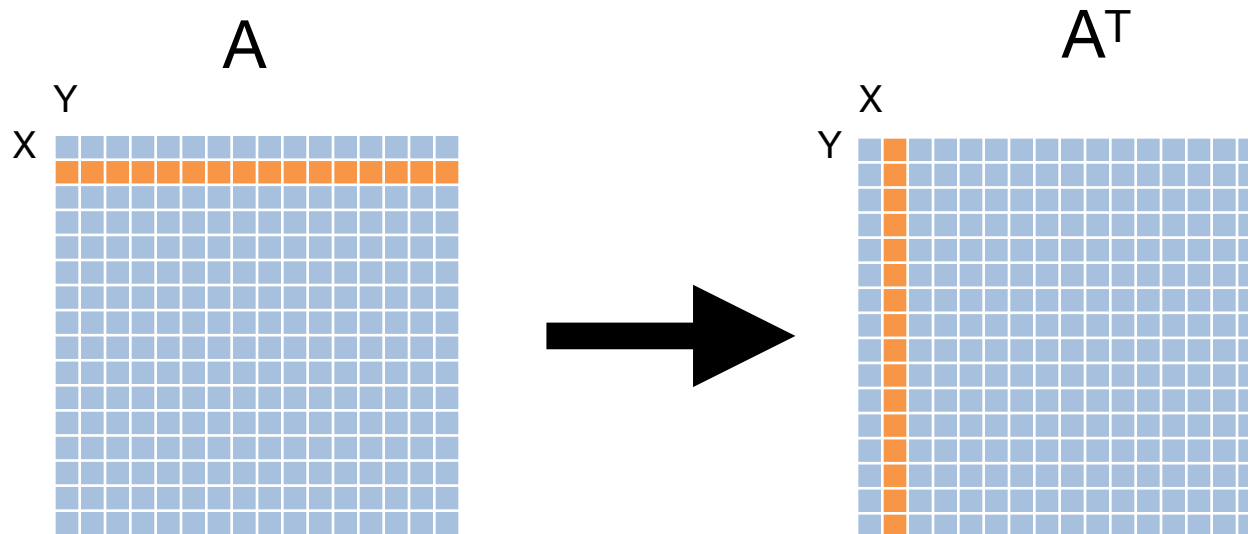
Fully associative cache
line size = 16 bytes
lines = 16

Matrix Transpose



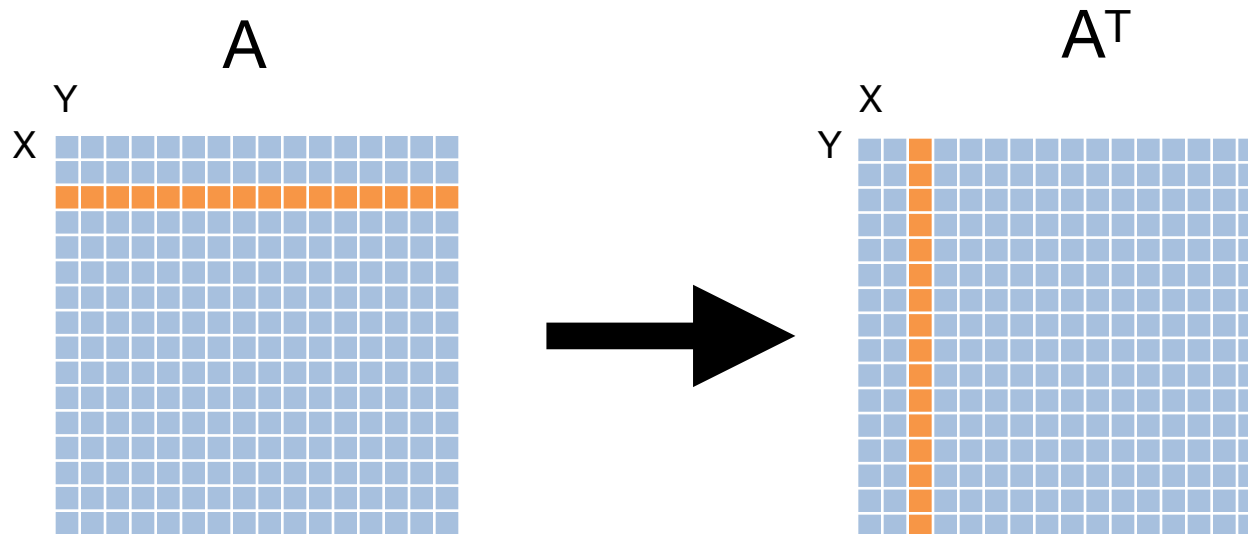
 Current Transpose

Matrix Transpose



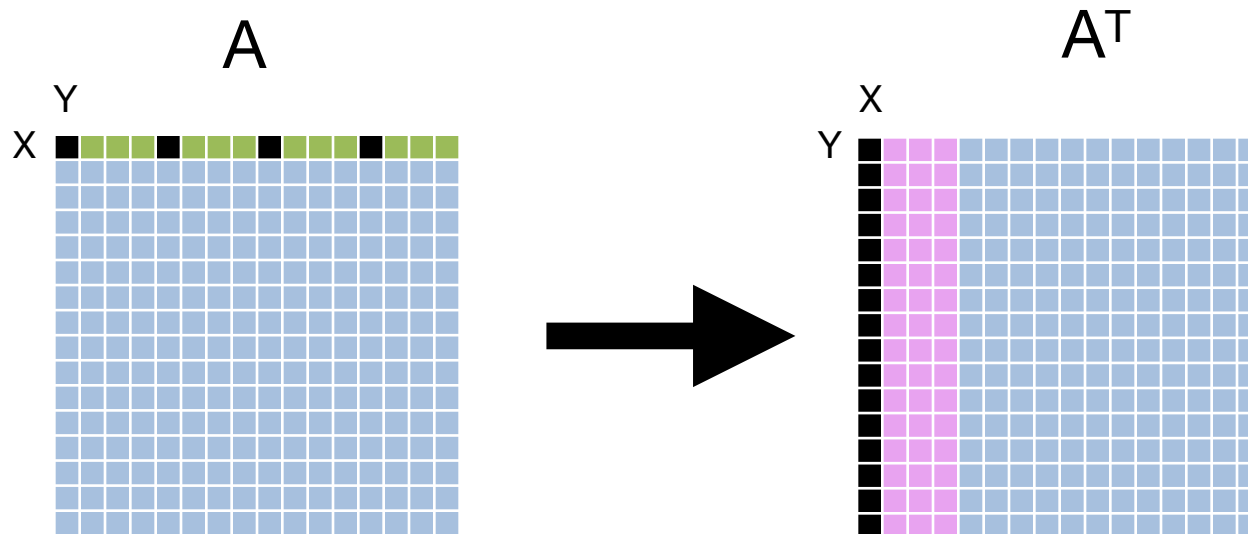
 Current Transpose

Matrix Transpose



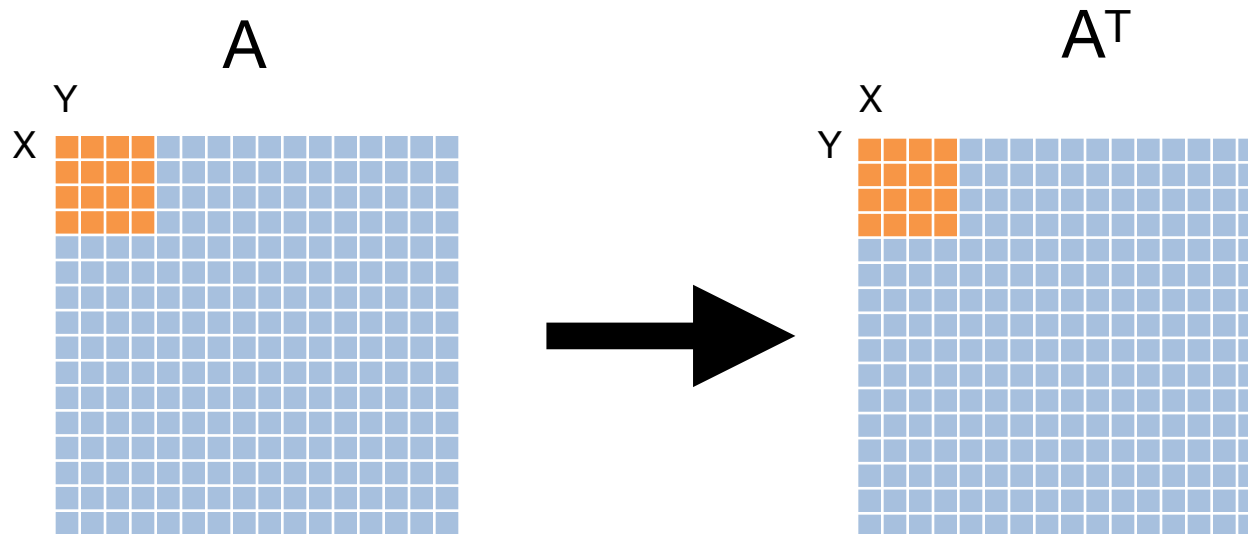
 Current Transpose

Matrix Transpose



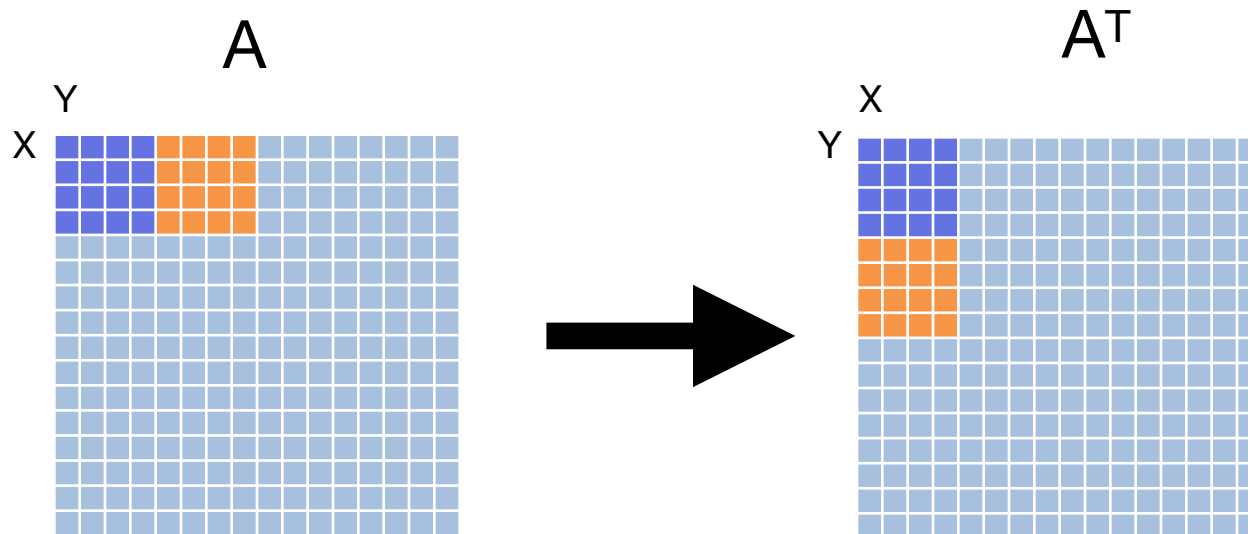
- Miss
- Hit
- Brought in, but unused before eviction



Matrix Transpose



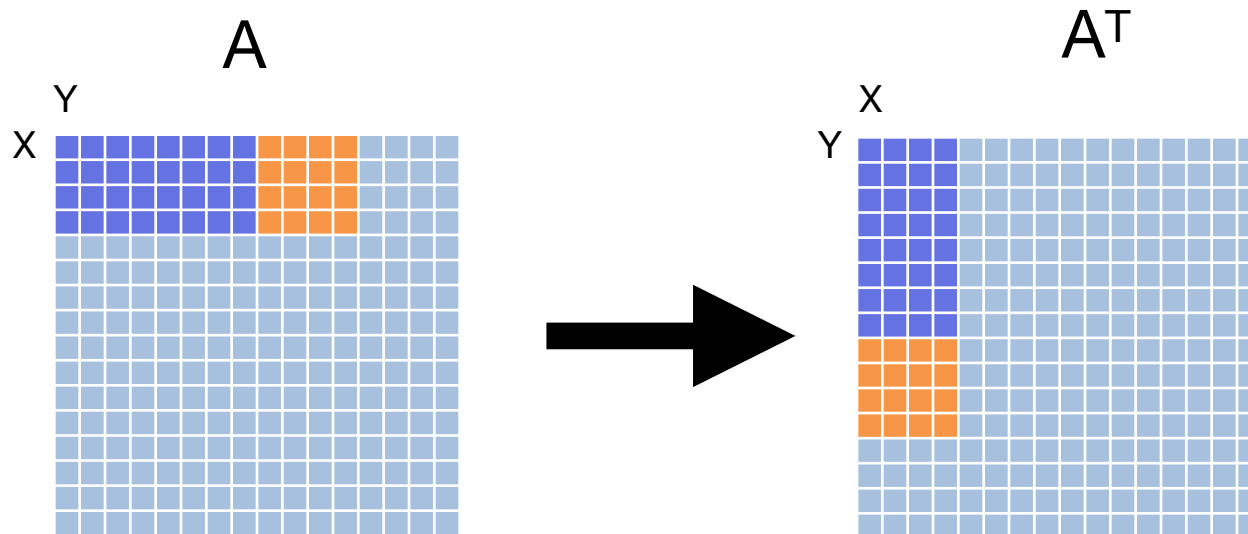
 Current Transpose



Matrix Transpose



-  Current Transpose
-  Already Transposed

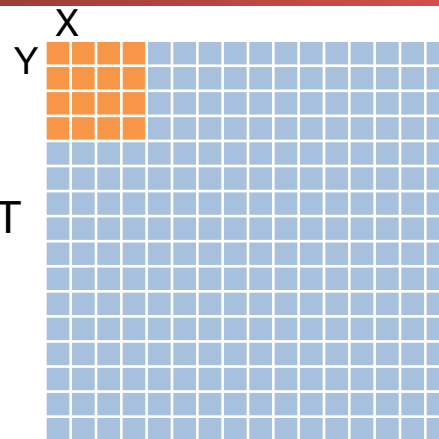
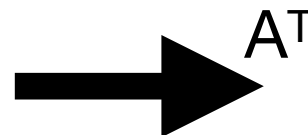
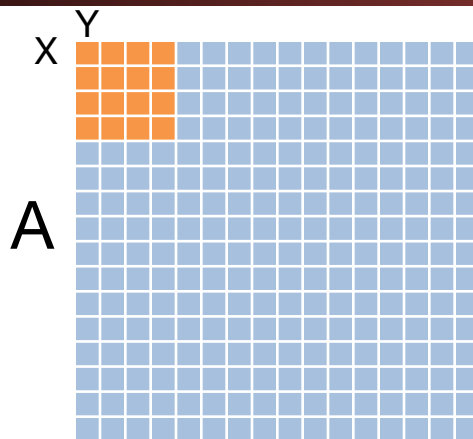
Matrix Transpose



-  Current Transpose
-  Already Transposed

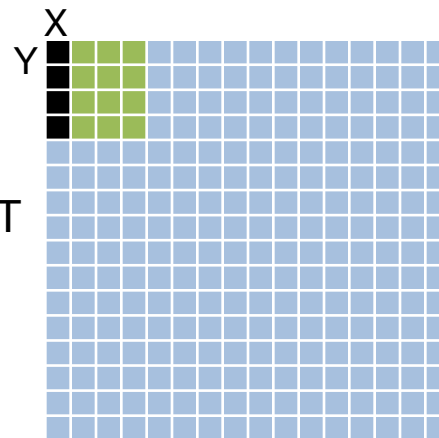
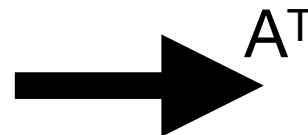
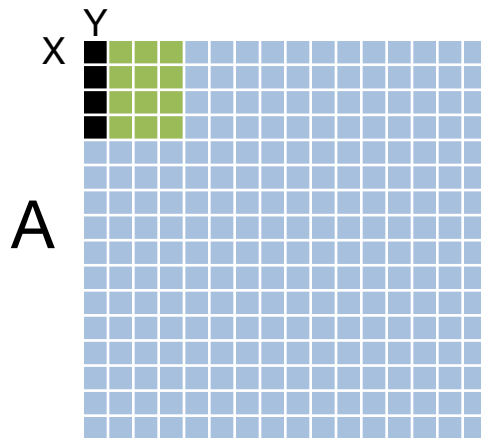
Matrix Transpose

 Current
Transpose



 Miss

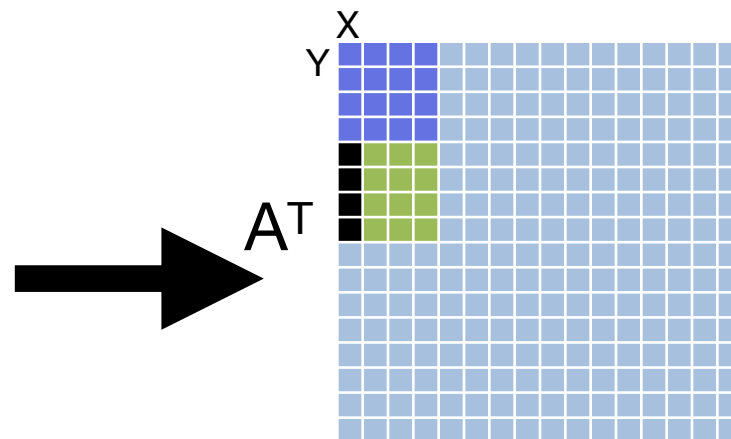
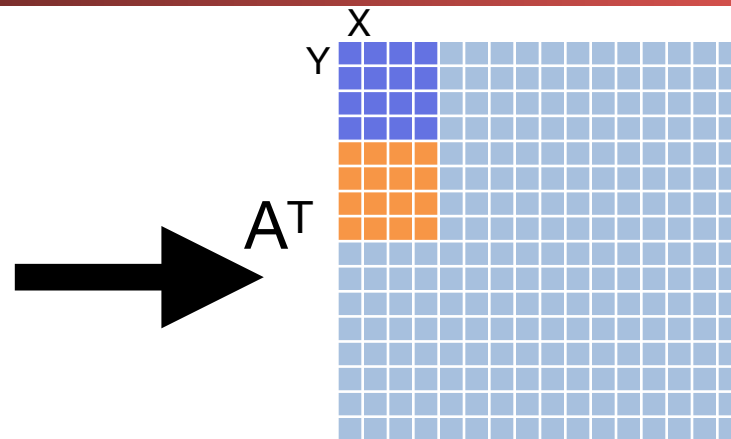
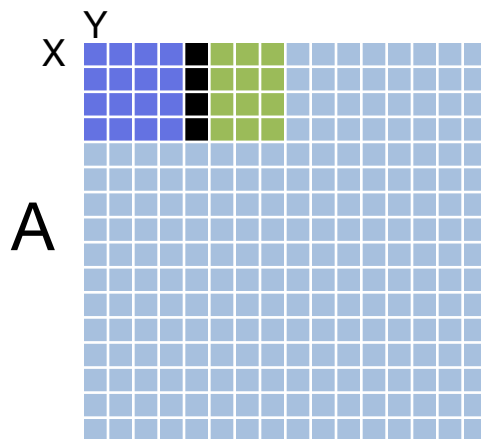
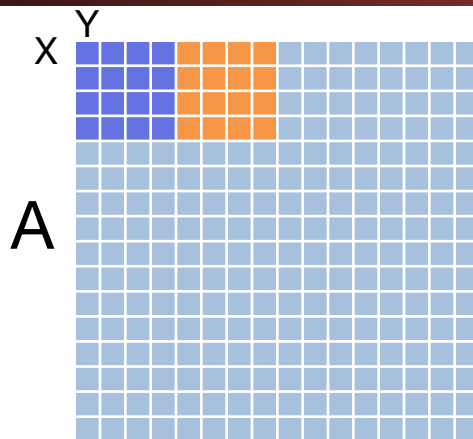
 Hit





Matrix Transpose

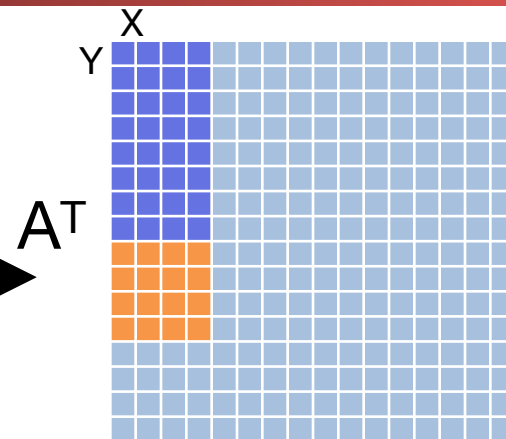
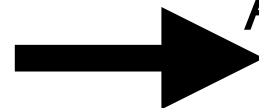
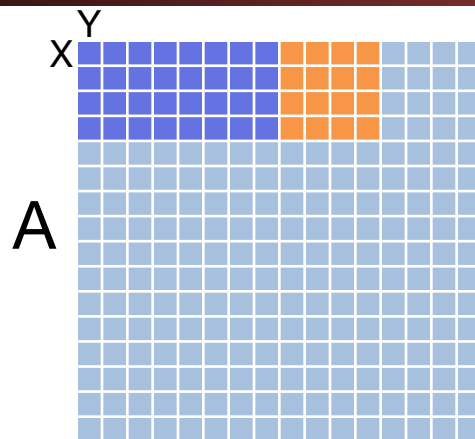
Current Transpose
Already Transposed

Miss
Hit

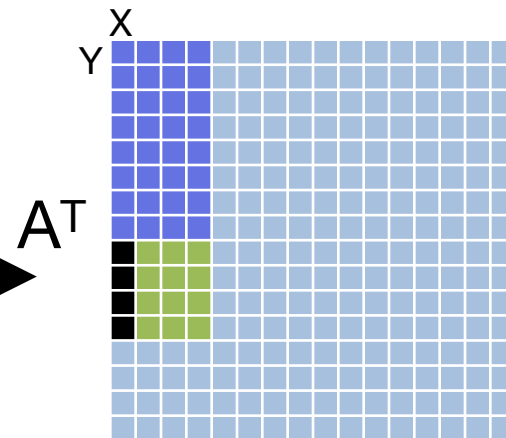
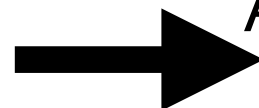
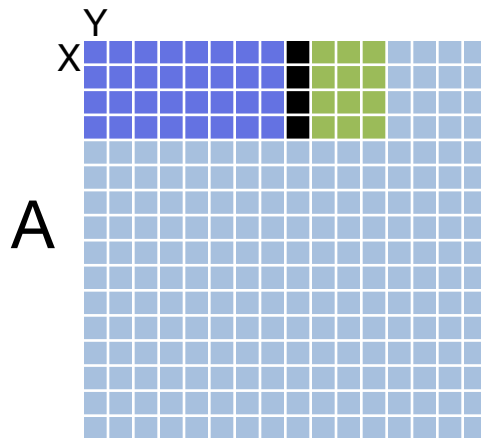


Matrix Transpose

 Current Transpose
 Already Transposed



 Miss
 Hit



Analyzing Cache Performance

Terminology

- Hit Rate
 - number of hits / number of accesses
- Miss Rate
 - $1 - \text{hit rate}$
- Hit Time
 - The time that it takes for you to access an item on a cache hit
- Miss penalty
 - On a miss, the time it takes to access the block after discovering that its not in the cache

Average Memory Access Time (AMAT)

- $AMAT = \text{hit time} + \text{miss rate} * \text{miss penalty}$

AMAT Example

- What is the AMAT of a system where
 - Hit rate = 90%
 - Hit time = 4 cycles
 - Miss penalty = 20 cycles

$$\text{AMAT} = \text{hit time} + \text{miss rate} * \text{miss penalty}$$

$$\text{AMAT} = 4 \text{ cycles} + 0.1(20 \text{ cycles})$$

$$\text{AMAT} = 6 \text{ cycles}$$

AMAT Example (Your turn)

- What is the AMAT of a system where
 - Hit rate = 75%
 - Hit time = 5 cycles
 - Miss penalty = 24 cycles

AMAT Example (Your turn)

- What is the AMAT of a system where
 - Hit rate = 75%
 - Hit time = 5 cycles
 - Miss penalty = 24 cycles

$$\text{AMAT} = \text{hit time} + \text{miss rate} * \text{miss penalty}$$

$$\text{AMAT} = 5 \text{ cycles} + 0.25(24 \text{ cycles})$$

$$\text{AMAT} = 11 \text{ cycles}$$

How Does Associativity Affect AMAT?

- Hit time as associativity increases?
 - Increases
 - Direct Mapped \rightarrow 2-way
 - Introduce a multiplexor to choose correct way
 - 2-way \rightarrow 4-way
 - Smaller increase than direct mapped \rightarrow 2-way
 - The multiplexor is larger for 4-way
- Miss rate as associativity increases?
 - Decreases due to less conflict misses
- Miss penalty as associativity changes?
 - Mostly unchanged, replacement policy runs in parallel with fetching missing line from memory

How does #entries affect AMAT?

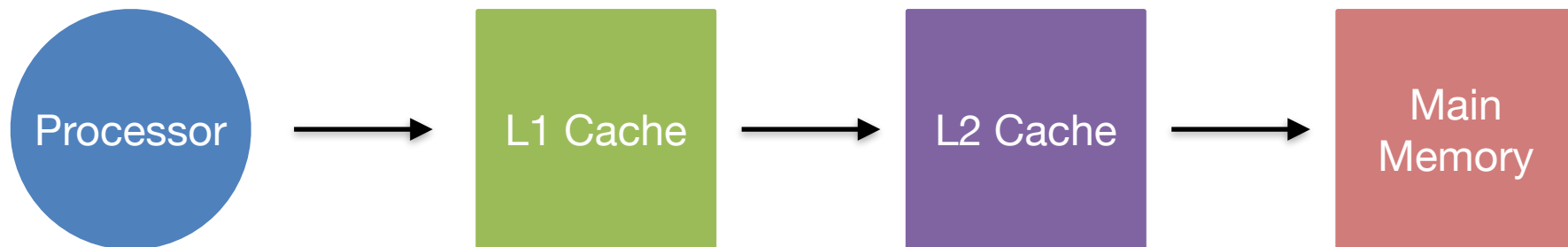
- Hit time as #entries increases?
 - Increases, since reading tags and data from larger memory structures
- Miss rate as #entries increases?
 - Goes down due to increased capacity and fewer conflict misses
- Miss penalty as #entries increases?
 - Unchanged
- At some point, the increase in hit time for a larger cache may overcome the improvement in hit rate, yielding a decrease in performance

How does block size affect AMAT?

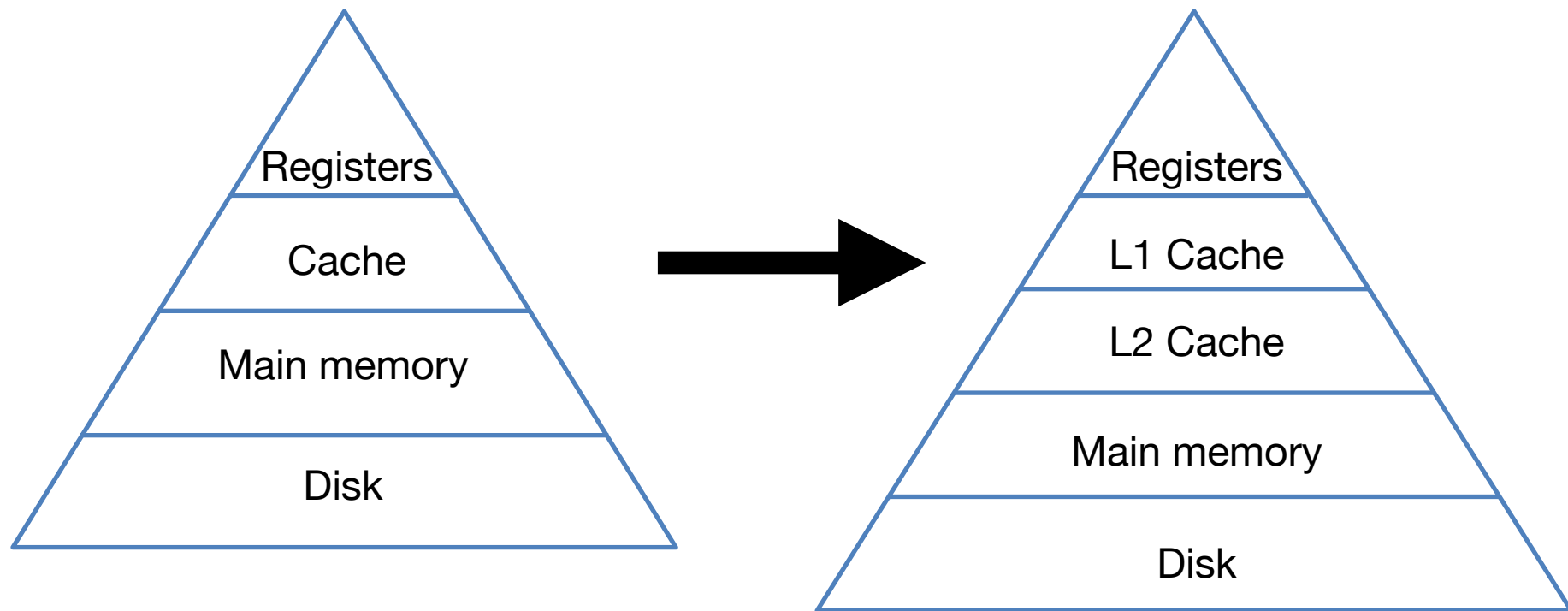
- Hit time as block size increases?
 - Hit time mostly unchanged, but might be slightly reduced as number of tags is reduced
- Miss rate as block size increases?
 - Goes down at first due to spatial locality, then increases due to increased conflict misses due to fewer blocks in the cache
- Miss penalty as block size increases?
 - Rises with larger block size

Another way to reduce miss penalty

- Include another cache!



Memory Hierarchy



L2 Cache

- L2 is bigger than L1
 - Leads to higher hit rate
- L2 is accessed only if the requested data is not found in L1
- L2 takes longer to access because it is larger and farther away from the processor
- All data in L1 can be found in L2 as well*
- If the line in L1 is dirty when it is evicted, you update the copy in L2*

* depends on policy

Additional Caches

- L1 cache
 - Embedded in the processor chip
 - Fast, but limited storage capacity
- L2 cache
 - Embedded on the processor chip OR on its own separate chip
 - Reduces L1 miss penalty
- L3 cache
 - On a separate chip
 - Reduces L1 and L2 miss penalty
- L4 cache (uncommon)

Core i7-6500U (Skylake)

- L1 data cache
 - Size: 32 kB
 - Associativity: 8
 - Number of sets: 64
 - Way size: 4 kB
 - Latency: 4 cycles [Link](#)
 - Replacement policy: Tree-PLRU (with linear insertion order if empty) [Link 1](#) [Link 2](#)
- L2 cache
 - Size: 256 kB
 - Associativity: 4
 - Number of sets: 1024
 - Way size: 64 kB
 - Latency: 12 cycles [Link](#)
 - Replacement policy: QLRU_H00_M1_R2_U1 [Link](#)
 - Similar to the Cannon Lake L2 policy, but:
 - If the cache is empty (after executing the WBINVD instruction), blocks are inserted from right to left
 - The initial ages of blocks inserted into an empty cache can depend on the previous state
 - See also [Vila et al.](#)
- L3 cache
 - Size: 4 MB
 - Associativity: 16
 - Number of CBoxes: 2
 - Number of slices: 4
 - Number of sets (per slice): 1024
 - Way size (per slice): 64 kB
 - Latency: 34 cycles [Link](#)
 - Replacement policy: Adaptive [Link](#)

Local vs Global Hit Rate

- Local Hit Rate
 - $\# \text{ hits at this level} / \# \text{ accesses to this level}$
- Global Hit rate
 - $\# \text{ hits at this level} / \# \text{ **total** number of accesses}$

AMAT with 2-level Cache

$$\text{AMAT} = \text{hit time} + \text{miss rate} * \text{miss penalty}$$

** all miss rates are local

AMAT with 2-level Cache

$$\text{AMAT} = \text{hit time} + \text{miss rate} * \text{miss penalty}$$

$$\text{AMAT} = \text{L1 hit time} + \text{L1 miss rate} * \text{L1 miss penalty}$$

$$\text{L1 miss penalty} = \text{L2 hit time} + \text{L2 miss rate} * \text{L2 miss penalty}$$

$$\text{AMAT} = \text{L1 hit time} + \text{L1 miss rate} * (\text{L2 hit time} + \text{L2 miss rate} * \text{L2 miss penalty})$$

** all miss rates are local

AMAT Example

- What is the AMAT of a system where
 - L1 hit rate = 75%
 - L1 hit time = 4 cycles
 - L2 hit rate = 90%
 - L2 hit time = 6 cycles
 - L2 miss penalty = 20 cycles

$$\text{AMAT} = \text{L1 hit time} + \text{L1 miss rate} * (\text{L2 hit time} + \text{L2 miss rate} * \text{L2 miss penalty})$$

$$\text{AMAT} = 4 + 0.25 * (6 \text{ cycles} + 0.1 * 20 \text{ cycles})$$

$$\text{AMAT} = 6 \text{ cycles}$$

AMAT Example (Your turn)

- What is the AMAT of a system where
 - L1 hit rate = 60%
 - L1 hit time = 5 cycles
 - L2 hit rate = 95%
 - L2 hit time = 8 cycles
 - L2 miss penalty = 40 cycles

$$\text{AMAT} = \text{L1 hit time} + \text{L1 miss rate} * (\text{L2 hit time} + \text{L2 miss rate} * \text{L2 miss penalty})$$

AMAT Example (Your turn)

- What is the AMAT of a system where
 - L1 hit rate = 60%
 - L1 hit time = 5 cycles
 - L2 hit rate = 95%
 - L2 hit time = 8 cycles
 - L2 miss penalty = 40 cycles

$$\text{AMAT} = \text{L1 hit time} + \text{L1 miss rate} * (\text{L2 hit time} + \text{L2 miss rate} * \text{L2 miss penalty})$$

$$\text{AMAT} = 5 + 0.4 * (8 \text{ cycles} + 0.05 * 40 \text{ cycles})$$

$$\text{AMAT} = 9 \text{ cycles}$$

Learn about your computer's caches

- **MacOS:** `sysctl -a hw machdep.cpu`
- **Linux:** `lscpu`
- **Windows:** `wmic memcache list brief`
 - (I don't know if this actually works...)