Introducing the Memory Hierarchy

HW1 due Friday;
HW2 released after class;
Check In 3 on Friday

The “Atlas 2” from Titan was one of the first processors with a CPU Cache

One of the first “time-sharing” processors ➡️ an early predecessor to cloud compute!

The “cache” was itself main memory — faster than the magnetic tape decks and rotating drum-stores!

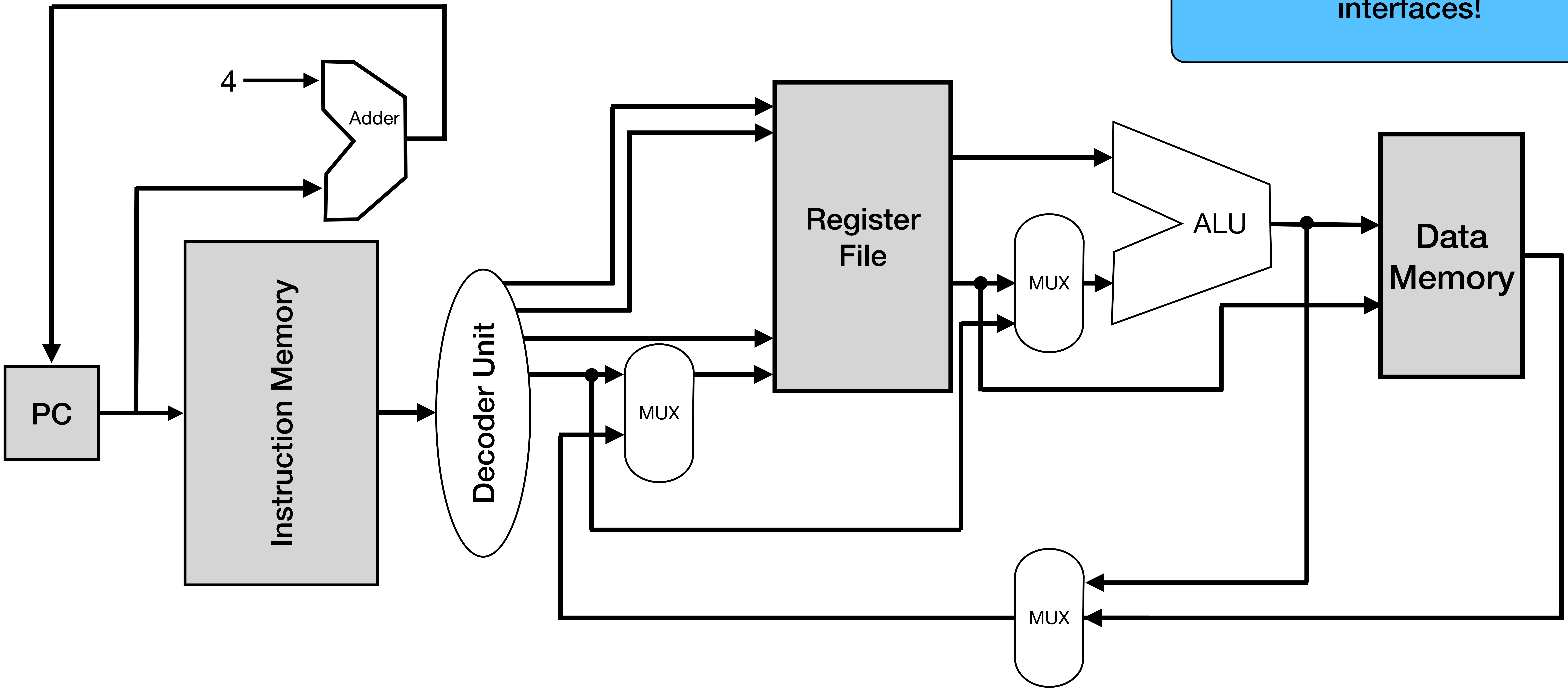
Image credit: https://en.wikipedia.org/wiki/Titan_(1963_computer)

Outline

- Memory hierarchy overview and principles
- Fleshing out the processor-memory interface
- Data storage methodologies

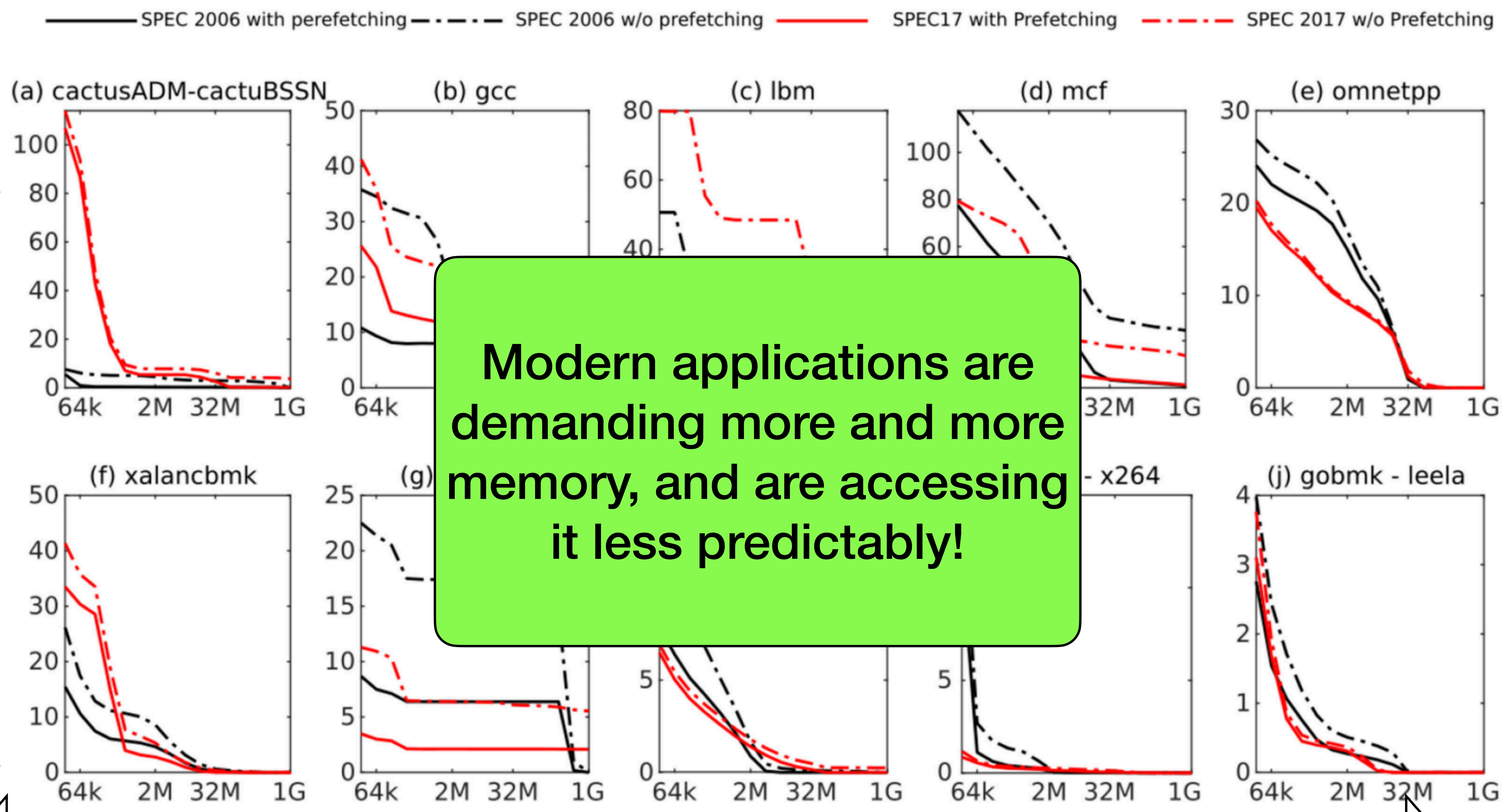
The Memory Story so Far…

Instruction and data memories as storage components (just like register files) with well-defined interfaces!



In Reality…

Credit: Hassan, et al. “A Reusable Characterization of the Memory System Behavior of SPEC2017 and SPEC2006”. ACM TACO 2021



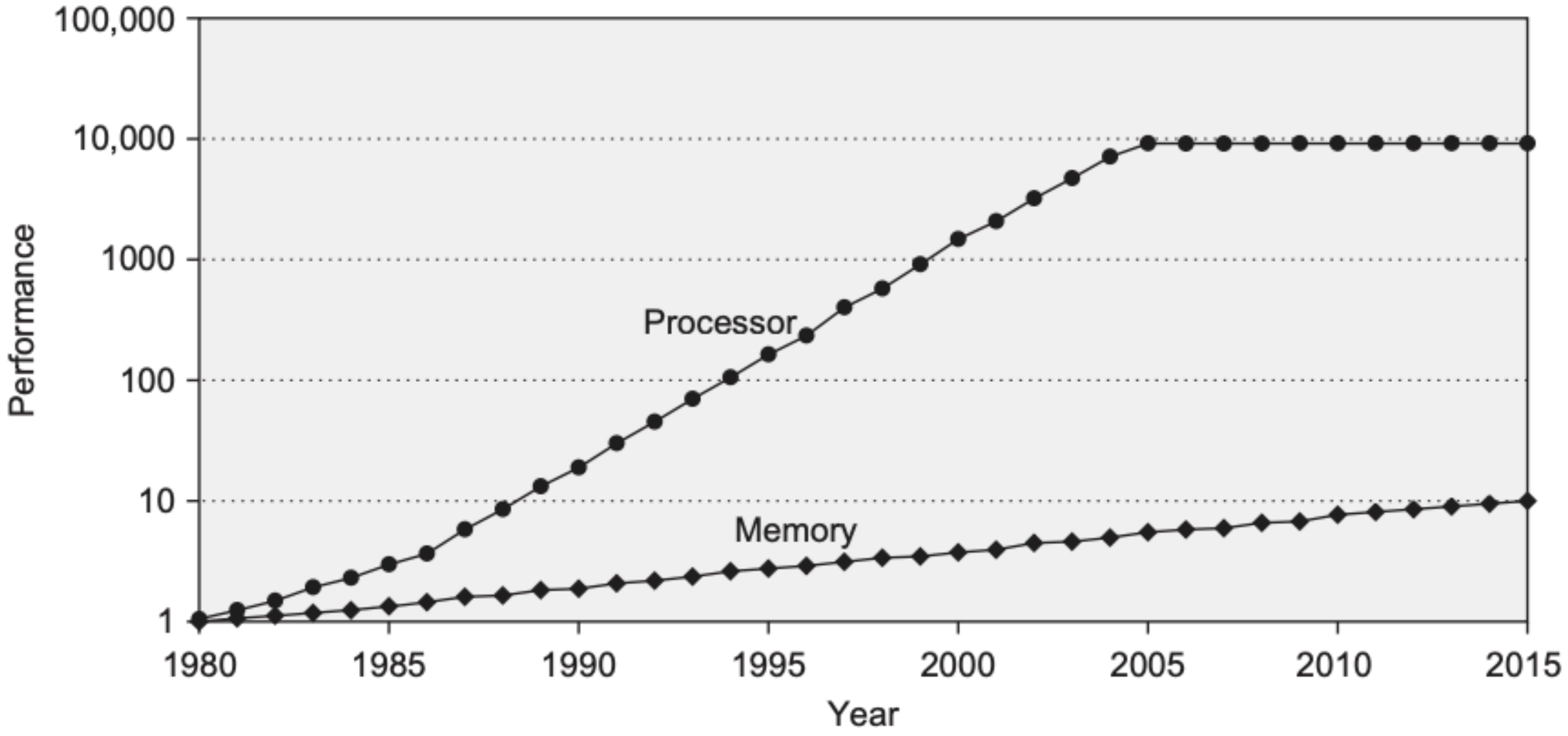
Modern applications are demanding more and more memory, and are accessing it less predictably!

Misses per 1000 instructions

Cache Size

In Reality…

Credit: *Computer Architecture: A Quantitative Approach* (p 80)



Advances in processor designs are directly related to the increased demand for memory!

Principles of Storage Device Design

- To accommodate increasing memory demands, hardware devices need to be large enough to meet t

- The performance impr oes not scale at the same rate as processo become a bottleneck over time!

- Smaller storage devices (e.g., registers) tend to be faster (and more expensive due to having more transistors) than larger storage devices… there is a size versus speed trade-off!

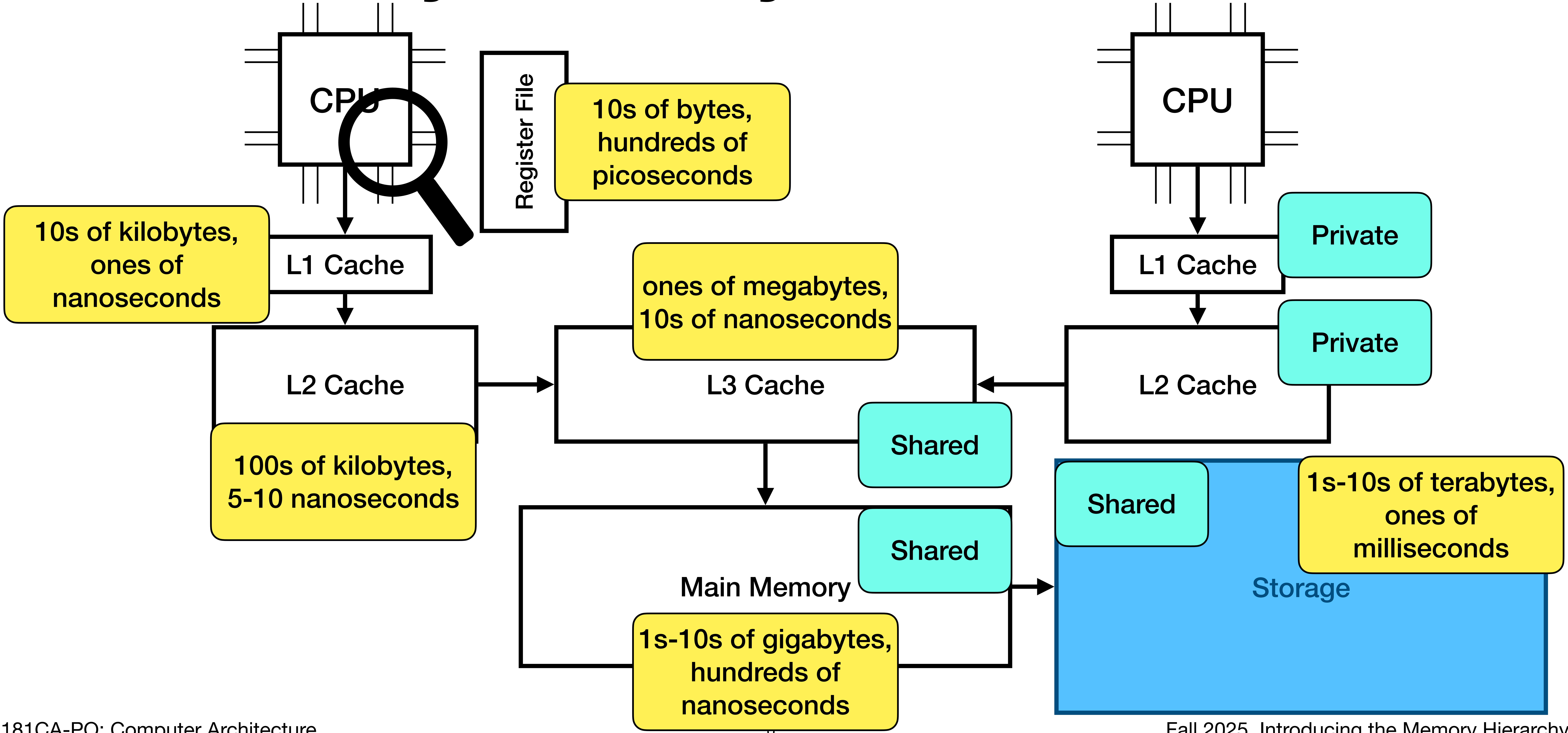
- Do we want a large memory or a fast memory?

We want a large *AND* fast memory!

The Memory Hierarchy

volatile

non-volatile



CPU

Register File

10s of bytes, hundreds of picoseconds

10s of kilobytes, ones of nanoseconds

L1 Cache

L2 Cache

100s of kilobytes, 5-10 nanoseconds

ones of megabytes, 10s of nanoseconds

L3 Cache

Shared

Main Memory

Shared

1s-10s of gigabytes, hundreds of nanoseconds

CPU

L1 Cache

Private

L2 Cache

Private

Storage

Shared

1s-10s of terabytes, ones of milliseconds

Principles of the Memory Hierarchy

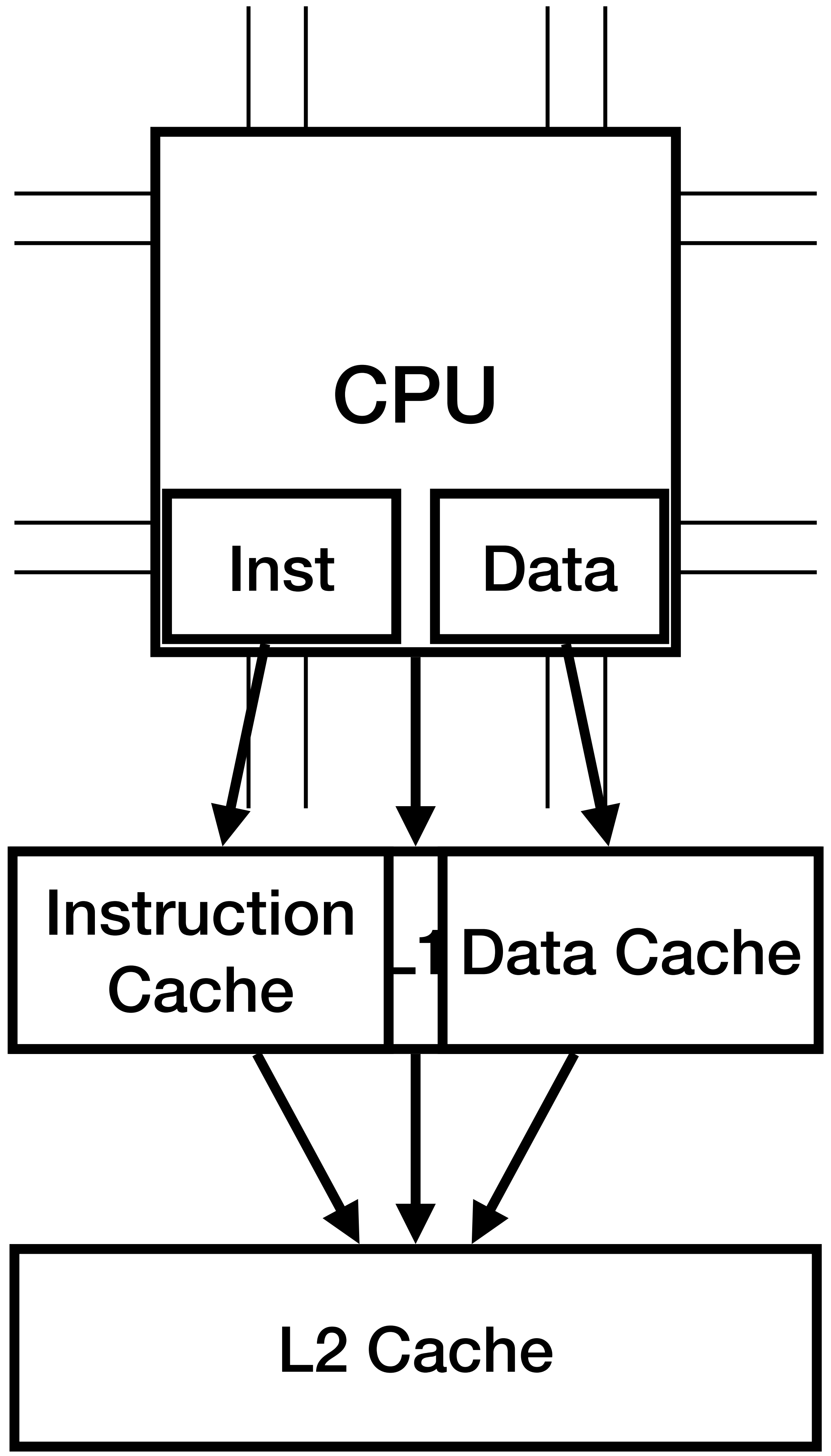
- Applications tend to exhibit *locality*, so it is likely that spatially similar (i.e., the next address) or temporally similar (i.e., recently used) data are going to be used again
- To capture varying degrees of locality, the memory system tends to be organized hierarchically so that faster devices (registers and caches) tend to be accessed first by the processor (i.e., smaller, faster devices towards the processor)
- As components move further away from the processor, they are more likely to be shared by multiple processors/caches for the same functionality
- “Storage” is said to be non-volatile as it retains its state throughout a crash ➡️ this is possible due to a different composition of integrated circuitry

Chat with your neighbor(s)!

Consider the Python program below. What are some of the steps happening in the microarchitecture (specifically the memory system!) to implement this behavior?

```
f = open(“filename.txt”, “w”)
f.write(“Hello world!”)
f.close()
```

Revisiting Instruction versus Data Memory

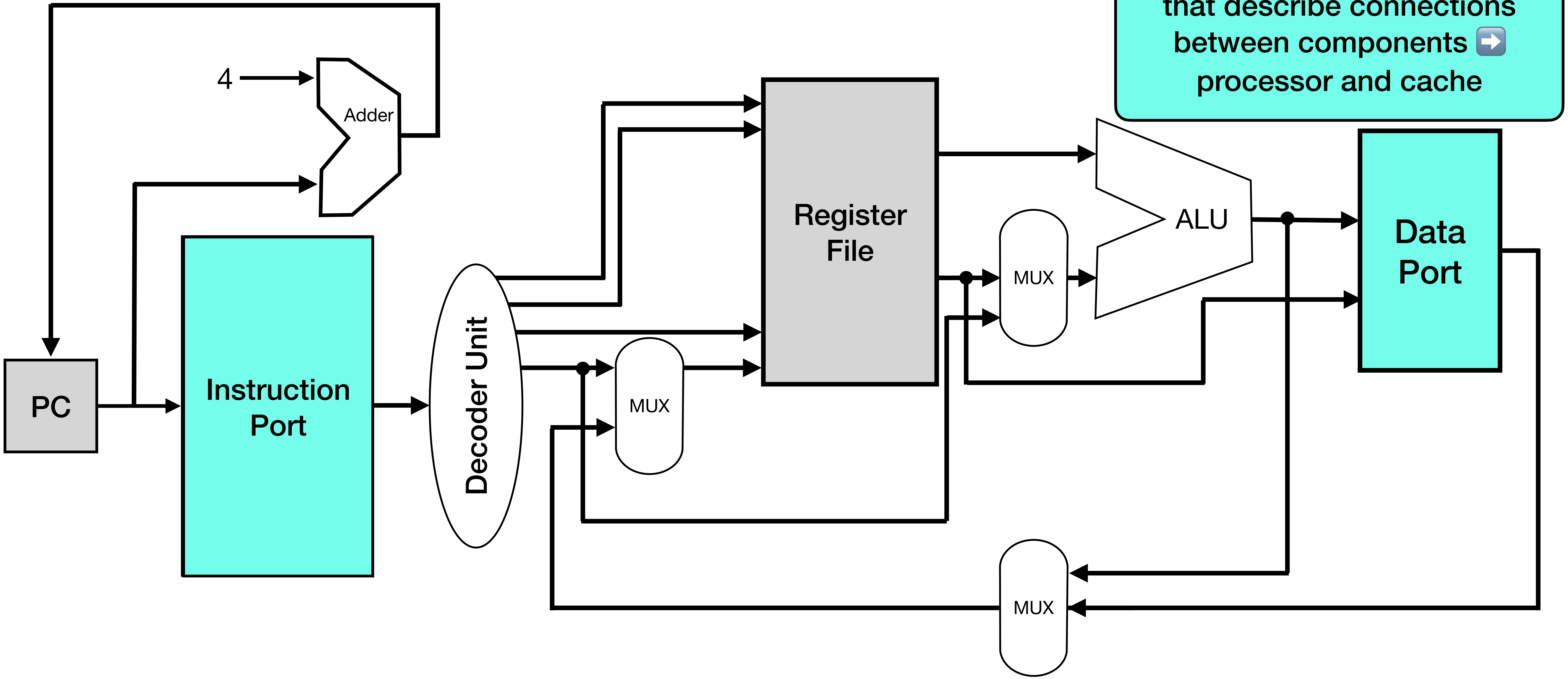


The processor views instruction/data memory as accessing the instruction/ data cache!

The L2 cache acts as a shared cache for both instructions and data! This way, there is only one memory device for all processed values

This means that the number of cycles to access data in memory is non-deterministic…

Revisiting Instruction versus Data Memory



"Ports" are physical interfaces that describe connections between components ➡️ processor and cache

Handling Non-Deterministic Access Latency

- Accesses to the instruction/data caches are fast if the data resides in the cache (i.e., a cache *hit*)
- On a cache miss, the latency may be longer… and our processor needs to be able to handle this case!
- In the implementation of the port, the processor must implement a *load-store queue* to track the presently outstanding requests
- If an operation takes longer than the expected cycle time, our processor pipeline will *stall* until the data comes back to the processor ➡️ this is the same mechanism as handling a hazard!

Chat with your neighbor(s)!

Suppose a processor with a five-stage pipeline is implemented with the expectation that the data cache hit rate will be 80%. How many entries should the load-store queue in the data port support?

Using Memory in Applications

- Typically, running applications use “main memory” to store global and local variables during its execution
- Using this terminology, “main memory” refers to all of the memory system *before* long term storage (i.e., flash, solid-state drives, hard disk drives)
- If an application would like data to be stored in a long-term storage device (i.e., a file in the file system), it needs to use a *system call* (i.e., “`open`” in Python) where the operating system will interface with that device directly
- In practice, this means using a special subset of the instruction set in which the processor issues a particular set of memory instructions that *flush* data from the volatile state (e.g. caches and memory) to the long term storage device

Takeaways

- Modern applications use a lot of memory, so we need a memory system!
- The memory system gives the illusion of a large, fast memory by leveraging typical application behaviors to organize storage components hierarchically
- We can leverage our memory system characteristics to implement “instruction memory” and “data memory” in our processor
- If we want to store data through a power loss (e.g., in the file system), we need to explicitly flush data contents to a long-term storage device