## Memory Hierarchy

**Definition:**

- several memory components, each of which has different sizes, speeds & cost per MB
  - close to the CPU: small, fast access, high cost
  - close to memory: large, slow access, lower cost

**Typical memory hierarchies today:**

- registers: 32, < half cycle access
- level 1 cache (on-chip, SRAM): 8KB-64KB, 1-2 cycle access
- level 2 cache (on-chip or on-board, SRAM): 128KB-4MB, 6-10 cycles access
- memory (DRAM): 32MB-1.5GB, 40-100 cycles
- disks: 1-12 GB, 10 ms
- archival storage: practically unlimited, not matter



## Memory Hierarchy

**Problem the memory hierarchy is trying to address:**

- the **processor-memory bottleneck**: the discrepancy between CPU & memory speeds
  - CPU speed increases 55% per year
  - DRAM speed increases 9% per year
- rate of increase is also widening

What design principle comes into play here?

**How memory hierarchies address this problem:**

- keep information that the CPU uses often or will use next in storage that is close to it
- storage is smaller than main memory, and therefore:
  - it is faster than memory
  - it doesn't hold much & you need to be smart about what you store in it

# Locality

**Principle of locality of reference**:

- programs repeatedly access a small portion of their instructions & data at any one time
- a reason you get benefit from small, demand-driven storage

- **temporal locality:** code/data that was used in the recent past will be referenced again soon
  examples:
  - code:
  - data:
- **spatial locality:** code/data that is near code/data that is currently being referenced will be referenced soon
  examples:
  - code:
  - data:

Caches are **demand-driven**:

- load data/instructions into them when they are needed
- once there,
  will be used again (temporal locality)
  locations brought in at the same time will be used (spatial locality)



# Memory Hierarchies Work!

Memory hierarchies improve performance

- locality of reference
- technology: speed vs. size

$\Rightarrow$ fast access for most references

Both factors are important:

- If you get program and data size down, you can fit it into a small memory & access it quickly.
- You can get away with a small size because of locality of reference.