UC San Diego

DSC 102
Systems for Scalable Analytics

Rod Albuyeh

Topic 1: Basics of Machine Resources
Part 1: Computer Organization

Ch. 1, 2.1-2.3, 2.12, 4.1, and 5.1-5.5 of CompOrg Book

Outline

- Basics of Computer Organization
  - Digital Representation of Data
  - Processors and Memory Hierarchy
- Basics of Operating Systems
  - Process Management: Virtualization; Concurrency
  - Filesystem and Data Files
  - Main Memory Management
- Persistent Data Storage

Basics of Processors

- **Processor:** Hardware to orchestrate and *execute instructions* to *manipulate data* as specified by a program
  - Examples: CPU, GPU, FPGA, TPU, embedded, etc.
- **Instruction Set Architecture (ISA):** The vocabulary of commands of a processor

Program in PL

Compile/Interpret ↓

Program in Assembly Language

Assemble ↓

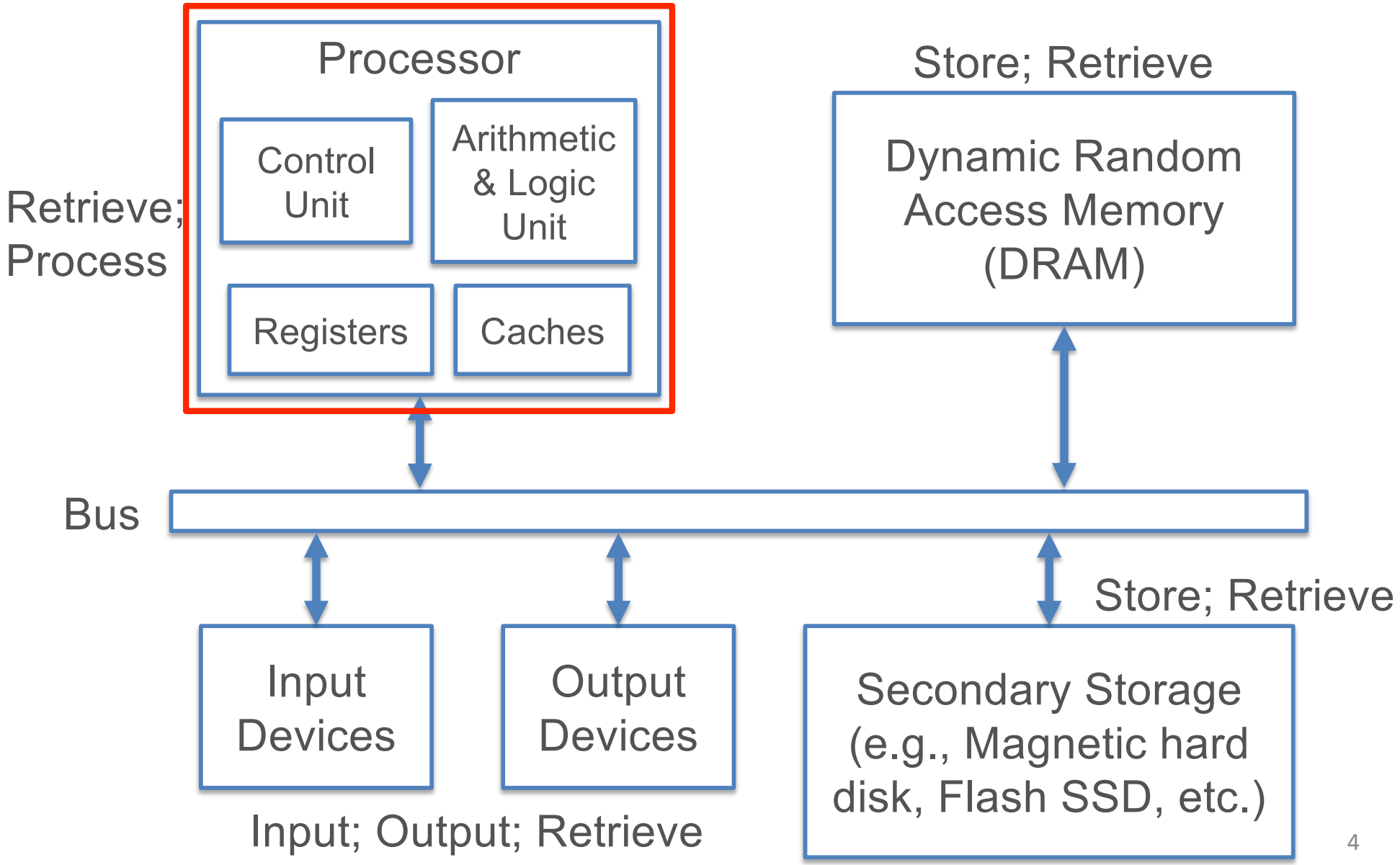
Machine code tied to ISA

↓

Run on processor

```
80483b4:    55                      push   %ebp
80483b5:    89 e5                   mov    %esp,%ebp
80483b7:    83 e4 f0                and    $0xfffffff0,%esp
80483ba:    83 ec 20                sub    $0x20,%esp
80483bd:    c7 44 24 1c 00 00 00    movl   $0x0,0x1c(%esp)
80483c4:    00
80483c5:    eb 11                   jmp    80483d8 <main+0x24>
80483c7:    c7 04 24 b0 84 04 08    movl   $0x80484b0,(%esp)
80483ce:    e8 1d ff ff ff          call   80482f0 <puts@plt>
80483d3:    83 44 24 1c 01          addl   $0x1,0x1c(%esp)
80483d8:    83 7c 24 1c 09          cmpl   $0x9,0x1c(%esp)
80483dd:    7e e8                   jle    80483c7 <main+0x13>
80483df:    b8 00 00 00 00          mov    $0x0,%eax
80483e4:    c9                      leave
80483e5:    c3                      ret
80483e6:    90                      nop
80483e7:    90                      nop
80483e8:    90                      nop
80483e9:    90                      nop
80483ea:    90                      nop
```

Abstract Computer Parts and Data



Basics of Processors

Q: *How does a processor execute machine code?*

- Most common approach: **load-store architecture**
- **Registers:** Tiny local memory ("scratch space") on proc. into which instructions and data are copied
- ISA specifies bit length/format of machine code commands
- ISA has several commands to manipulate register contents

Basics of Processors

Q: *How does a processor execute machine code?*

- Types of ISA commands to manipulate register contents:
  - ***Memory access*: load** (copy bytes from DRAM address to register); **store** (reverse)
  - ***Arithmetic & logic*** on data items in registers: add/multiply/etc.; bitwise ops; compare, etc.
  - ***Control flow*** (branch, call, etc.)
- **Caches:** Small local memory to buffer instructions/data

This might help give better intuition: https://www.youtube.com/watch?v=cNN_tTXABUA

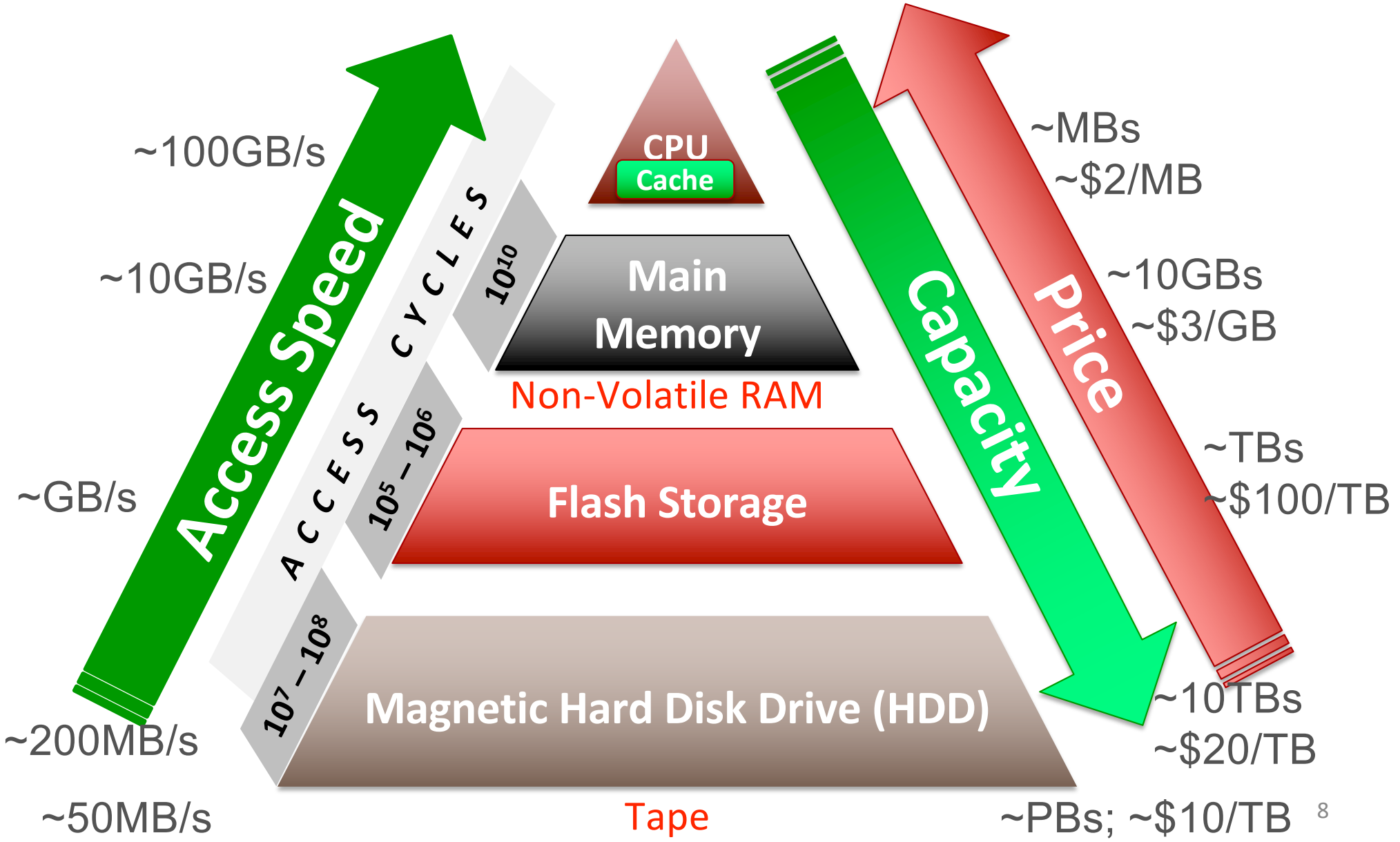
Processor Performance

***Q:** How fast can a processor process a program?*

- Modern CPUs can run millions of instructions per second!
  - ISA influences #**clock cycles** each instruction needs
  - CPU's **clock rate** lets us convert that to runtime (ns)
- Alas, most programs do not keep CPU always busy
  - Memory access commands **stall** the processor; Arithmetic & Logic Unit and Control Unit are *idle* during memory-register transfer
  - Worse, data may not be in DRAM—wait for disk I/O!
  - So, actual *execution runtime* of program may be orders of magnitude higher than what clock rate calculation suggests

Key Principle: Optimizing access to main memory and use of processor cache is critical for processor performance!

Memory/Storage Hierarchy



Memory/Storage Hierarchy

- Typical desktop computer today ($700):
  - 1 TB magnetic hard disk (SATA HDD); 32 GB DRAM
  - 3.4 GHz CPU; 4 cores; 8MB cache
- High-end enterprise rack server for RDBMSs ($8,000):
  - 12 TB Persistent memory; 6 TB DRAM
  - 3.8 GHz CPU; 28-core per proc.; 38MB cache
- Renting on Amazon Web Services (AWS):
  - EC2 m5.large: 2-core, 8GiB: $0.115 / hour
  - EC2 m5.24xlarge: 96-core, 384 GiB, $5.53 per hour
  - EBS general SSD: $0.12 per GB-month
  - S3 store / read: $0.023 / 0.05-0.09 per GB-month

Key Principle: Locality of Reference

Carefully handling/optimizing access to main memory and use of processor cache is critical for processor performance!

Due to OOM access latency differences across memory hierarchy, optimizing access to lower levels and careful use of higher levels is critical for overall system performance!

- **Locality of Reference:** Many programs tend to access memory locations in a somewhat *predictable* manner
  - **Spatial:** Nearby locations will be accessed soon
  - **Temporal:** Same locations accessed again soon
- Locality can be exploited to reduce runtimes using **caching** and/or **prefetching** across all levels in the hierarchy

Concepts of Memory Management

- **Caching:** Buffering a copy of bytes (instructions and/or data) from a lower level at a higher level to exploit locality
- **Prefetching:** Preemptively retrieving bytes (typically data) from addresses not explicitly asked yet by program
- **Spill/Miss/Fault:** Data needed for program is not yet available at a higher level; need to get it from lower level
  - **Register Spill** (register to cache); **Cache Miss** (cache to main memory); **"Page" Fault** (main memory to disk)
- **Hit:** Data needed is already available at higher level
- **Cache Replacement Policy:** When new data needs to be loaded to higher level, which old data to evict to make room? Many policies exist with different properties

Memory Hierarchy in Action

Q: *What does this program do when run with 'python'?*
(Assume tmp.csv is in current working directory)

tmp.py

```
import pandas as pd
df = pd.read_csv('tmp.csv',header=None)
s = df.sum().sum()
print(s)
```

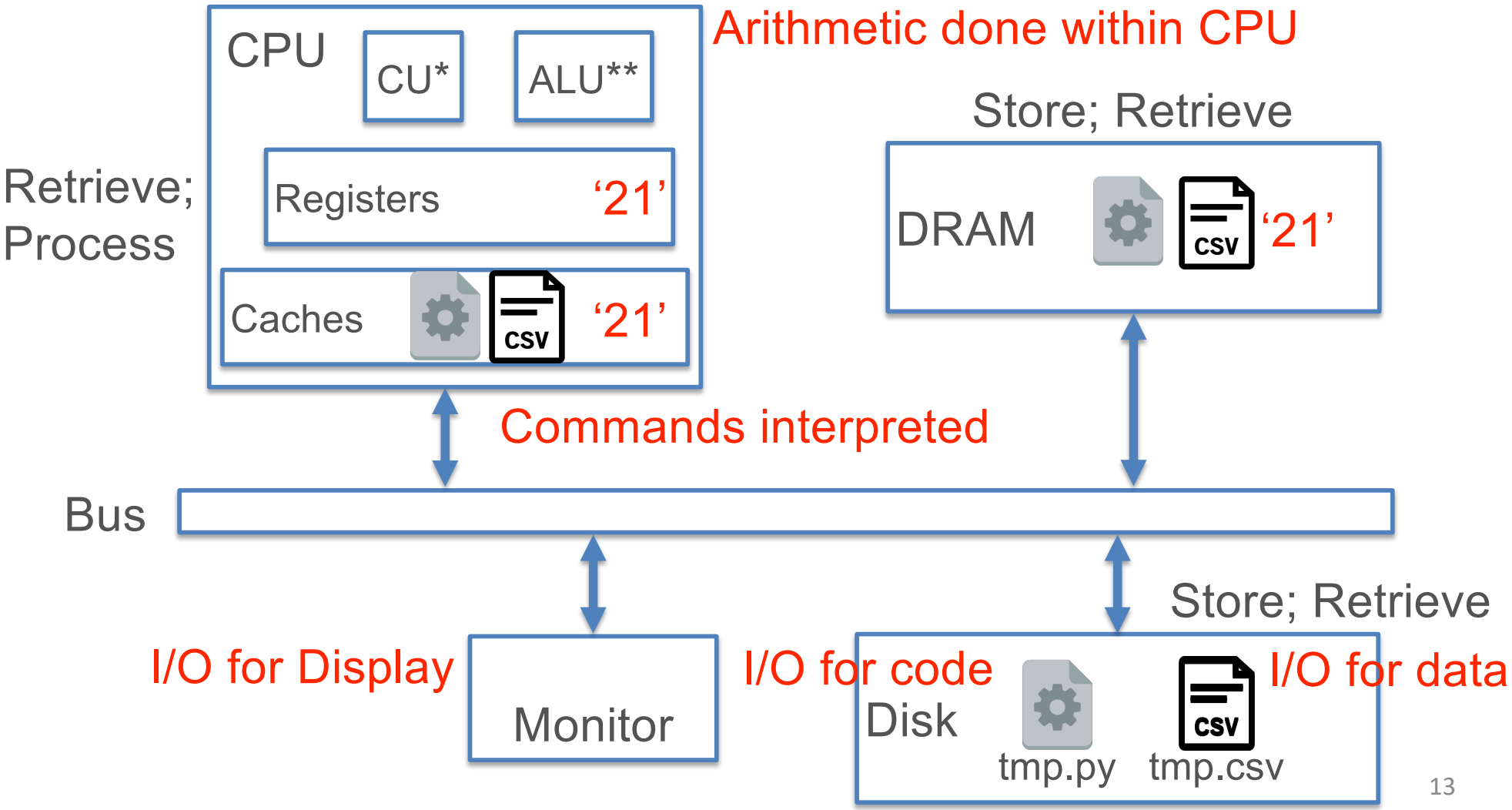
tmp.csv

```
1,2,3
4,5,6
```

Memory Hierarchy in Action

Rough sequence of events when program is executed

[*] CU: Control Unit, [**] ALU: Arithmetic Logic Unit



Locality of Reference for Data

- **Data Layout:**
  - The *order* in which data items of a complex data structure or an abstract data type (ADT) are laid out in memory/disk
- **Data Access Pattern** (of a program on a data object):
  - The *order* in which a program has to access items of a complex data structure in memory
- **Hardware Efficiency** (of a program):
  - How close *actual execution runtime* is to best possible runtime given the CPU clock rate and ISA
  - Improved with careful data layout of all data objects used by a program based on its data access patterns
  - **Key Principle:** Raise cache hits; reduce memory stalls!

Locality of Reference in Data Science

- Common example: matrix multiplication (>1m cells each)
- Suppose data layout in DRAM is in ***row-major*** order

$$C_{n\times m} = A_{n\times p}\ B_{p\times m}$$

DRAM A[1;] A[2;] A[3;] … B[1;] B[2;] …

Caches

```
for i = 1 to n
  for j = 1 to m
    for k = 1 to p
      C[i][j] += A[i][k] * B[k][j]
```

- Not too hardware-efficient
- Prefetching+caching means full row based on innermost loop is brought to CPU cache
- A[i][.] Hits but B[k][j] Misses
- So each * op is a stall! :(

Locality of Reference in Data Science

- Common example: matrix multiplication (>1m cells each)
- Suppose data layout in DRAM is in ***row-major*** order

$$C_{n\times m} = A_{n\times p}\ B_{p\times m}$$

DRAM A[1;] A[2;] A[3;] … B[1;] B[2;] …

Caches

```
for i = 1 to n
  for k = 1 to p
    for j = 1 to m
      C[i][j] += A[i][k] * B[k][j]
```

- *Logically equivalent* computation but different order of ops!
- C[i][.] and B[k][.] Hits
- A[i][k] also Hit (unaffected by j)
- Orders of magnitude fewer stalls!
- Lot more hardware-efficient

Locality of Reference in Data Science

- Common example: matrix multiplication (>1m cells each)
- Suppose data layout in DRAM is in ***row-major*** order

$$C_{n\times m} = A_{n\times p}\ B_{p\times m}$$

```
for i = 1 to n
  for j = 1 to m
    for k = 1 to p
      C[i][j] += A[i][k] * B[k][j]
```

Rewrite ⇩

```
for i = 1 to n
  for k = 1 to p
    for j = 1 to m
      C[i][j] += A[i][k] * B[k][j]
```

- Although the math is the same and gives the same results ("logically equivalent"), the physical properties of program execution are vastly different
- Commonly used in *compiler optimization* and later on, also in *query optimization*

Locality of Reference in Data Science

- Matrices/tensors are ubiquitous in statistics/ML/DL programs

Q: *Would you like to write ML code in a cache-aware manner? :)*

- Decades of optimized hardware-efficient libraries exist for matrix/tensor arithmetic (*linear algebra*) that *reduce memory stalls* and *increase parallelism* (more on parallelism later) for you
  - **Multi-core CPUs:** BLAS/LAPACK (C), Eigen (C++), la4j (Java), NumPy/SciPy (Python; can wrap BLAS)
  - **GPUs:** cuBLAS, cuSPARSE, cuDNN, cuDF, cuGraph

If interested, some benchmark empirical comparisons:

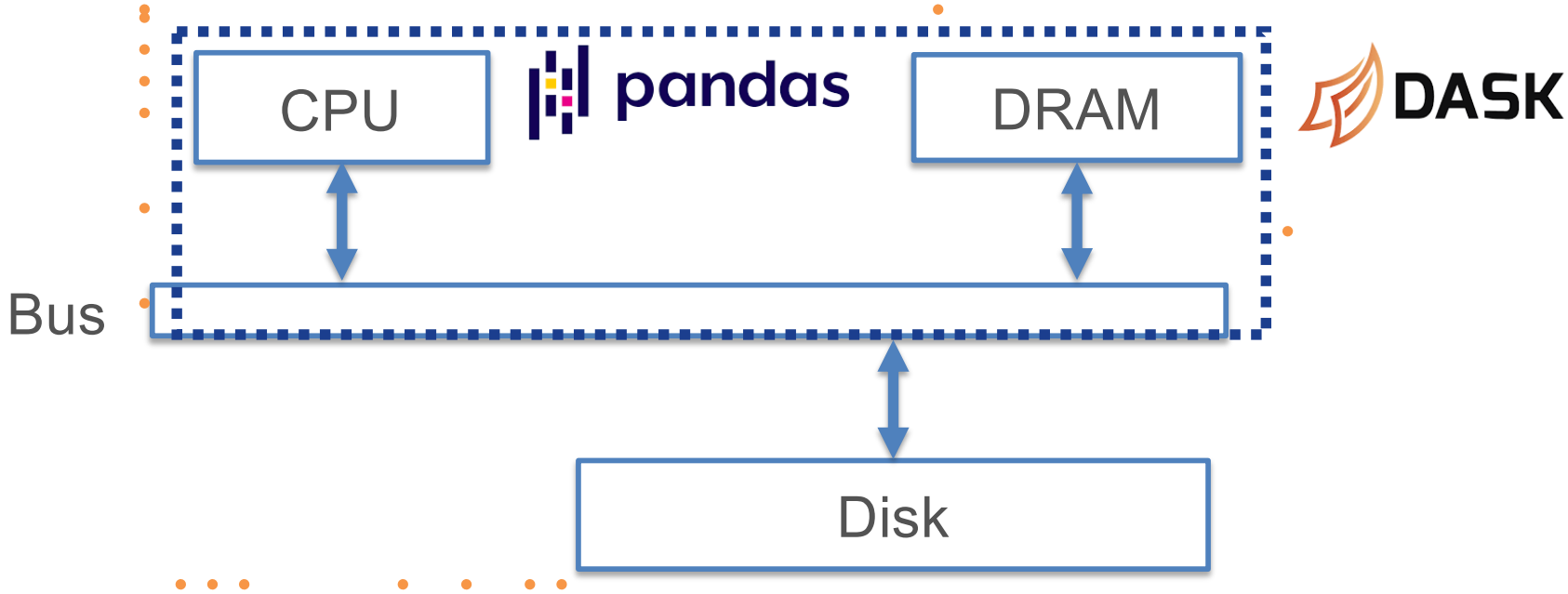
https://medium.com/datathings/benchmarking-blas-libraries-b57fb1c6dc7

https://github.com/andre-wojtowicz/blas-benchmarks

https://eigen.tuxfamily.org/index.php?title=Benchmark

Memory Hierarchy in PA0

- Pandas DataFrame needs data to fit entirely in DRAM
- **Dask DataFrame** automatically manages Disk vs DRAM for you
  - Full data sits on Disk, brought to DRAM upon compute()
  - Dask stages out computations using Pandas



- **Tradeoff:** Dask may throw memory configuration issues. :)

Review Questions

- What is an ISA?
- What are the three main kinds of commands in an ISA?
- Why do CPUs have both registers and caches?
- Why is it typically impossible for data processing programs to achieve 100% processor utilization?
- Which of these layers in the memory hierarchy is the costliest: CPU cache, DRAM, flash disks, or magnetic hard disks?
- Which of the above layers is the slowest for data access?
- What is spatial locality of reference? Briefly describe a simple program that exhibits that.
- Why does data layout matter for a program's hardware efficiency?
- Which is more important for optimizing a program's hardware efficiency: data layout or data access pattern?

Outline

- Basics of Computer Organization
  - Digital Representation of Data
  - Processors and Memory Hierarchy
- Basics of Operating Systems (OS)
  - Process Management: Virtualization; Concurrency
  - Filesystem and Data Files
  - Main Memory Management
- Persistent Data Storage