Notes and Updates

- Office hours for Dr. Simon: Tues 3:30-5 and Wed 12-1:30
- Last homework is up! Due Tues.
- Today:
  - Finish various cache types
  - Performance
  - Stores and caches
  - Alignment
- Tues:
  - Multilevel Caches
  - Virtual Memory.
- Questions?

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)
2 different 4-way Set Associative Caches

Line/block size

Members of the same cache set (4-way)

Effects of Cache Associativity

Different lines show different cache sizes
Longer Cache Lines

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

<table>
<thead>
<tr>
<th>tag</th>
<th>data</th>
</tr>
</thead>
</table>

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

<table>
<thead>
<tr>
<th>Miss rate (%)</th>
<th>1 KB</th>
<th>8 KB</th>
<th>16 KB</th>
<th>64 KB</th>
<th>256 KB</th>
</tr>
</thead>
<tbody>
<tr>
<td>40%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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 011000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20 010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28 011100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12 001100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20 010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>08 001000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>44 101100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>04 000100</td>
<td></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 011000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>20 010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>28 011100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>12 001100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>20 010100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>08 001000</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td>44 101100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>04 000100</td>
<td></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>
<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>
<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>
<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 lines, 32-byte block size, direct mapped, size =
-128 KB cache, 64-byte block size, 512 lines, associativity =
Putting it all together (pg 478)

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

A set associative cache

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

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

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

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!

---

Measuring Cache Performance

- Book says:
  CPU time =
  \[(\text{CPU execution clock cycles} \times \text{Memory stall clock cycles}) \times \text{Clock cycle time}\]

- Before we discuss memory, let’s review CPU execution clock cycles
Take a step back...
$ET = IC \times CPI \times CT$?

- 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?

Calculating “CPU cycles”

- $TCPI = \text{Total CPI}$
- $BCPI = \text{Base CPI} = \text{CPI assuming perfect memory}$
- $MCPI = \text{Memory CPI} = \text{cycles waiting for memory per instruction}$

- $PSPI = \text{pipeline stalls per instruction}$
- $BSPI = \text{branch hazard stalls per instruction}$
Calculating “Memory Stall” cycles

- this assumes we stall the pipeline on both read and write misses, that the miss penalty is the same for both, that cache hits require no stalls.
- If the miss penalty or miss rate is different for I-cache and D-cache (which is common), then
  \[
  \text{MCPI} = \\
  \%\text{InstMemRefs}\times\text{InstMissRate}\times\text{InstMissPenalty} + \\
  \%\text{DataMemRefs}\times\text{DataMissRate}\times\text{DataMissPenalty}
  \]

Cache Performance (do examples on pg 494-495)

- Instruction cache miss rate of 4%, data cache miss rate of 9%, BCPI = 1.0, 20% of instructions are loads and stores, miss penalty = 10 cycles, TCPI = ?

\[
\text{MCPI} = \%\text{InstMemRefs}\times\text{InstMissRate}\times\text{InstMissPenalty} + \\
\%\text{DataMemRefs}\times\text{DataMissRate}\times\text{DataMissPenalty}
\]
Cache Performance: For your practice

- Unified cache, 25% of instructions are loads and stores, BCPI = 1.2, miss penalty of 10 cycles. If we improve the data miss rate from 10% to 4% (e.g. with a larger cache), how much do we improve performance?

- BCPI = 1, miss rate of 20% for data, 2% for instructions, 20% loads, miss penalty 20 cycles (both instruction and data cache). What is the speedup from doubling the cpu clock rate?

Can you calculate a “real” Base CPI?

- See some of the last week's recommended problems
- Things like: Suppose you have 1000 instructions composed of a repeated pattern of a load followed by and add that depends on that load. Additionally, the next load depends on the add directly before it.
- What is the Base CPI? (i.e. CPI for our 5 stage pipeline with the forwarding that we developed).
Can you do?

- Can you calculate a “real” Base CPI?
  - Given a code with a loop - and a misprediction rate for that loop. (ie the branch misses 50% of the time).

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.

- 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 \textbf{cache}, write the data immediately to memory.
  - In a \textbf{cache}
    • Mark the line as \textit{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 \textbf{cache},
    • Initiate a cache block load from memory.
  - In a \textbf{cache},
    • Write directly to memory.

Cache Alignment

\begin{center}
\begin{tabular}{l|c|c|c}
memory address: & tag & index & offset \\
\hline
\end{tabular}
\end{center}

• 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

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

cache = new cache_line[1024];
```

- Alignment required

```plaintext
    tag  1 word data
    M[0-3]  
    M[4-7]  
    M[8-11] 
    M[12-15] 
    M[16-19] 
    M[20-23] 
    M[24-27] 
    M[28-31] 
```
Three types of cache misses (page 543)

- **Compulsory misses**
  - number of misses needed to bring every cache line referenced by program into an infinitely large cache.

- **Capacity misses**
  - number of misses in a fully associative cache of the same size as the cache in question minus the compulsory misses.

- **Conflict misses**
  - number of misses in actual cache minus number there would be in a fully-associative cache of the same size.

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Three types of cache misses

Total misses = (Compulsory + Capacity + Conflict) misses

- **Example: 16-byte, direct-mapped, 1 word per cacheline**
  - Reference sequence: 4, 8, 12, 4, 8, 20, 4, 8, 20, 24, 12, 8, 4
    - Compulsory misses:
    - Capacity misses:
    - Conflict misses:

- **Example 16-byte, direct-mapped, 2 words per cache line**
  - What is main difference?
So, then, how do we decrease...

- Compulsory misses?
- Capacity misses?
- Conflict misses?