Notes and Updates

- Office hours for Dr. Simon: Tues moved to Thurs 12-1:30
- Last homework is up!
- Today:
  - TA evaluations (just for 141)
  - Finish up exceptions for pipelines
  - More caches
- Thursday:
  - Caches and performance!
- Questions?
  - How did last lecture go?

Exceptions and Pipelines:

More Complexity

- Exceptions (overflow, invalid instruction, etc) also must be caught in a pipelined world
  - Save the PC (into EPC)
  - Save the Cause
  - Transfer control to Exception Handling routine (set PC to 0)
Dealing with Exceptions in a Pipeline

- Save the PC of the offending instruction (they use PC+4 in book, don’t worry about that detail too much)
- FLUSH the instructions following the offending instruction
- Start fetching from address “0” (or wherever).
Pipelines with more types of exceptions

- In pipelines that support more types of exceptions, it can be difficult to figure out which instruction to handle
  - What if multiple exceptions (from different instructions in different stages) happen at the same time?
  - Choose which to handle first.
- Some architectures don’t promise to get it perfectly correct
  - Imprecise interrupts
- Alternative: precise interrupts

Chapter 7 Caches:
Issues we touched on

- How data moves from memory to cache
  - Benefits: Temporal locality
- What to do when cache is full
  - Replacement policies
- Placement options for where data can “go” in cache
  - Direct-mapped, Set-associative, Fully-associative
- Moving “lines”/“blocks” into cache
  - Benefit: Spatial locality
- Writing values in a code
  - Cache/MM out of synch with registers
  - Write back policies in caches
Cache Vocabulary

- cache hit: an access where data is already in cache
- cache miss: an access where data isn't in cache
- Hit time: time to access the cache
- miss penalty: time to move data from further level to closer, then to cpu
- hit rate: percentage of time the data is found in the cache
- miss rate: (1 - hit rate)

Cache Vocabulary

- cache block size or cache line size: the amount of data that gets transferred on a cache miss.
- instruction cache (I-cache): cache that can only hold instructions.
- data cache (D-cache): cache that can only hold data.
- unified cache: cache that holds both data & instructions.

A typical processor today has separate "Level 1" I- and D-caches on the same chip as the processor (and possibly a larger, unified "L2" on-chip cache), and larger L2 (or L3) unified cache on a separate chip.
Cache Issues

On a memory access -
- How does hardware know if it is a hit or miss?

On a cache miss -
- where to put the new data?
- what data to throw out?
- how to remember what data is where?

A simple cache

<table>
<thead>
<tr>
<th>address trace</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
</tr>
<tr>
<td>24</td>
<td>00011000</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
</tbody>
</table>

- A cache that can put a line of data anywhere is called
- The most popular replacement strategy is
A simpler cache

Address trace:

<table>
<thead>
<tr>
<th>Address</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>20</td>
<td>00010100</td>
</tr>
<tr>
<td>24</td>
<td>00011000</td>
</tr>
<tr>
<td>12</td>
<td>00001100</td>
</tr>
<tr>
<td>8</td>
<td>00001000</td>
</tr>
<tr>
<td>4</td>
<td>00000100</td>
</tr>
</tbody>
</table>

- A cache that can put a line of data in exactly one place is called
- What’s the tag in this case?

Direct-mapped cache

- Keeping track of when cache entries were last used (for LRU replacement) in big cache needs lots of hardware and can be slow.
- In a direct mapped cache, each memory location is assigned a single location in cache.
  - Usually* done by using a few bits of the address

* Some machines use a pseudo-random hash of the address
  But it is DETERMINISTIC!
Or another way to look at it:
an 8 entry cache

Compare and Contrast Direct Mapped and Fully Associative Caches

- What were the “hit rates” for each cache design in our example?

- Which has better temporal locality in our example?

- Which cache would likely have a faster hit time?
Choosing bits for the index

If line length is n Bytes, the low-order $\log_2 n$ bits of a Byte-address give the location of address within a line.

The next group of bits is the address -- this ensures that if the cache holds X bytes, then any block of X contiguous Byte addresses can co-reside in the cache. (Provided the block starts on a cache line boundary.)

The remaining bits are the tag.
Anatomy of an address:

```
  +-----------+---------+---------+
  |           |         |         |
  |           |         |         |
  +-----------+---------+---------+
```

Direct Mapped Caches vs. Fully Associative Caches

- Direct mapped was “easy” to implement
  - Each address can go in exactly one “spot” (line) in cache
  - But sometimes addresses used “together” (4,20,4,20) mapped to the same cache location, causing

```
Address 4 and 20 both map to same line in a 4-entry cache
```

- Fully Associative always kicked out least recently used address
Compromise: A set-associative cache

address string:
4 00000100
8 00001000
12 00001100
4 00001000
8 00001000
20 00010100
4 00001000
8 00001000
20 00010100
24 00011000
12 00001100
8 00001000
4 00001000

- A cache that can put a line of data in exactly \( n \) places on a line is:
- The cache lines that share the same index are a

A Better Cache Design

- Direct mapped caches are simpler
  - Less hardware; possibly faster
- Fully associative caches usually have fewer misses.
- Set associative caches try to get best of both.
  - An index is computed from the address
  - In a "k-way set associative cache", the index specifies a set of \( k \) cache locations where the data can be kept.
    - \( k=1 \) is direct mapped.
    - \( k= \)cache size (in lines) is fully associative.
  - Use LRU replacement (or something else) within the set

index tag data tag data tag data tag data tag data tag data

Effects of Cache Associativity

Different lines show different cache sizes

**Longer Cache Lines**

<table>
<thead>
<tr>
<th>address string</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 00000100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 00001000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 00011000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 00000100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 00001000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 00010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 00000100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 00001000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20 00010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24 00011000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 00001100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 00001000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 00000100</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Large cache blocks take advantage of spatial locality.
- Longer cache blocks require less tag space

32 bit address
Disadvantages of longer cache lines

- Spatial locality is GOOD! Let’s make cache lines 128 words (array elements) long!
  - What if I make a bunch of 30 element arrays?

- Transfer time affected...
  - Different memory types had different “access times” (for 1 word to be “accessed”)
    • To get “more memory” will take longer.

Line Filling Options: Requested Word First

- When we request one “load word” all data for that cache line is brought in - even if the line holds more than one word
  - Order differs by memory design
  - Assume memory can be brought in at rate of 1ns/word

- EXAMPLE:

  ```java
  int [] a = new int[10];
  ... a[1];
  ... a[0];
  ... a[3];
  ... a[2];
  ```
Line/Block Size and Miss Rate

Rule of thumb #1: block size should be less than square root of cache size.

Rule of thumb #2: block size should consider likely programming uses.

2-way set associative cache in action

Sequence of memory references: 24, 20, 28, 12, 20, 08, 44, 04,

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>data</th>
<th>tag</th>
<th>data</th>
<th>address</th>
<th>bin</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
<td>011000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>010100</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28</td>
<td>011100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td>001100</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>010100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>08</td>
<td>001000</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44</td>
<td>101100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>04</td>
<td>000100</td>
<td></td>
</tr>
</tbody>
</table>
**Larger line/block size in action**

Sequence of memory references: 24, 20, 28, 12, 20, 08, 44, 04,

<table>
<thead>
<tr>
<th>index</th>
<th>tag</th>
<th>8 Bytes of data</th>
<th>address</th>
<th>bin</th>
<th>index</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>24</td>
<td>010000</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>20</td>
<td>010100</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>28</td>
<td>011100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>12</td>
<td>001100</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>20</td>
<td>010100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>08</td>
<td>001000</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>44</td>
<td>101100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>04</td>
<td>000100</td>
<td></td>
</tr>
</tbody>
</table>

**Cache Size: Which of these things is not like the other?**

<table>
<thead>
<tr>
<th>tag</th>
<th>1 word data</th>
<th>time since last ref</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>2 words data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>1 word data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tag</th>
<th>1 word data</th>
<th>tag</th>
<th>1 word data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cache Parameters

Cache size =

Note: tag bits, LRU bits, not counted as part of "size", still matter in design
-128 blocks, 32-byte block size, direct mapped, size =
-128 KB cache, 64-byte blocks, 512 sets, associativity =

Putting it all together

64 KB cache, direct-mapped, 32-byte cache line/block
A set associative cache

32 KB cache, 2-way set-associative, 16-byte blocks

128 KB cache, 4-way set-associative, 8-byte blocks

This picture doesn't show the "most recent" bits (need one bit per set)

This picture doesn't show the "most recent" bits (?? bits per set)
Key Points

- Caches give illusion of a large, cheap memory with the access time of a fast, expensive memory.
- Caches take advantage of memory locality, specifically temporal locality and spatial locality.
- Cache design presents many options (block size, cache size, associativity) that an architect must combine to minimize miss rate and access time to maximize performance.
  - Good design requires COMPROMISE!
    - Block size
    - Associativity
    - Temporal and Spatial Locality

Loads, caches, and our pipeline

- Hazard detection only checks for “next instruction” to see if it should bubble.

- We MUST have all load values back in 1 cycle
  - MEM phase - (in case an instruction 2 after the load needs the result)
  - Pipeline stalls on a cache miss

That's why cache miss rates matter sooooo much!
Dealing with Stores

- Stores must be handled differently than loads, because...
  - they don’t necessarily require the CPU to stall
    - Who “needs” the value?
  - they change the content of cache
    Creates a memory consistency question ... how do you ensure memory gets the correct value?

Policy decisions for stores

- Do you keep memory and cache identical?
  - write-through cache: all writes go to both cache and main memory
  - write-back cache: writes go only to cache. Modified cache lines are written back to memory when the line is replaced.
Policy decisions for stores

- Do you make room in cache for store miss?
  - write-allocate: on a store miss, bring target line into the cache.
  - write-around: on a store miss, ignore cache

Dealing with stores

- On a store hit, write the new data to cache.
  - In a cache, write the data immediately to memory.
  - In a cache
    - Mark the line as dirty.
      - means cache has correct value, but memory doesn’t
    - On any subsequent cache miss in a write-back cache, if the line to be replaced in the cache is dirty, write it back to memory.

- On a store miss,
  - In a cache
    - Initiate a cache block load from memory.
  - In a cache
    - Write directly to memory.
Cache Alignment

- A cache line is all the data whose address share the tag and index.
  
  Example: Suppose offset if 5 bits,
  - Bytes 0-31 form the first cacheline
  - Bytes 32-63 form the second, etc.
  - When you load location 40, cache gets Bytes 32-63

- This results in
  - no overlap of cache lines
  - easy to find if address is in cache (no additions)
  - easy to find the data within the cache line

- Think of memory as organized into cache-line sized pieces (because in reality, it is!)

Can a word overlap two cache lines?

- i.e. must all integer addresses end in 00?

- Depends on the architecture ...
  - Some require words to be word-aligned ...
    - and double words to be double-word aligned
    - Every load and store is to a single cacheline
  - Others allow data to span cache lines
    - Can be two cache misses for a single reference!
    - Requires more shifting logic
    - But allows more compact data structures.
Two memory allocation schemes

- No alignment required
- Alignment required

```c
struct{
    char valid;
    int tag;
} cache_line;

cache = new cache_line[1024];
```

---

Take a step back...

ET = IC * CPI * CT?

- Can our old execution time equation support our new pipelined world?
- What parts of the equation are “unphased” - “can be constantly represented” by pipelining and cache?
- What effects will differ for different instructions?