Datapath Q&A

What is the role of the ALU in lw, sw, lb, and sb?

Instruction opcode tells vs we have an lw instruction.
Your microprogram configures the ALU to perform addition.

How should we test lb and sb?

Probably just a very simple test for now.
Dilemma: we want memory to be both fast and large. Large memories are inherently slower than small ones. Memory trade-off:

Fast and Small $\rightarrow$ Large and Slow

Why do we have this trade-off?

- Small memories can be closer to the datapath, so there's less travel time. The physical size of the memory itself can slow it down too.
- Decoding/looking up entries in large memory takes longer.
- Large memories usually use less fast technology like DRAM, while small memory can use SRAM.

The fix: caching
Caching

Any time the datapath accesses memory, look in the cache first.

Cache hit: the cache contained the memory location we were looking for.

Cache miss: the cache did not contain the location we needed.

What makes caching effective?

Temporal locality - when a program accesses a location in memory, it is likely to access that location again soon.

Spatial locality - when a program accesses a location in memory, it is likely to access nearby locations soon.

Caching Policy:

- When the datapath accesses a memory location, keep it in the cache - this exploits temporal locality.
- When we cache one location, also cache nearby locations - this exploits spatial locality.
Cache Design: Fully Associative Cache

Address:

\[
\begin{array}{c}
\text{tag} \\
\text{offset}
\end{array}
\]

\[
\begin{array}{c}
15 \ldots 4 \ 3 \ldots 0
\end{array}
\]

\[
\text{cache line} \leftarrow 4 \text{ bit offset}
\]

When we store data in the cache, we also store nearby data. A cache line is a collection of adjacent bytes whose addresses have the same tag.

A cache line is a sequence of bytes we cache together, in this case it is 16 bytes \((2^4)\).

Cache Structure

\[
\begin{array}{lll}
\text{tag} & \text{data} & \text{compulsory misses} \\
1 & 0x111 \ldots \ldots & \text{(never accessed before)} \rightarrow \text{M} \\
? & 0x555 \ldots \ldots & \text{LUV} \rightarrow \text{M} \\
? & 0x333 \ldots \ldots & \text{LUV} \rightarrow \text{M} \\
? & 0x444 \ldots \ldots & \text{LUV} \rightarrow \text{M}
\end{array}
\]

Cache eviction: When we remove a cached entry to make room for a new cache line.

\[
\begin{array}{c}
0x1110 \\
0x2224 \\
0x1118 \\
0x3330 \\
0x4444 \\
0x5550 \\
0x6660 \rightarrow \text{Okylllo}
\end{array}
\]

capacity miss
Cache Design: Direct Mapped Cache

Address: \[ \begin{array}{c|c|c|c} \text{tag} & \text{index} & \text{offset} \end{array} \]
\[ 15 \ldots 7 \ldots 4 \ldots 0 \]

For a given address, a direct-mapped cache has exactly one place to cache the data loaded from that address.

4 index bits \( \rightarrow \) 16 entries in the cache \( (2^4) \)

```
index  valid  tag  data
0      1 aggregating 1077.
1      0
2      1 aggregating 1022.
3
5      1
```

Advantages over fully-associative:
- no LRU logic required
- large cache still has simple lookup logic (scales up)

Disadvantages relative to fully-associative:
- accessing two things with same index and repeated evictions (thrashing)
- direct mapped caches don't do as good a job at exploiting temporal locality
Cache Design: Set-Associative Cache

- **Direct Mapped**: 1 set, fully associative cache
- **Set Associative**: \( M \) sets of \( \frac{N}{M} \) entries
- **Fully Associative**: 1 set of \( N \) entries

For a cache with \( N \) entries, the address is divided into:

- \( 2middleware \) bits for index
- \( 2way \) set associative cache

### Example

<table>
<thead>
<tr>
<th>Set</th>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
<th>LRU</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Set 0**: Valid = 1, Data = 0x11, LRU = 1
- **Set 1**: Valid = 1, Data = 0x55, LRU = 0
- **Set 15**: Valid = 1

- **Eviction**: 0x2214 moved to Set 0 (Set 1)
- **Access**: 0x1104, 0x7700, 0x1100, 0x5500, 0x1100