What should we do to this code to run it on a CPU?

With delay slots?
Memory Hierarchy: Caches, Virtual Memory

Need to get fast big memories

Fast memories are small
Big memories are slow

Readings: 5.1, 5.4, 5.8
<table>
<thead>
<tr>
<th>Cost/Capacity</th>
<th>Access Time</th>
<th>Technology</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>10,000,000 cycles</td>
<td>Disk</td>
</tr>
<tr>
<td>15x</td>
<td>10,000 cycles</td>
<td>Flash</td>
</tr>
<tr>
<td>200x</td>
<td>100 cycles</td>
<td>DRAM</td>
</tr>
<tr>
<td>10,000x</td>
<td>1-7 cycles</td>
<td>SRAM</td>
</tr>
</tbody>
</table>

Fast access time, often 2 to 10 times faster than DRAM.

Static since data held as long as power is on

Low density, high power, expensive

Random Access Memory

Random Access Memory (RAM)

Dynamically accessed, must be "refreshed" regularly

High density, low power, cheap, but slow

Dynamic Random Access Memory (DRAM)
Make the common case fast.

Place everything else in slower/inexpensive memory (even disk).

Keep frequently used things in a small amount of fast/expensive memory.

Use a hybrid approach that uses aspects of both.

Design Philosophy:

Slow memory can significantly affect performance.

Fast memory is expensive.

Cost vs. Performance.

The Problem.
Locality guides caching

Spatial locality - If an item has been accessed recently, nearby items will tend to be accessed again soon.

Temporal locality - If an item has been accessed recently, it will tend to be accessed soon again.

Types of Locality:

```c
// (x, y, z) = index
if (x > y) x = y;

/* C strings end in 0 */
char* index = string;
```
Provide access at the speed offered by the fastest technology.
Provide as much memory as is available in the cheapest technology.
By taking advantage of the principle of locality.

The Solution
Miss penalty is usually MUCH larger than the hit time.

Miss penalty = Lower level access time + Replacement time + Time to deliver to processor.

Overhead in getting data from a lower level. Miss penalty = percent of misses at that level = (1 − hit rate).

Miss is Data does not appear in that level and must be fetched from lower level.

Hit time = Access time + Time to determine hit/miss.

Hit time = Time to access this level.

Hit rate = percent of accesses hitting in that level.

Hit − Data appears in a block in that level.

Upper vs. lower level − “upper” is closer to CPU. “Lower” is further away.

Blocks are moved one level at a time.

Block addressing varies by technology at each level.

Block = Minimum unit of information transfer between levels of the hierarchy.
Apply average access time to entire hierarchy.

\[ \text{Average Memory Access Time (AMAT)} \]

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

\[ \text{Access time} = (\text{hit time}) + (\text{miss penalty}) \times (\text{miss rate}) \]

Cache Access Time
**Note:** Numbers are **local** hit rates – the ratio of access that go to that cache that is hit (remember, higher levels fill lower levels to lower levels).

<table>
<thead>
<tr>
<th>Access Time</th>
<th>Hit Rate</th>
<th>Hit Time</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>50,000 cycles</td>
<td>100%</td>
<td>50 cycles</td>
<td>Disk</td>
</tr>
<tr>
<td>10 + 0.05 * 65 = 69</td>
<td>99%</td>
<td>10 cycles</td>
<td>Main Memory</td>
</tr>
<tr>
<td>1 + 0.05 * 65 = 4.25</td>
<td>95%</td>
<td>1 cycle</td>
<td>L2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>L1</td>
</tr>
</tbody>
</table>

**Cache Access Time Example**