CSE 351

Caches
Before we start...

• A lot of people confused lea and mov on the midterm
  • Totally understandable, but it’s important to make the distinction

• The lea instruction is used to compute a number
  • Usually, this is an address that it is computing, as that is what it was designed to do
  • Very useful for adding some offset to a pointer, which can be used to calculate the address of some index in an array
  • However, it does NOT dereference the address that it computes

• The mov instruction also does some arithmetic with its arguments to resolve some address
  • However, it actually dereferences the address that it computes, and moves the value there into some register
Performance Bottlenecks

• How do we define “computer performance”
  • Instruction throughput, i.e. how many instructions can we execute per second?

• What factors affect this?
  • CPU architecture will give us an estimate for maximum instructions per second (IPS)
  • The CPU can only process data at the rate it receives the data, however
  • Reading from memory has become increasingly slow relative to the rate at which modern processors can execute instructions
  • Memory is a bottleneck!
What is a cache?

• A CPU cache is a small, fast region of memory that is actually on the same piece of silicon as the processor
  • Modern processors have as many as 4 caches, each progressively larger and slower than the next
  • They form a cache hierarchy

• Caches are orders of magnitude faster than the DRAM used for main memory

• However, they fill up quickly, so the CPU needs to be smart about how it uses its caches
Caches Improve Performance

• A common optimization in many systems is data caching

• Example: web browsers
  • In this scenario, the network latency is a bottleneck
    • Web browsers can’t load pages until they have received the data
  • Your web browser caches static files (images, .html, .css, .js, etc) that you have recently downloaded onto your hard drive
    • Relative to the network, your hard drive is speedy

• Processors use the same concept to improve performance, by caching data as well as instructions
Why do caches work?

• To make caches as effective as possible, a profile of the flow in/out of data is necessary
  • We want to make sure that we minimize the amount of data we cache while maximizing its effect
  • Cache more things => larger caches => slower => less effective

• In a processor, there are two common patterns that emerge in analyzing memory reads and writes
  • These patterns are described by temporal and spatial locality
  • Caches are designed to take advantage of both
Temporal Locality

• Temporal locality: accessing the same data repeatedly
• Almost all applications of caches take advantage of some degree of temporal locality
  • We save the most recently-used data, because it has the highest likelihood of being used next
• Programs are run sequentially, thus memory accesses are often bunched together
  • Memory accesses show clear signs of temporal
  • We declare variables, use them, and then move on
Temporal Locality Example

• Assume that this code runs like we expect (a, b, c, and result are in registers %rdi, %rsi, %rdx, and %rax respectively):

```c
int example(int* a, int b, int c) {
    int result = *a;
    result += b;
    result += c;
    result += *a;
    return result
}
```

• We access memory twice, reading whatever is stored at address a

• The cache would have stored this data after the first read, so when we read it again, we just have to go to the cache to get the data

• If there were many operations in between the two accesses, then *a might no longer be in the cache, but this code exhibits temporal locality by accessing *a close together in time
Spatial Locality

• Spatial locality: using data that is physically close together

• This is a huge pattern when looking at memory accesses
  • What is an instance in which this might occur during the execution of a program?

• CPU caches take advantage of this by storing more than we need, on the assumption that we will want the extra data soon
Spatial Locality Example

```c
int example(int* array, int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i) {
        sum += array[i];
    }
    return sum;
}
```

- This code accesses 4 bytes of memory at a time
- The CPU will grab more than that though; depends on the cache size
  - If it loads 16-byte blocks into the cache, then we will have to go to memory for array[0], but array[1], array[2], and array[3] will be in the cache already when we need to read those values
Cache Architecture

- Each cache entry is called a “line”
  - The size of the line in bytes is a power of 2
  - Usually on the order of 16 bytes, but that can change
- There are many lines in a cache
  - The number of lines is also a power of 2
- To get the total size of a cache, we multiply the size of a line by the number of lines
- The cache to the right has 8 lines of 16 bytes each for a total of 128 bytes
Cache Interface

• The cache interface is the same as the memory interface
• When we reference memory using an instruction, the CPU will first look in the cache for that data
  • The cache is NOT directly addressable
  • For example, there is no instruction that allows me to access CACHE[4]
  • Instead, we ask for the data at 0x7FFFFFF0, and the CPU checks the cache for us
  • If it is not there (called a cache MISS), then it will grab the data from memory
• To make the interfaces the same, we use the same addresses we use to access memory to index into the cache
Cache Indexing

- We break up an address into the following components when we wish to index into a cache:

  $0x7F04$

  
  - **t** Tag Bits
  - **s** Index Bits
  - **b** Offset Bits

- In this example, we are in a 16-bit address space
  - $t + s + b$ will always equal the number of bits in our address space

- How do we choose these numbers though?
Determining b

• The value $b$ defines the number of bits used to represent an offset into a line of the cache

• As we said before, caches are broken into lines, where each line is some power of 2 in size
  • However, we want our cache to be able to access individual bytes

• Let’s say our cache lines are 16 bytes in size
  • How many bits do we need to represent every possible byte offset into the block below?
  • $b = \log_2 B$, where $B =$ Block size = size of data in cache line
Determining $s$

• The value $s$ defines the number of bits needed to represent every index in the cache

• We now introduce another concept: cache sets

• There are different types of caches, and they differ based on the number of lines in each set
  • Direct-mapped cache: one line per set
  • N-way set-associative cache: N lines per set
  • Fully associative cache: all lines are in the same set
  • You will discuss these more in lecture

• When we talk about cache indexes, we are referring to particular sets in the cache, not lines
Determining s (cont.)

Direct-mapped cache
Determining s (cont.)

Since this cache has two entries per set, it is a 2-way associative cache.

2-way set associative cache
Determining $s$ (cont.)

• In order to find $s$, we need to know the number of sets in the cache
  • This value is rarely given, it needs to be calculated
• Often you are given the size of a cache, along with its block size and associativity
  • The block size is the size of the data each line holds
  • The associativity is the number of lines in each set
• To compute the number of sets, simply divide the cache size by the block size, then divide again by the associativity

\[
\text{Cache size (Bytes)} \times \frac{1}{\text{Bytes}} \times \frac{1}{\text{Lines}} = \text{Sets} = S
\]
Determining \( s \) (cont.)

- Once you have the number of sets, you simply need to take \( \log_2 S \)
- This will give you the number of bits needed to represent the values \((0, S - 1)\)
- Let's try some examples. Find \( b \) and \( s \) for:
  - Example Cache 1:
    - Size: 128 bytes
    - Associativity: 1
    - Block size: 16
  - Example Cache 2:
    - Size: 64 bytes
    - Associativity: 4
    - Block size: 4
  - Example Cache 3:
    - Size: 64 bytes
    - Associativity: full
    - Block size: 16
Determining t

- Finally, we need to know the number of bits to use for the tag.
- Once you know \( b \) and \( s \), finding \( t \) is easy, because it is all the higher-order bits remaining.
  - Remember? \( b + s + t = \) width of address space in bits.
- What is the tag used for?
  - Because the address space is huge, there are many, many addresses that map to the same set in the cache.
  - In order to determine if the data in a cache line belongs to a particular address, the tags of the data will be compared because they are always unique for two different addresses in the same set.
    - Think about it, if they are in the same set, with the same tag...then the bits making up the address are the same, meaning the addresses are the same.
Putting it all together

• When we try to reference some address, the CPU will first check to see if the data at that address has been placed in the cache
  • To index into the cache, it will use that address
  • It does not calculate t, s, and b; it already knows what they should be, because the CPU and cache are integrated pieces of hardware
  • It uses the s bits of the address to locate the correct set
  • It uses the t bits of the address to compare tags
  • If it finds a matching tag with one of the lines in that set, then it will use the b bits of the address to grab the correct byte from the block in that line

• You just received a crash course in caches

• Luckily, you will be covering this in lecture extensively, so this was meant to be more of a primer