Cache Introduction
233 in one slide!

- The class consists roughly of 4 quarters: (Bolded words are the big ideas of the course, pay attention when you hear these words)
  1. You will build a simple computer processor
     Build and create state machines with data, control, and indirection
  2. You will learn how high-level language code executes on a processor
     Time limitations create dependencies in the state of the processor
  3. You will learn why computers perform the way they do
     Physical limitations require locality and indirection in how we access state
  4. You will learn about hardware mechanisms for parallelism
     Locality, dependencies, and indirection on performance enhancing drugs

- We will have a SPIMbot contest!
Compare and Contrast

I like to eat eggs for breakfast, so every Saturday, I buy two dozen eggs. Now that it’s getting cold outside, I put my shorts and t-shirts into storage and put my sweaters and pants into my closet.
Today’s Lecture

- Main memory is **HUGE** and sssllllloooowwww
- Use small but fast caches to access parts of main memory
- Temporal and spatial **locality** in code allow caches to improve performance
- **WARNING:** Caches require a lot of conceptual and actual overhead to implement
We have assumed that our CPUs can access memory twice in one cycle...we may have lied...

- We need increasingly more storage
- Increasing memory size comes at a cost
- But we need to maintain speed
- Pipelined CPUs and superscalar CPUs decrease CPI
To increase capacity, we lose speed and increase cost

<table>
<thead>
<tr>
<th>Storage</th>
<th>Speed</th>
<th>Cost</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static RAM</td>
<td>Fastest</td>
<td>Expensive</td>
<td>Smallest</td>
</tr>
<tr>
<td>Dynamic RAM</td>
<td>Slow</td>
<td>Cheap</td>
<td>Large</td>
</tr>
<tr>
<td>Hard disks</td>
<td>Slowest</td>
<td>Cheapest</td>
<td>Largest</td>
</tr>
</tbody>
</table>

- Fast memory is too expensive for most people to buy a lot of.
- Dynamic memory is slower than our pipeline stages
- Can’t perform \texttt{lw} or \texttt{sw} with dynamic memory and not stall!

<table>
<thead>
<tr>
<th>Storage</th>
<th>Delay</th>
<th>Cost/MB</th>
<th>Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static RAM</td>
<td>1-20 cycles</td>
<td>~$0.5</td>
<td>128KB-64MB</td>
</tr>
<tr>
<td>Dynamic RAM</td>
<td>100-200 cycles</td>
<td>~$0.01</td>
<td>10-100GB</td>
</tr>
<tr>
<td>Hard disks</td>
<td>10,000,000 cycles</td>
<td>~$0.00005</td>
<td>1-10TB</td>
</tr>
</tbody>
</table>
A cache provides quick access to a small amount of frequently used data

- Memory access speed increases overall, because we’ve made the common case faster.
  - Reads and writes to the most frequently used addresses will be serviced by the cache.
  - We only need to access the slower main memory for less frequently used data.
Caches are small memories located on the chip with the processor.

Main Memory is far away off chip this way.
Visual comparison of memory sizes

Reg File ↓ L1 Cache

Main memory
Problem: How do we predict what data should be in the small amount of cache?
Locality means that we will access data based on physical or temporal proximity

- **Temporal locality**: if a program accesses one memory address, there is a good chance that it will access the same address again.

- **Spatial locality**: if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.
Loops create temporal locality in the instructions we execute

- The loop body will be executed many times.
- Access those a few locations of the instruction memory repeatedly.
- For example:

```assembly
Loop: lw $t0, 0($s1)
      add $t0, $t0, $s2
      sw $t0, 0($s1)
      addi $s1, $s1, -4
      bne $s1, $0, Loop
```

- Each instruction will be fetched over and over again, once on every loop iteration.
Loops also create temporal locality in how we access data

- `sum` and `i` are repeatedly read and written.

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + f(i);
```

- Commonly-accessed variables can sometimes be kept in registers, but this is not always possible.
  - There are a limited number of registers.
  - You can’t take the address of a register (i.e., create a pointer to it)
  - There are situations where the data must be kept in memory, as is the case with shared or dynamically-allocated memory.
Because we increment PC by default to access the next instruction, most instructions exhibit spatial locality

```assembly
sub $sp, $sp, 16
sw $ra, 0($sp)
sw $s0, 4($sp)
sw $a0, 8($sp)
sw $a1, 12($sp)
```
Arrays and structs store data in adjacent memory addresses, creating spatial locality.

```c
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + a[i];
```

```c
employee.name = "Homer Simpson";
employee.boss = "Mr. Burns";
employee.age = 45;
```
The first time an address in main memory is accessed, the data is also copied to the cache.

- The next time that same address is read, we use the copy of the data in the cache *instead* of accessing the slower dynamic memory.
- First access is slow, but subsequent accesses are fast.
When the CPU accesses an address, it copies that address’s data and the data at several adjacent addresses into the cache

- Example: If the CPU accesses \(a[i]\), it may also copy \(a[i+1]\), \(a[i+2]\), and \(a[i+3]\) into the cache
- The first access of \(a[i]\) is slow, but the accesses of \(a[i+1]\), \(a[i+2]\), and \(a[i+3]\) may be faster
Compare and Contrast (revisited)

I like to eat eggs for breakfast, so every Saturday, I buy two dozen eggs and put them in my refrigerator.

Now that it’s getting cold outside, I put my shorts and t-shirts into storage and put my sweaters and pants into my closet.

What are some other examples of caches?
To improve performance, we want to maximize cache hits and minimize cache misses

- **Cache hit:** the cache contains the data we need when accessing memory
- **Cache miss:** the cache does not contain the data we need when accessing memory

Cache misses decrease performance because they mean we need to access the slower main memory.
Measure performance of a cache by the percentage of accesses that are hits or misses

- **Hit rate**: the percentage of memory accesses that are result in cache hits and can be serviced by the cache.

- **Miss rate**: \((1 - \text{hit rate})\) - the percentage of memory accesses that miss the cache and must be handled by the slower main RAM.

- Typical caches have a hit rate of 95% or higher.
Caches are divided into **blocks**

- The number of blocks in a cache is usually a power of 2.
- The **block index** of a cache is analogous to the **address** of main memory.
Cache blocks may contain multiple bytes of data to take advantage of spatial locality.
Four guiding questions for implementation

1. When we copy a block of data from main memory to the cache, where exactly should we put it?

2. How can we tell if a word is already in the cache, or if it has to be fetched from main memory first?

3. Eventually, the small cache memory might fill up. To load a new block from main RAM, we’d have to replace one of the existing blocks in the cache... which one?

4. How can write operations be handled by the memory system?
Q1: A direct-mapped cache maps each main memory address to exactly one cache block.
Use the least significant bits to identify block index (i.e., modulo number of cache blocks)

16-byte main memory

Memory Address 8-bit data
0000
0001
0010
0011
0100
0101
0110
0111
1000
1001
1010
1011
1100
1101
1110
1111

8-bit data
Block Index
00
01
10
11

4-byte cache

$6 \mod 4 = 2$

$15 \mod 4 = 3$
Quick look ahead: Larger cache blocks -> adjacent data moves together

Create an “array of arrays”

16-byte main memory

8-byte cache

Block offset
0
1

Block Index
0
1
2
3
Q2: How do we determine the memory address from the cache block index?

16-byte main memory

Memory Address

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

8-bit data

4-byte cache

Block Index

0 1 2 3
Store most-significant bits of the address (the tag) as additional data in the cache.
Combine the index and tag to recreate the main memory address

<table>
<thead>
<tr>
<th>Tag</th>
<th>8-bit data</th>
<th>Block Index</th>
<th>Tag + index = address</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td></td>
<td>00</td>
<td>01+00 = 0100</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>01</td>
<td>10+01 = 1001</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>10</td>
<td>10+10 = 1010</td>
</tr>
<tr>
<td>01</td>
<td></td>
<td>11</td>
<td>01+11 = 0111</td>
</tr>
</tbody>
</table>
The **valid bit** indicates whether our program has previously loaded data into the cache

- Valid bits start as 0
- Valid bit is set to 1 when data is loaded into a cache block

<table>
<thead>
<tr>
<th>Block Index</th>
<th>Valid bit</th>
<th>Tag</th>
<th>8-bit data</th>
<th>Tag + index = address</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>1</td>
<td>01</td>
<td>Red</td>
<td>01 + 00 = 0100</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>10</td>
<td>Blue</td>
<td>invalid</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>01</td>
<td>Green</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>01</td>
<td>Gray</td>
<td></td>
</tr>
</tbody>
</table>
The cache controller parses the memory address into a tag and index

- Lowest $k$ bits of the address become the block index
- Upper $(m - k)$ bits of the $m$-bit address become the tag
- Compare tags and check valid bit to determine a cache hit
The 2ns memory delay in our pipeline is only true for cache hits!

- Main memory is much too slow
- Cache misses must access main memory
- We assume that most memory accesses will be cache hits
  - ~95% cache hit rate
- Cache misses require longer stalls (or a different solution!)
On a cache miss, use the index to determine where to store, the tag is stored as data:

- The lowest $k$ bits of the address specify a cache block.
- The upper $(m - k)$ address bits are stored in the block’s tag field.
- The data from main memory is stored in the block’s data field.
- The valid bit is set to 1.