Memory Hierarchy

7 min

What is a *memory hierarchy* and why is it important? Let’s answer the second part of that question by looking at the graph below:

Year after year,

Preview: Docs Loading link description

processor

performance increases at a much higher rate than that of memory. This is the processor-memory performance gap and results in a computer that can process data much faster than it can retrieve data from memory.

In the gardening example from the previous exercise, you needed to get fertilizer from the store. The garden is equivalent to the processor and retrieving the fertilizer from the store is the same as retrieving data from the main memory.

The image above represents a simple memory hierarchy. At the top is the processor with the best performance. But it can only hold a small amount of data. At the bottom is memory with decreased performance but increased size for data. This memory is the DDR memory used widely in computers today and throughout this lesson, it will be called the main memory.

The middle section of the memory hierarchy is the

Preview: Docs Loading link description

cache

and is equivalent to your storage shed in the gardening example. The shed is where the extra fertilizer is stored for later use. Cache memory is similar in that it stores data for faster access times to help bridge the processor-memory performance gap.

Instructions

1. Checkpoint 1 Passed

1.

Throughout this lesson, you will work with a simulated memory hierarchy. A description of each file is in the Hint.

In **app.py** an instance of the ISA() class is assigned to cache_arch. This is the simulated architecture that will run instructions to access the memory hierarchy.

The .read_instructions() method is called and passed **ex2_instructions**. This will run the code to retrieve data from memory.

Instead of using binary or hex data in the simulation, the data stored in memory will be alphanumeric characters, such as "a" or "Z".

Run the code. The output shows there is no memory found in the architecture.

Hint

The code through this lesson is a simulated memory hierarchy and consists of 5 files:

- **isa.py:** implements a simple architecture including registers and a reference to the next level of memory in the hierarchy. The module also defines few memory access commands and allows for the reading of instructions from a file. -**code.txt** is a group of instructions that the architecture will run for each exercise.
- **memory.py** defines the different memory types included in the architecture. To start there is a generic memory class Memory() which
the Register() and MainMemory() classes inherit from. The distinctive features of each memory type are the data size and the access time.
- **app.py** is the file you will use to configure the architecture and output important information regarding the memory access process. -**cache.py** is where you will implement the cache. It is currently empty.

2. Checkpoint 2 Passed

2.

Next, we'll add main memory to the architecture.

Directly following cache_arch initialization, call the .set_memory() method of cache_arch with the argument MainMemory().

When you run **app.py** you should see the console output of each instruction that accesses the main memory. In the end, the simulation outputs a text string of all the data accessed from the main memory and the total execution time.

Hint

Call cache_arch.set_memory() with MainMemory(). Use the following syntax:

ISA_instance.method(MemorySubClass())

app.py

```
from isa import ISA

from memory import Memory, MainMemory


if __name__ == "__main__":

  cache_arch = ISA()

  # Write your code below

  cache_arch.set_memory(MainMemory())


  # Architecture runs the instructions

  cache_arch.read_instructions("ex2_instructions")
```

```
# This outputs the memory data and code execution time
exec_time = cache_arch.get_exec_time()
if exec_time > 0:
  print(f"OUTPUT STRING: {cache_arch.output}")
  print(f"EXECUTION TIME: {exec_time:.2f} nanoseconds")
```

- **Output:**

```
➢ ISA memory: Main Memory
➢ lb r0 r1 - Main Memory read: M
➢ lb r0 r1 - Main Memory read: i
➢ lb r0 r1 - Main Memory read: s
➢ lb r0 r1 - Main Memory read: s
➢ lb r0 r1 - Main Memory read: i
➢ lb r0 r1 - Main Memory read: s
➢ lb r0 r1 - Main Memory read: s
➢ lb r0 r1 - Main Memory read: i
➢ lb r0 r1 - Main Memory read: p
➢ lb r0 r1 - Main Memory read: p
➢ lb r0 r1 - Main Memory read: i
➢ OUTPUT STRING: Mississippi
➢ EXECUTION TIME: 334.40 nanoseconds
```