Caches
If we want to read byte $99_{10}$ and it's not in the cache, we would bring in bytes $96_{10}$-$127_{10}$ into the cache.
If we then wanted to read byte $126_{10}$, we would get a cache hit because we just brought in the line that it's 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

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
<th>cache line size = 32 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>address size = 8 bits</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$96_{10} = 0b\ 0110\ 0000$  
$99_{10} = 0b\ 0110\ 0011$  
$126_{10} = 0b\ 0111\ 1110$
Fully Associative Cache Example

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0x3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- $96_{10} = 0b\ 0110\ 0000$
- $99_{10} = 0b\ 0110\ 0011$
- $126_{10} = 0b\ 0111\ 1110$

- $96_{10} = 0b\ 0110\ 0000$
- $99_{10} = 0b\ 0110\ 0011$
- $126_{10} = 0b\ 0111\ 1110$

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

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

cache line size = 32 bytes
address size = 8 bits
### Fully Associative Cache Example

- **Valid, Dirty, Tag, Data**
- **Cache line size = 32 bytes**
- **Address size = 8 bits**

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0x3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0x3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- \(96_{10} = 0b\ 0110\ 0000\)
- \(99_{10} = 0b\ 0110\ 0011\)
- \(126_{10} = 0b\ 0111\ 1110\)

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

\(#\) tag bits = \(\#\) address bits - \(\#\) offset bits = 3
Direct Mapped Cache Example

- Valid
- Dirty
- LRU
- Tag
- Data

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>Tag</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- cache line size = 32 bytes
- address size = 8 bits

- $96_{10} = 0b\ 0110\ 0000$
- $99_{10} = 0b\ 0110\ 0011$
- $126_{10} = 0b\ 0111\ 1110$
Direct Mapped Cache Example

- **Valid**: 0 or 1
- **Dirty**: 0 or 1
- **LRU**: 0 or 1
- **Tag**: 0 or 1
- **Data**: 0x0

**Valid**: 0
**Dirty**: 0
**LRU**: 0
**Tag**: 0x0
**Data**: 0x0

- **96_{10} = 0b 0110 0000**
- **99_{10} = 0b 0110 0011**
- **126_{10} = 0b 0111 1110**

- **96_{10} = 0b 0110 0000**
- **99_{10} = 0b 0110 0011**
- **126_{10} = 0b 0111 1110**

**# byte offset bits = log_2(line size) = log_2(32) = 5**

**# index bits = log_2(# lines) = log_2(4) = 2**

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

- **cache line size = 32 bytes**
- **address size = 8 bits**
### Direct Mapped Cache Example

- **Cache Line Size**: 32 bytes
- **Address Size**: 8 bits

<table>
<thead>
<tr>
<th>Valid</th>
<th>Dirty</th>
<th>LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0x0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
<td>0x0</td>
</tr>
</tbody>
</table>

**Addresses**

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

- Cache line size = 32 bytes
- Address size = 8 bits

**Set 0**
- Valid: 0
- Dirty: 0
- LRU: 0
- Tag: 0
- Data: 96₁₀ = 0b 0110 0000

**Set 1**
- Valid: 0
- Dirty: 0
- LRU: 0
- Tag: 0
- Data: 99₁₀ = 0b 0110 0011

**LRU**
- 0

**Example Addresses**
- 96₁₀ = 0b 0110 0000
- 99₁₀ = 0b 0110 0011
- 126₁₀ = 0b 0111 1110
2-way Set-Associative Cache Example

- **Valid**: Indicates if the cache line is valid in memory.
- **Dirty**: Indicates if the cache line is dirty.
- **LRU**: Least Recently Used.
- **Tag**: Identifies the cache set.
- **Data**: Represents the data stored in the cache line.

**Cache Line Details**:
- **Cache line size**: 32 bytes
- **Address size**: 8 bits

**Address Calculation**:
- # byte offset bits = $\log_2(\text{line size}) = \log_2(32) = 5$
- # index bits = $\log_2(\# \text{ sets}) = \log_2(2) = 1$
- # tag bits = # address bits - # offset bits = 3

**Example Addresses**:
- $96_{10} = 0b\ 0110\ 0000$
- $99_{10} = 0b\ 0110\ 0011$
- $126_{10} = 0b\ 0111\ 1110$
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.
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

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

```c
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

```c
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

```c
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` 

<table>
<thead>
<tr>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
<th>Hit</th>
</tr>
</thead>
</table>

`Miss`

`Hit`
Array Strides

```c
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` array:

  ![Array Diagram]

- `Miss`
- `Hit`
- `Brought in, but unused`
Array Strides

```c
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

```c
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

```c
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

### Parameters
- `my_arr`: Pointer to an array of 32 elements.
- `size`: The size of the array.
- `stride`: The stride of the array.

### Example
- `stride = 4`

```
my_arr:
[ 0  4  8 12 16 20 24 28 32 36 40 44 48 52 56 60 ]
```

- **Miss**: Elements not in cache.
- **Hit**: Elements in cache.
- **Brought in, but unused**: Elements brought in but not used.

---

**Computer Science 61C Spring 2022**

Kolb and Weaver

McMahon and Weaver

Berkeley EECS
Array Strides

```c
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

\[
\begin{array}{ccc}
\text{X} & \text{Y} & \text{A} \\
\text{X} & \text{Y} & \text{B} \\
\text{X} & \text{Y} & \text{C} \\
\end{array}
\]

=
Matrix Multiply

\[ \begin{array}{ccc}
X & Y & X \\
A & B & C
\end{array} \]
### Matrix Multiply

Matrix multiplication involves multiplying the rows of the first matrix by the columns of the second matrix. The resulting matrix has the number of rows of the first matrix and the number of columns of the second matrix.

Let's consider two matrices, $A$ and $B$, and their product $C$.

**Matrix $A$**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Matrix $B$**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

**Matrix $C$**

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

The product $C = A \times B$ is computed by taking the dot product of each row of $A$ with each column of $B$.
Matrix Multiply

\[ A \times B = C \]
Matrix Multiply

A

B

C

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

\[
\begin{array}{ccc}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{array}
\]
Matrix Multiply

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

Fully associative cache
- line size = 16 bytes
- # lines = 16
Matrix Multiply

A
\[
\begin{array}{ccc}
X & Y & X \\
\end{array}
\]

B
\[
\begin{array}{ccc}
X & Y & X \\
\end{array}
\]

C
\[
\begin{array}{ccc}
X & Y & X \\
\end{array}
\]

Miss

Hit

Brought in, but unused before eviction

Fully associative cache
line size = 16 bytes
# lines = 16
Matrix Multiply

```
  X  Y
A: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
  X  Y
B: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
  X  Y
C: [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ] [ ]
```

- 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

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 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

\[ A \] \rightarrow \[ A^T \]

- 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

A

A^T

A

A^T
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 it's not in the cache
Average Memory Access Time (AMAT)

- AMAT = hit time + miss rate * miss penalty
AMAT Example

What is the AMAT of a system where

- Hit rate = 90%
- Hit time = 4 cycles
- Miss penalty = 20 cycles

\[
\text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty}
\]

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

\[
\text{AMAT} = 6 \text{ cycles}
\]
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

AMAT = hit time + miss rate * miss penalty

AMAT = 5 cycles + 0.25(24 cycles)

AMAT = 11 cycles
How Does Associativity Affect AMAT?

- Hit time as associativity increases?
  - Increases
  - Direct Mapped -> 2-way
    - Introduce a multiplexor to choose correct way
  - 2-way -> 4-way
    - Smaller increase than direct mapped -> 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

Registers → Cache → Main memory → Disk

Registers → L1 Cache → L2 Cache → Main memory → Disk
L2 Cache

- L2 is bigger than L1
  - Leads to higher hit rate
- L2 is accessed only if the requested data in 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 Cache Info

https://uops.info/cache.html#SKL

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**
  • \( \frac{\text{# hits at this level}}{\text{# accesses to this level}} \)

• **Global Hit rate**
  • \( \frac{\text{# hits at this level}}{\text{# total number of accesses}} \)
AMAT with 2-level Cache

\[
\text{AMAT} = \text{hit time} + \text{miss rate} \times \text{miss penalty}
\]

** all miss rates are local
AMAT with 2-level Cache

\[
AMAT = \text{hit time} + \text{miss rate} \times \text{miss penalty}
\]

\[
AMAT = L1 \ \text{hit time} + L1 \ \text{miss rate} \times L1 \ \text{miss penalty}
\]

L1 miss penalty = L2 hit time + L2 miss rate * L2 miss penalty

\[
AMAT = L1 \ \text{hit time} + L1 \ \text{miss rate} \times (L2 \ \text{hit time} + L2 \ \text{miss rate} \times L2 \ \text{miss penalty})
\]

** 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

AMAT = L1 hit time + L1 miss rate * (L2 hit time + L2 miss rate * L2 miss penalty)

AMAT = 4 + 0.25 * (6 cycles + 0.1 * 20 cycles)

AMAT = 6 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

\[
\text{AMAT} = \text{L1 hit time} + \text{L1 miss rate} \times (\text{L2 hit time} + \text{L2 miss rate} \times \text{L2 miss penalty})
\]
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

AMAT = L1 hit time + L1 miss rate * (L2 hit time + L2 miss rate * L2 miss penalty)

AMAT = 5 + 0.4 * (8 cycles + 0.05 * 40 cycles)

AMAT = 9 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...)