Review Problem 34

- What forwarding happens on the following code?

```
LDUR X0, [X1, #0]
ADD X2, X3, X3
ORR X31, X0, X4
CBNZ X2, END
SUB X5, X31, X2
```

- Cycle 4: X0 forwarded from LDUR's mem stage.
- Cycle 5: X2 forwarded from ADD's mem stage.
- X2 is sent from ADD to SUB via Regfile.
- Never forward X31.
- X2: CBNZ doesn't write the Regfile.
- CBNZ Reg WE = 0.
Memory Hierarchy: Caches, Virtual Memory

Readings: 5.1-5.4, 5.8

Big memories are slow
Fast memories are small

Need to get fast, big memories

Computer
Processor
Control
Datapath
Memory
Devices
Input
Output

DRAM
4GB-16GB
Random Access Memory

Dynamic Random Access Memory (DRAM)
- High density, low power, cheap, but slow
- Dynamic since data must be "refreshed" regularly
- Random Access since arbitrary memory locations can be read

Static Random Access Memory
- Low density, high power, expensive
- Static since data held as long as power is on
- Fast access time, often 2 to 10 times faster than DRAM

<table>
<thead>
<tr>
<th>Technology</th>
<th>Access Time</th>
<th>Cost/Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRAM</td>
<td>1-7 cycles</td>
<td>10,000x</td>
</tr>
<tr>
<td>DRAM</td>
<td>100 cycles</td>
<td>200x</td>
</tr>
<tr>
<td>Flash</td>
<td>10,000 cycles</td>
<td>15x</td>
</tr>
<tr>
<td>Disk</td>
<td>10,000,000 cycles</td>
<td>1x</td>
</tr>
</tbody>
</table>
The Problem

Cost vs. Performance
  Fast memory is expensive
  Slow memory can significantly affect performance

Design Philosophy
  Use a hybrid approach that uses aspects of both
  Keep frequently used things in a small amount of fast/expensive memory
    “Cache”
  Place everything else in slower/inexpensive memory (even disk)
  Make the common case fast
Locality

Programs access a relatively small portion of the address space at a time

```c
char *index = string;
while (*index != 0) { /* C strings end in 0 */
    if (*index >= 'a' && *index <= 'z')
        *index = *index + ('A' - 'a');
    index++;
}
```

Types of Locality

Temporal Locality – If an item has been accessed recently, it will tend to be accessed again soon

Spatial Locality – If an item has been accessed recently, nearby items will tend to be accessed soon

Locality guides caching
The Solution

By taking advantage of the principle of locality:

- Provide as much memory as is available in the cheapest technology.
- Provide access at the speed offered by the fastest technology.

<table>
<thead>
<tr>
<th>Name</th>
<th>Register</th>
<th>Cache</th>
<th>Main Memory</th>
<th>Disk Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Speed</td>
<td>1 cycle</td>
<td>1-7 cycles</td>
<td>100 cycles</td>
<td>10,000 cycles</td>
</tr>
<tr>
<td>Capacity</td>
<td>1x (norm.)</td>
<td>64-4Kx</td>
<td>4Mx</td>
<td>1Gx</td>
</tr>
</tbody>
</table>
Cache Terminology

**Block** – Minimum unit of information transfer between levels of the hierarchy
   - Block addressing varies by technology at each level
   - Blocks are moved one level at a time

**Upper vs. lower** level – “upper” is closer to CPU, “lower” is further away

**Hit** – Data appears in a block in that level
   - **Hit rate** – percent of accesses hitting in that level
   - **Hit time** – Time to access this level
     - Hit time = Access time + Time to determine hit/miss

**Miss** – Data does not appear in that level and must be fetched from lower level
   - **Miss rate** – percent of misses at that level = (1 – hit rate)
   - **Miss penalty** – Overhead in getting data from a lower level
     - Miss penalty = Lower level access time + Replacement time + Time to deliver to processor
     - Miss penalty is usually MUCH larger than the hit time
Cache Access Time

Average access time

Access time = (hit time) + (miss penalty)x(miss rate)

Want high hit rate & low hit time, since miss penalty is large

Average Memory Access Time (AMAT)

Apply average access time to entire hierarchy.
Cache Access Time Example

<table>
<thead>
<tr>
<th>Level</th>
<th>Hit Time</th>
<th>Hit Rate</th>
<th>Access Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1 cycle</td>
<td>95%</td>
<td>1 + .05(65) = 4.25</td>
</tr>
<tr>
<td>L2</td>
<td>10 cycles</td>
<td>90%</td>
<td>10 + .1(550) = 65</td>
</tr>
<tr>
<td>Main Memory</td>
<td>50 cycles</td>
<td>99%</td>
<td>50 + .01(50,000) = 550</td>
</tr>
<tr>
<td>Disk</td>
<td>50,000 cycles</td>
<td>100%</td>
<td>50,000</td>
</tr>
</tbody>
</table>

Note: Numbers are local hit rates – the ratio of access that go to that cache that hit (remember, higher levels filter accesses to lower levels)
Review Problem 37

- If you can speed up any level’s hit time by a factor of two, which is the best to speed up?

<table>
<thead>
<tr>
<th>Level</th>
<th>Hit Time</th>
<th>Hit Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>1 cycle</td>
<td>95%</td>
</tr>
<tr>
<td>L2</td>
<td>10 cycles</td>
<td>90%</td>
</tr>
<tr>
<td>Main Memory</td>
<td>50 cycles</td>
<td>99%</td>
</tr>
<tr>
<td>Disk</td>
<td>50,000 cycles</td>
<td>100%</td>
</tr>
</tbody>
</table>

\[
AMAT = 1 + 0.05 \times 10 + 0.05 \times 1 \times 50 + 0.05 \times 1 \times 0.1 \times 50,000
\]

\[
= 1 + 0.5 + 2.5 + 2.5
\]

= 4.25

Speed up

Disk
Handling A Cache Miss

Processor expects a cache hit (1 cycle), so no effect on hit.

Instruction Miss
1. Send the original PC to the memory
2. Instruct memory to perform a read and wait (no write enables)
3. Write the result to the appropriate cache line
4. Restart the instruction

Data Miss
1. Stall the pipeline (freeze following instructions)
2. Instruct memory to perform a read and wait
3. Return the result from memory and allow the pipeline to continue
Exploiting Locality

Spatial locality
Move blocks consisting of multiple contiguous words to upper level

Temporal locality
Keep more recently accessed items closer to the processor
When we must evict items to make room for new ones, attempt to keep more recently accessed items
Cache Arrangement

How should the data in the cache be organized?

Caches are smaller than the full memory, so multiple addresses must map to the same cache “line”

- **Direct Mapped** – Memory addresses map to particular location in that cache
- **Fully Associative** – Data can be placed anywhere in the cache
- **N-way Set Associative** – Data can be placed in a limited number of places in the cache depending upon the memory address
Direct Mapped Cache

4 byte direct mapped cache with 1 byte blocks
Optimize for spatial locality (close blocks likely to be accessed soon)

Memory Address

0 1 2 3 4 5 6 7 8 9 A B C D E F

Address M

Cache Address

<cache index>
Finding A Block

Each location in the cache can contain a number of different memory locations:
Cache 0 could hold 0, 4, 8, 12, ...

We add a **tag** to each cache entry to identify which address it currently contains.

What must we store?

- **Valid bit**
- **Tag**

Tag: all of the address bits that are **NOT** part of cache index.

Valid bit: 1 → good data
0 → garbage

* or byte select
Cache Tag & Index

Assume $2^9$ byte direct mapped cache with 1 byte blocks

Cache Tag = 57

Cache Index = 03
Cache Access Example

Assume 4 byte cache

Access pattern:

<table>
<thead>
<tr>
<th>Tag 1</th>
<th>Tag 2</th>
<th>Tag 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>011</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>110</td>
<td>110</td>
</tr>
<tr>
<td>011</td>
<td>001</td>
</tr>
</tbody>
</table>

Data: [Em[513], Em[563], Em[257]]