In 6.1910 you were introduced to the the von Neumann architecture of a computer:

```
---------------       -------       ----------------
| Main memory | <---> | CPU | <---> | Input/Output |
---------------       -------       ----------------
```

Here,

- **Memory** holds programs and data.
- The **CPU** accesses and processes memory values.
- **Input/output** lets the machine communicate with the outside world.

In 6.1800, we're going to think about computers in this way:

```
-----------       ---------------       -------       ----------------
| Storage | <---> | Main memory | <---> | CPU | <---> | Input/Output |
-----------       ---------------       -------       ----------------
```

Notice that we've differentiated **memory** and **storage**.

Storage vs. Memory

Data can be stored on a computer in a variety of places: program registers, various caches, DRAM, on a disk, etc. These technologies differ in how much data they can store, how fast they can be accessed, and whether or not they provide volatile storage:

- Program registers: Very very fast to access. Very small; they might only store a single number.
- Caches: Very fast to access, still quite small. A machine can have multiple caches (e.g., L1, L2, and L3 caches), which differ based on their size and speed of access.
- Random access memory (most commonly, DRAM: Dynamic random access memory). Fast to access, and a medium amount of storage.

All of the above technologies provide *volatile* storage: when you turn the machine off, anything stored in the registers, caches, and RAM won't be there when you turn the machine back on. If you want to store a piece of data permanently, it needs to go on non-volatile storage:

- Solid-state drives: Large, much slower to access than memory.
- Hard-disk drives: Very large, much slower to access than memory, and also slower than solid-state drives.

A typical computer will either have a solid-state drive or a hard-disk drive, but not both. It's possible to use multiple drives with one computer (Lecture 15!), but even then, we tend to stick with a single technology. In many situations in 6.1800, we won't need to worry about whether the drive is solid-state or a hard-disk, so we'll just talk about "the drive" (or "the disk", or "the disk drive").

If data is stored on a disk drive, it will persist after the machine is turned off (i.e., when you turn the machine back on, that data will still be there); that is what it means to be non-volatile.

In 6.1800, when we use the term **storage**, we're referring to a disk drive. This means that storage is large, slow to access, and non-volatile. When we use the term **memory**, we are typically talking about DRAM (we might occasionally include caches + registers into that abstraction; we usually don't need to in this class). Memory is small, fast, and volatile.

This distinction matters **very much** in 6.1800. We are often faced with decisions about where to keep a piece of data. Storing very large pieces of data, or data that needs to be accessed for a long time (especially data that needs to persist through a failure) in memory doesn't make sense.

Moving data

Your computer does some a lot of work when it comes to deciding where a piece of data should be at any given time. For example, you might have a program — call it program.py — saved on disk. When you want to run that program, its instructions (in assembly) and data will be loaded into memory, and as the CPU starts running the program, certain pieces of data will be pulled into various caches, loaded into registers, etc.

The CPU

Once a program has been loaded into memory, the CPU begins executing its instructions. An *instruction pointer* (or *program counter*) is a special register that keeps track of the next instruction to execute. You're familiar with the typical five-stage pipelined processor from 6.1910:

- Instruction Fetch (IF): Fetch the next instruction
- Instruction Decode (DEC): Decode that instruction
- Execute (EX): Execute the instruction. Perform the indicated operation in the Arithmetic Logic Unit (ALU)
- Memory access (MEM): Access any memory needed
- Write back (WB): Write the result back to a register

We call this a pipelined processor because we don't have to wait for a single instruction to complete all five stages before beginning to work on the next instruction. The CPU can fetch instruction 1, and decode it as it's fetching instruction 2, etc. There are a lot of details to take care of here: for example, an instruction may depend on something produced by a previous instruction. In that case the pipeline might stall, it might bypass the instruction, or it might speculate about what the result will be.

This is a good time to differentiate between a program and a process. A **program** is just a collection of instructions. A **process** is an instance of a program that is being executed. That means that a process includes the program code as well as the program *state* (values of registers, memory, etc.).

Why this matters in 6.1800

In the very first part of 6.1800, we're going to build on this knowledge from 6.1910. We'll learn how an operating system *virtualizes* your computer's physical hardware. This will require us to talk about how memory is virtualized, how processes are virtualized, and how communication between processes works. We'll start with virtual memory, which you might remember from 6.1910; we'll focus more on why virtual memory is a useful abstraction (and why virtualization in general is useful) than you did in 6.1910.