Parallelism [*optional lecture*]

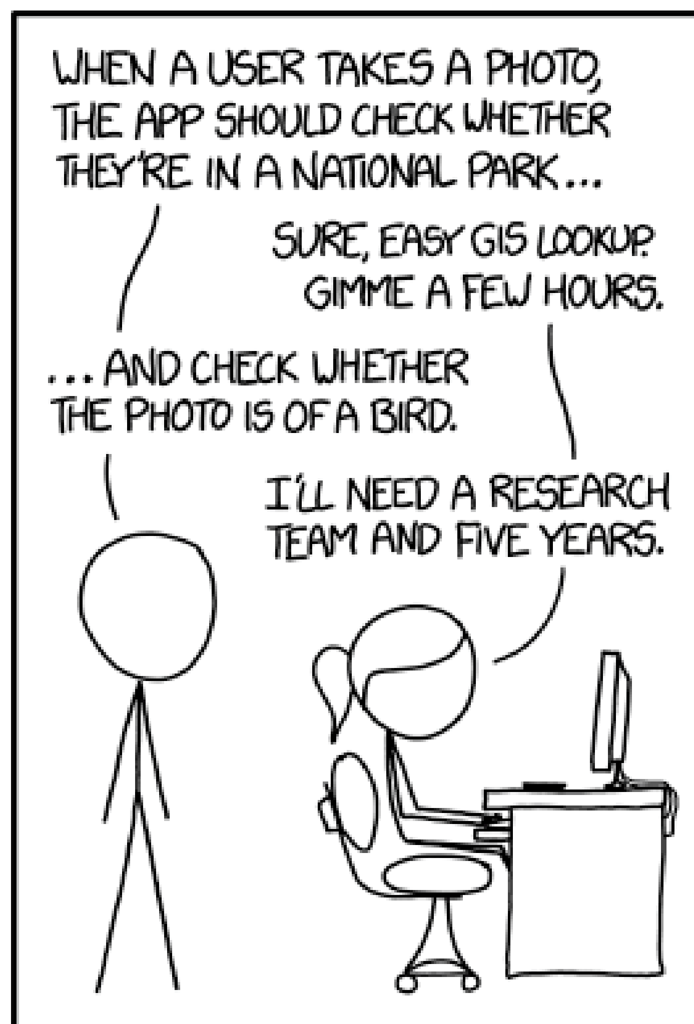
CSE 351 Autumn 2018

Instructor:

Justin Hsia

Teaching Assistants:

Akshat Aggarwal
An Wang
Andrew Hu
Brian Dai
Britt Henderson
James Shin
Kevin Bi
Kory Watson
Riley Germundson
Sophie Tian
Teagan Horkan



IN CS, IT CAN BE HARD TO EXPLAIN THE DIFFERENCE BETWEEN THE EASY AND THE VIRTUALLY IMPOSSIBLE.

https://xkcd.com/1425/

Administrivia

- Lab 5 due Friday (12/7)
  - Hard deadline on Sunday (12/9)
- Course evaluations now open
  - See Piazza post @485 for links (separate for Lec and Sec)
- **Final Exam:** Wed, 12/12, 12:30-2:20 pm in KNE 120
  - Review Session: Sun, 12/9, 5-7 pm in EEB 105
  - You get TWO double-sided handwritten 8.5×11” cheat sheets
  - Additional practice problems on website

Concurrency vs. Parallelism

- **Concurrency** in CS is "the ability of different parts or units of a program, algorithm, or problem to be executed out-of-order or in partial order, without affecting the final outcome." – Wikipedia
  - Concurrent computing is when the execution of multiple computations (or processes) *overlap*
  - Parallel computing is when the execution of multiple computations (or processes) occur *simultaneously*
- These terms are related, but independent

Concurrency in Hardware and Software

- Choice of hardware setup and software design are independent
  - *Concurrent* software can also run on *serial* hardware
  - *Sequential* software can also run on *parallel* hardware

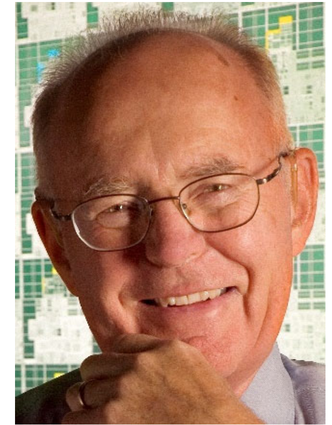
		Software	
		Sequential	Concurrent
Hardware	Serial	Matrix Multiply written in MatLab running on an Intel Pentium 4	Windows Vista Operating System running on an Intel Pentium 4
	Parallel	Matrix Multiply written in MATLAB running on an Intel Xeon e5345 (Clovertown)	Windows Vista Operating System running on an Intel Xeon e5345 (Clovertown)

need multiple processors

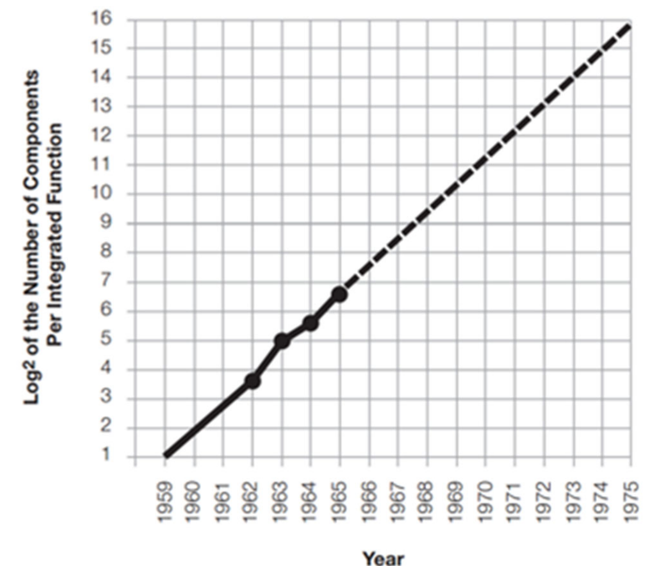
Types of Parallelism

- **Why Parallelism?**
- Thread-Level Parallelism
- Data-Level Parallelism
- Instruction-Level Parallelism

Moore's Law

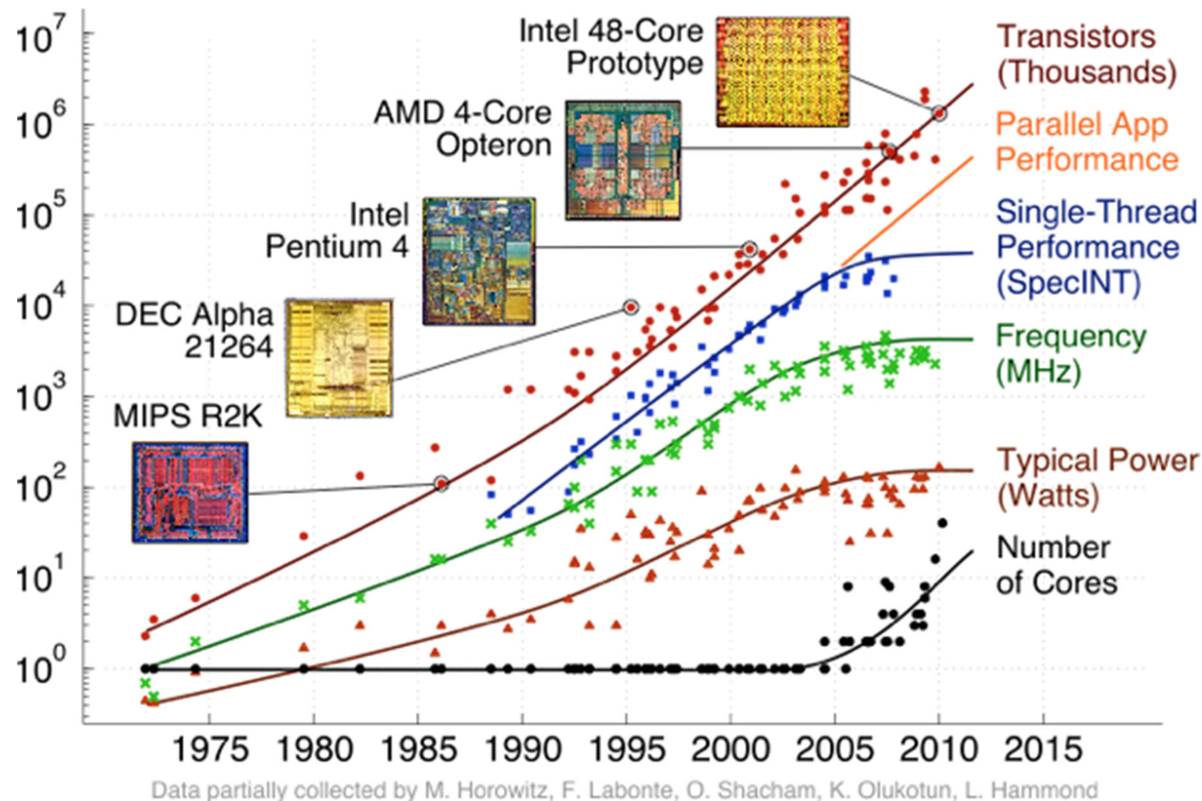


- In 1965, Gordon Moore observed that transistor density had ~doubled each year
  - ~16/chip in 1962 to ~128/chip in 1965
  - Predicted doubling would continue every year for a decade
- Became a self-fulfilling prophecy for industry
  - Up to ~65,000/chip in 1975
- Rate slowed after 1975 to doubling every ~2 years
  - Now slowing further



Transistor and Chip Scaling

- In order to fulfill Moore's Law, need to fit more transistors per chip
  1) Decrease transistor size to increase chip density & speed
  2) Increase chips size, but limited by defect rate



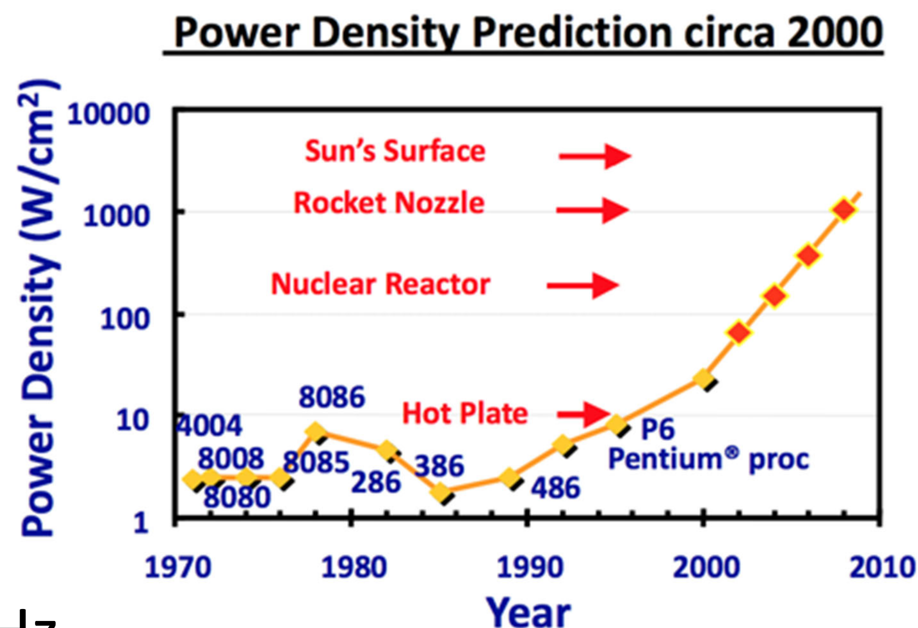
Computers today are *obscenely* powerful compared to their forebears!

Running into Physical Limits

- Power densities over time
  - Roughly constant from 1958 to late 1990s
  - Grow rapidly starting in late 1990s
- Since 2000, transistors too small to shrink further (just a few molecules thick)
  - Current leakage makes power scale with *frequency*
  - Power = C × V^2 × *f*

switching freq ~ clock speed

- <u>Heat Death</u>:
  - Processors stopped around 4 GHz



Source: S. Borkar (Intel)

The Rise of Multi-core

- Multi-core: Stick multiple processors on single chip
  - Modern laptops and desktops usually have ~4 CPUs
  - Each CPU can run a separate process or *thread*
  - Speed boost on *parallelizable* tasks

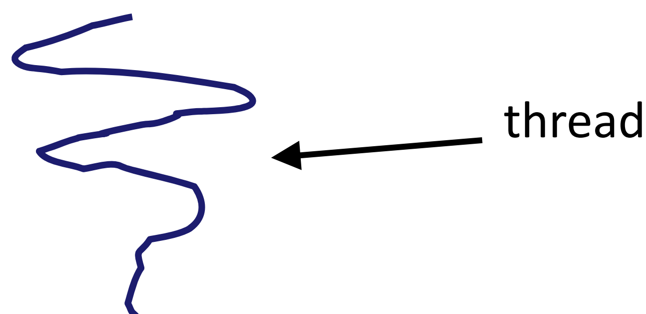
- The Challenges:
  - Requires new programming language (and hardware) tools for maximum effectiveness
  - Concurrent & parallel programs can be tricky to get right

Types of Parallelism

- Why Parallelism?
- **Thread-Level Parallelism**
  - **Multithreading**
  - **Synchronization**
  - **Cache Coherence**
- Data-Level Parallelism
- Instruction-Level Parallelism

Introducing Threads

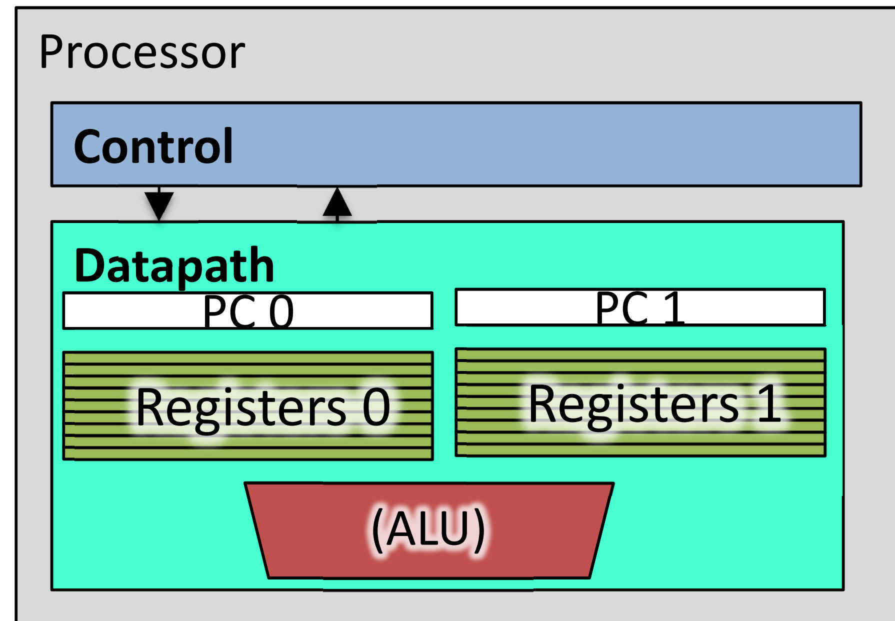
- Separate the concept of a process from that of a minimal "*thread of control*"
  - Usually called a thread (or a *lightweight process*), this is a sequential execution stream within a process



- In most modern OS's:
  - Process: address space, OS resources/process attributes (shared)
  - Thread: stack, stack pointer, program counter, registers (separate)
  - Threads are the unit of scheduling and processes are their containers

Hardware Support for Multithreading

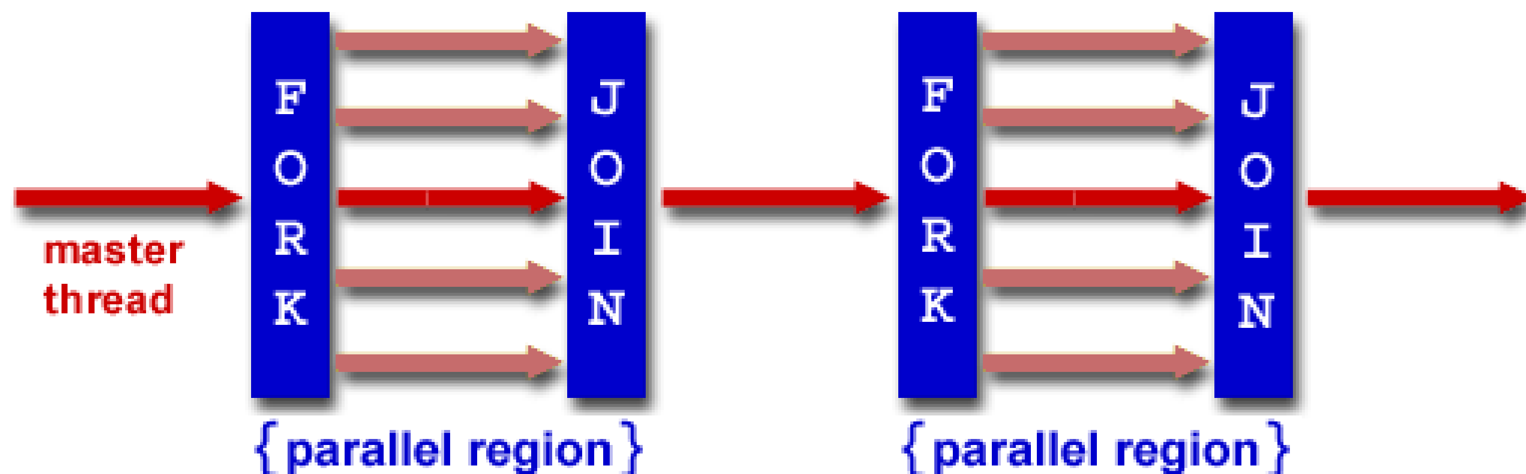
- Two copies of PC and Registers inside processor hardware:



  - Looks like two processors to software (hardware thread 0, hardware thread 1)
  - Control logic decides which thread to execute an instruction from next

Multithreading a Process

- Must explicitly tell machine how to do this
- There are many parallel/concurrent programming models
  - POSIX Threads (pthread) model (*taught in CSE333*)
    - `#include <pthread.h>`, compiled with `-pthread` flag in `gcc`
  - Fork-Join model (*taught in CSE332*)
    - `#include <omp.h>`, compiled with `-fopenmp` flag in `gcc`



Multithreading Limits

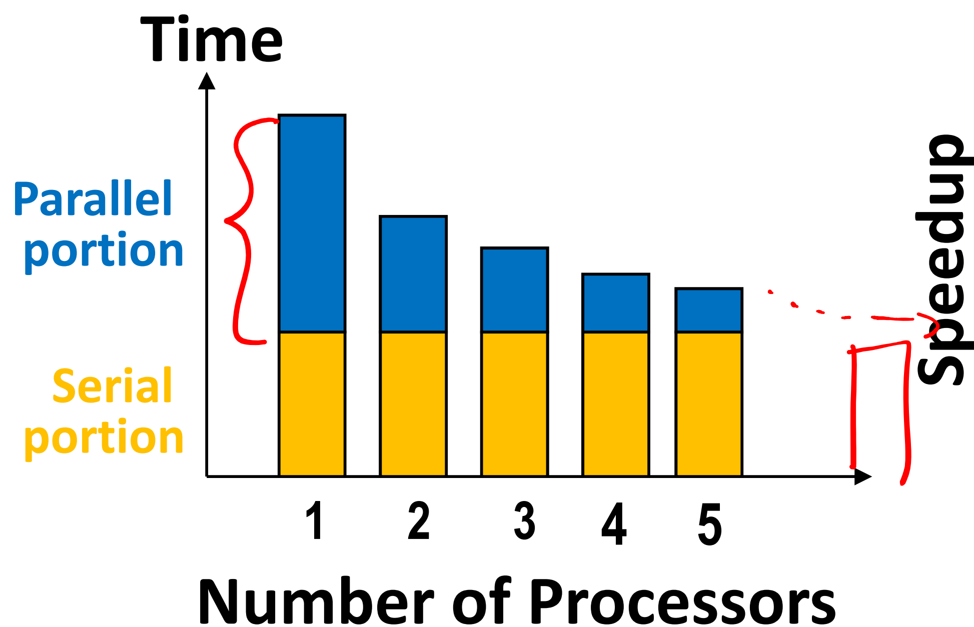
- "52-Card Pickup is a children's game… that is usually played as a practical joke"
  - Scatter a deck of cards, then someone picks up
  - In our variant, want *ordered* deck of cards
- Picking up the cards by yourself takes time
  - A team completes the work more quickly

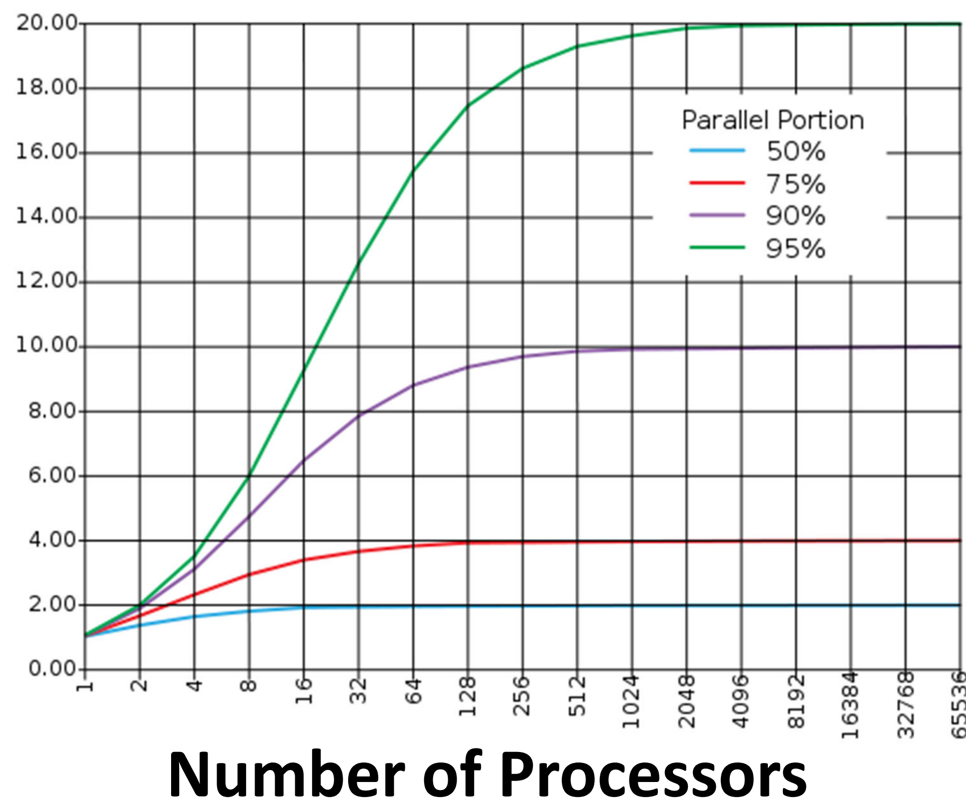
Multithreading Limits

- Why wouldn't a team of 100 people complete the 52-Card Pickup game much faster than a team of just 50?
  - There's only so many things to be done – only so much can be parallelized
  - Some tasks need to wait on others – are inherently sequential

Amdahl's Law

- The amount of speedup that can be achieved through parallelism is limited by the *non-parallel* portion
  - Programs can almost never be completely parallelized; some serial portion remains





Multithreading: Synchronization

- Two memory accesses form a *data race* if different threads access the same location, and at least one is a write, and they occur one after another
  - Means that the result of a program can vary depending on chance (which thread ran first?)

- Avoid data races by *synchronizing* writing and reading to get deterministic behavior

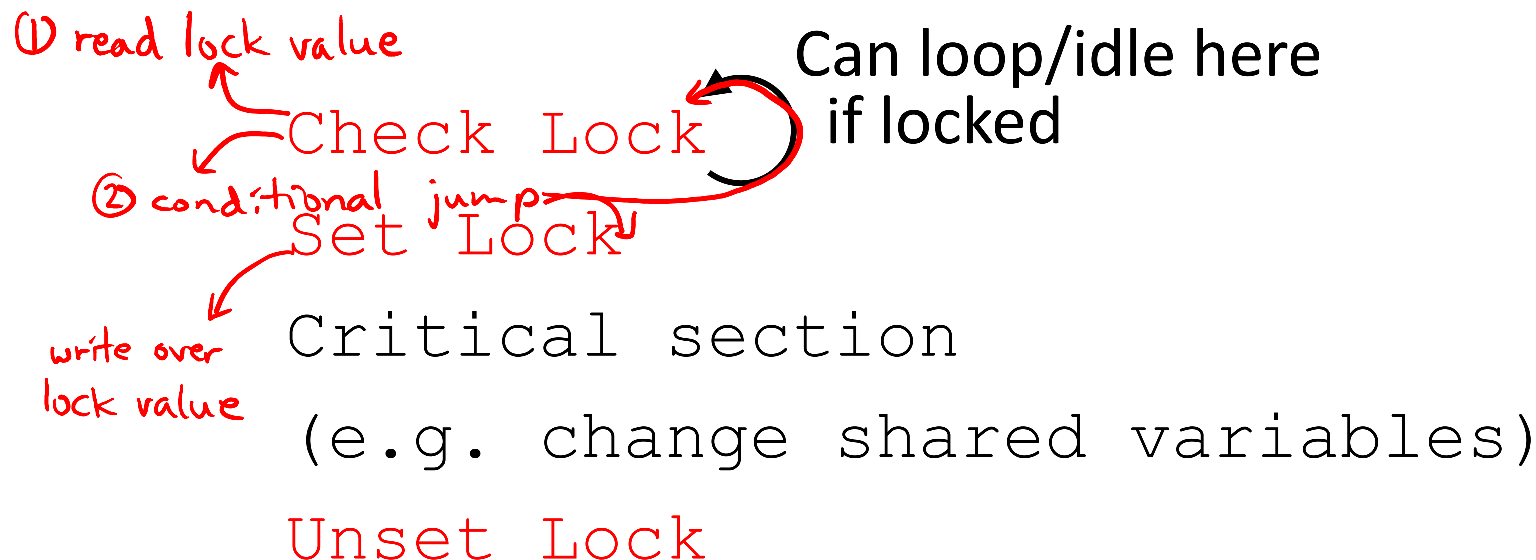
Data Race Analogy: Buying Milk

- Your fridge has no milk:
  - You and your roommate will return from classes at some point and check the fridge
  - Whoever gets home first will check the fridge, go and buy milk, and return
- What if the other person gets back while the first person is buying milk?
  - You've just bought twice as much milk as you need!
- It would've helped to have left a note…

Lock Synchronization

- Use a "Lock" to grant access to a *critical section* so that only one thread can operate there at a time

- Pseudocode:



Lock Problem

- Thread 1

```
read lock (check)

conditional jump

write lock (set)

critical section…
```

- Thread 2

```
read lock (check)

conditional jump
write lock (set)

critical section…
```

Time

unlocked!
lock

Both threads think they have set the lock!
Exclusive access not guaranteed!

Hardware Synchronization

- Hardware support is *required* to prevent an interloper (another thread) from changing the value
  - *Atomic* read/write memory operation
  - No other access to the location allowed between the read and write

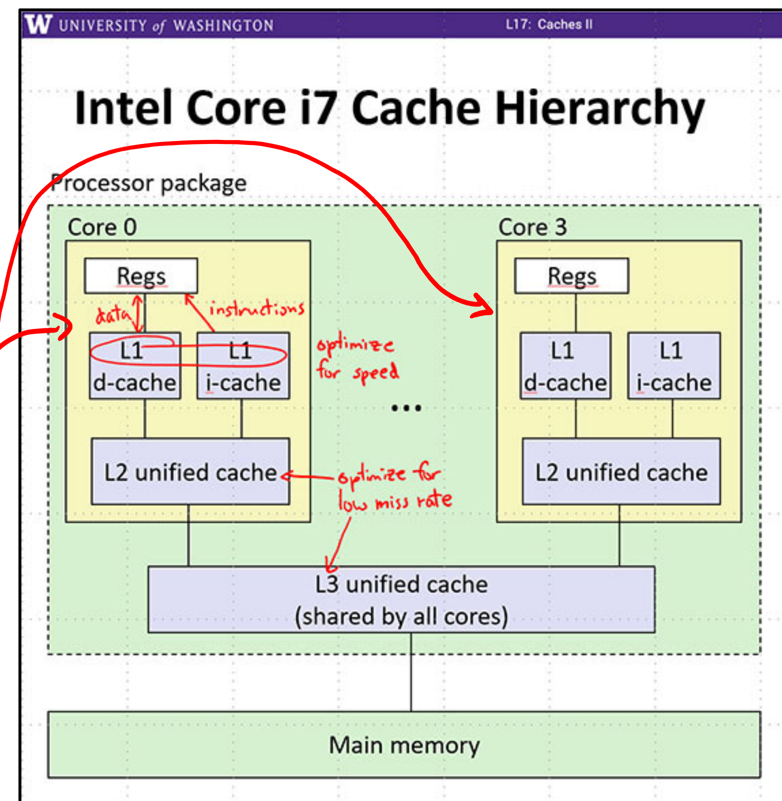
- One idea:
  - When you Read Lock, save a copy in hardware
  - When you try to Set Lock, stop if the Lock value has changed

(1) read value
(2) save value elsewhere in hardware

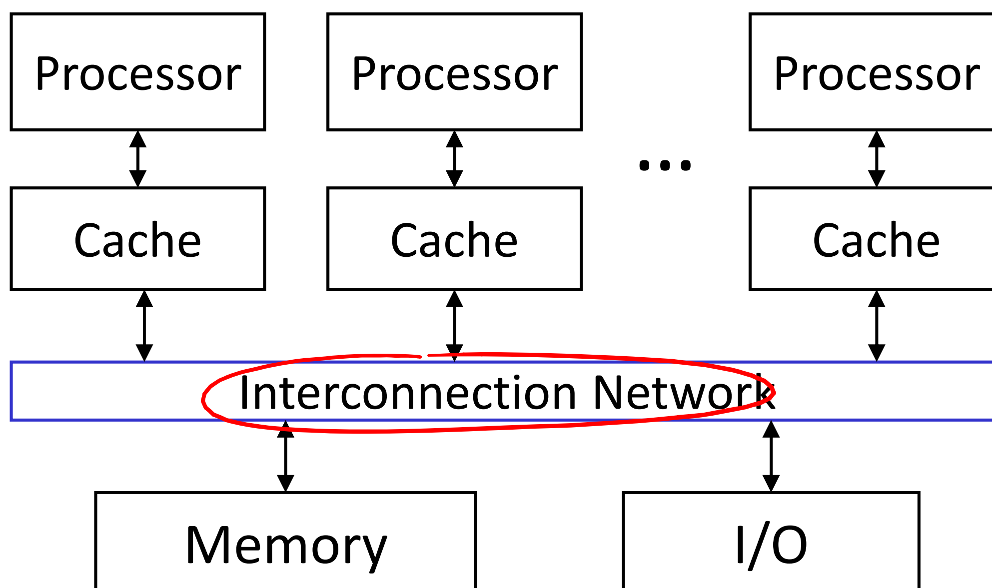
(1) re-read value
(2) write if no change

Multiprocessing: Caches

- Each core shares the same Memory, but has local L1 (and sometimes L2) caches:

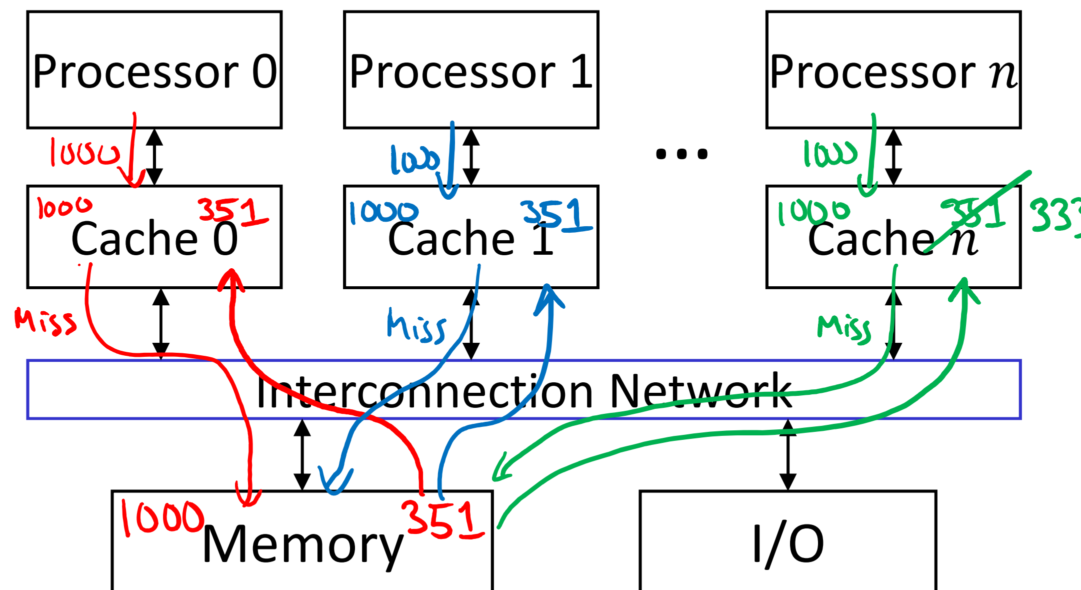


- Logically:



Shared Memory and Caches Scenario

- What if?
  1) Processor 0 reads Mem[1000] (value 351)
  2) Processor 1 reads Mem[1000]
  3) Processor n writes Mem[1000] with 333



Keeping Multiple Caches Coherent

- Architect's job: keep cache values coherent with shared memory
- Idea: on cache miss or write, notify other processors via interconnection network
  - If reading, many processors can have copies
  - If writing, *invalidate* all other copies
- 4th "C" of cache misses: coherence miss!

Types of Parallelism

- Why Parallelism?
- Thread-Level Parallelism
- **Data-Level Parallelism**
  - **SIMD**
  - **Loop Unrolling**
- Instruction-Level Parallelism

Data-Level Parallelism

- *Data-Level Parallelism*: Executing one operation on multiple data "streams"

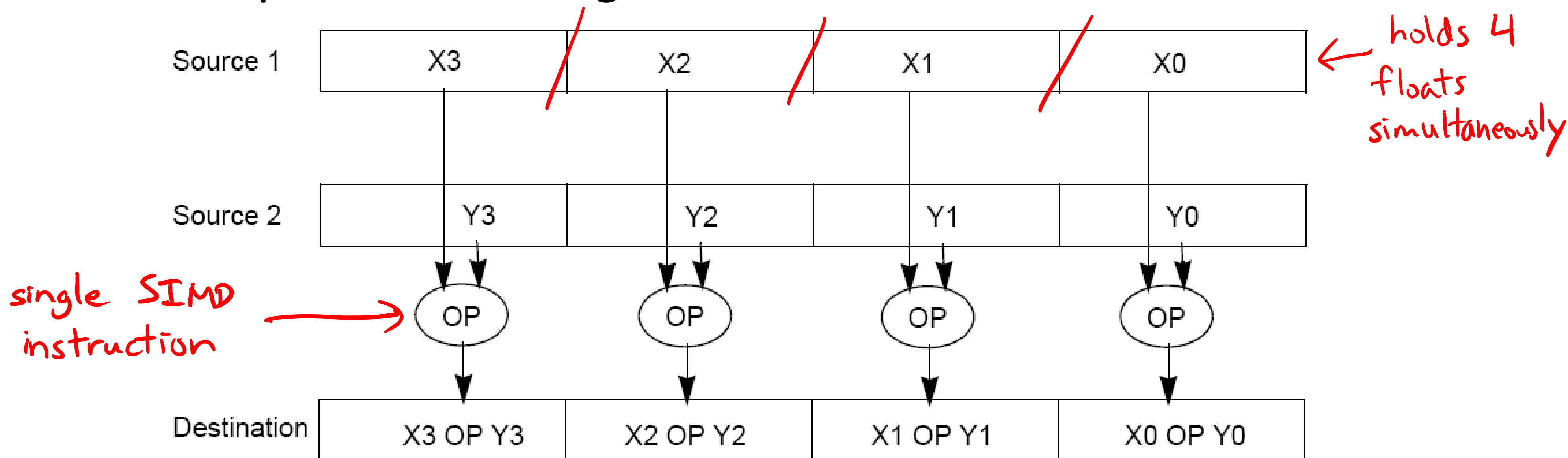
- **Examples:** Vector dot product (*e.g.* in filtering) or matrix multiply (*e.g.* in image processing)

$$y[i] \coloneqq c[i] \times x[i], \;\; 0 \le i < n$$

- Sources of performance improvement:
  - Single instruction for entire operation
  - Each operation is *independent*
  - Concurrency in memory access as well (pull all data at once)

Intel's Streaming SIMD Extensions (SSE)

- *SIMD*: single instruction, multiple data
- SSE is a SIMD instruction set for x86
  - Mostly for `float` data (digital signal & graphics processing)
  - Uses special `%xmm` registers that are 128 bits wide



  - If your machine supports SSE, `gcc` may add these automatically! (try -O3)

Taking Advantage of SIMD

- SIMD wants adjacent values in memory that can be operated in parallel
  - These are usually specified in programs as loops:

(C shown, but actually happening in assembly via the compiler)

```
for(i=0; i<1000; i++)
    x[i] = x[i] + s;
```

- How can we reveal more data level parallelism than is available in a single iteration of a loop?
  - *Unroll the loop* and adjust iteration rate:

```
for(i=0; i<1000; i+=4) {
    x[i]   = x[i]   + s;
    x[i+1] = x[i+1] + s;
    x[i+2] = x[i+2] + s;
    x[i+3] = x[i+3] + s;
}
```

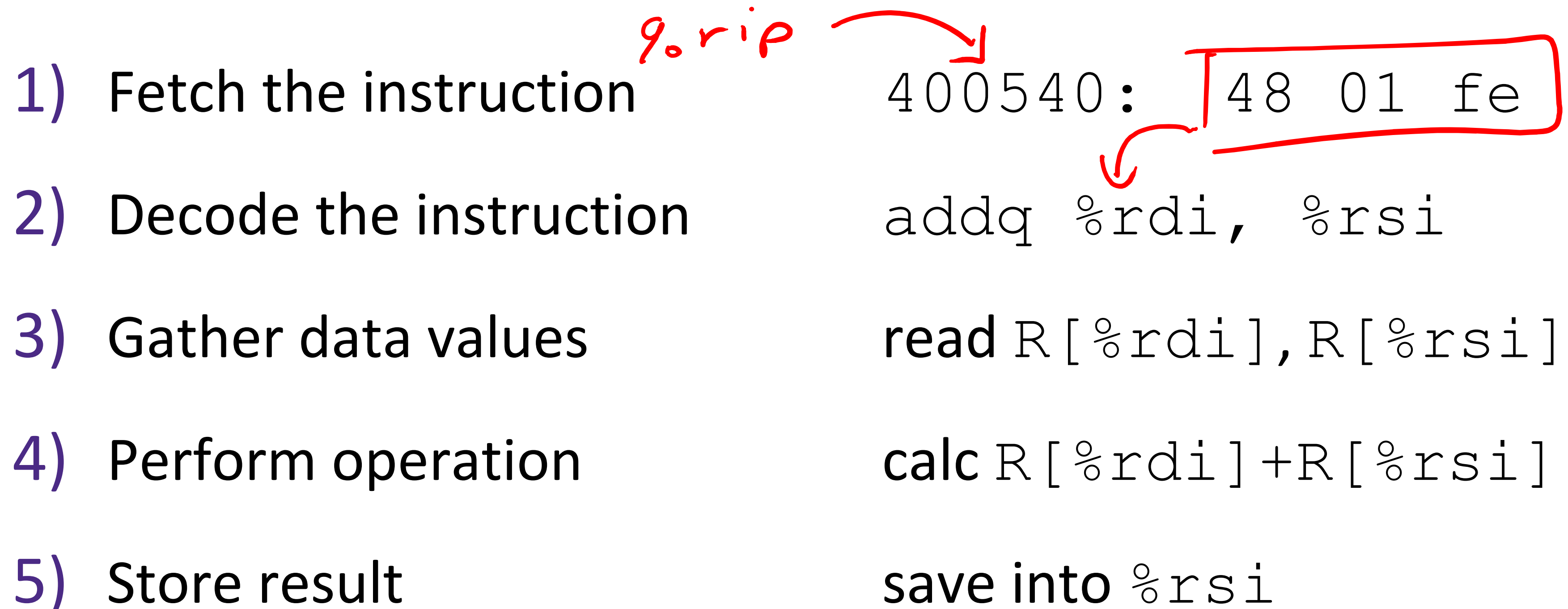
replaced by 1 SIMD instruction

Types of Parallelism

- Why Parallelism?
- Thread-Level Parallelism
- Data-Level Parallelism
- **Instruction-Level Parallelism**
  - **Pipelining**

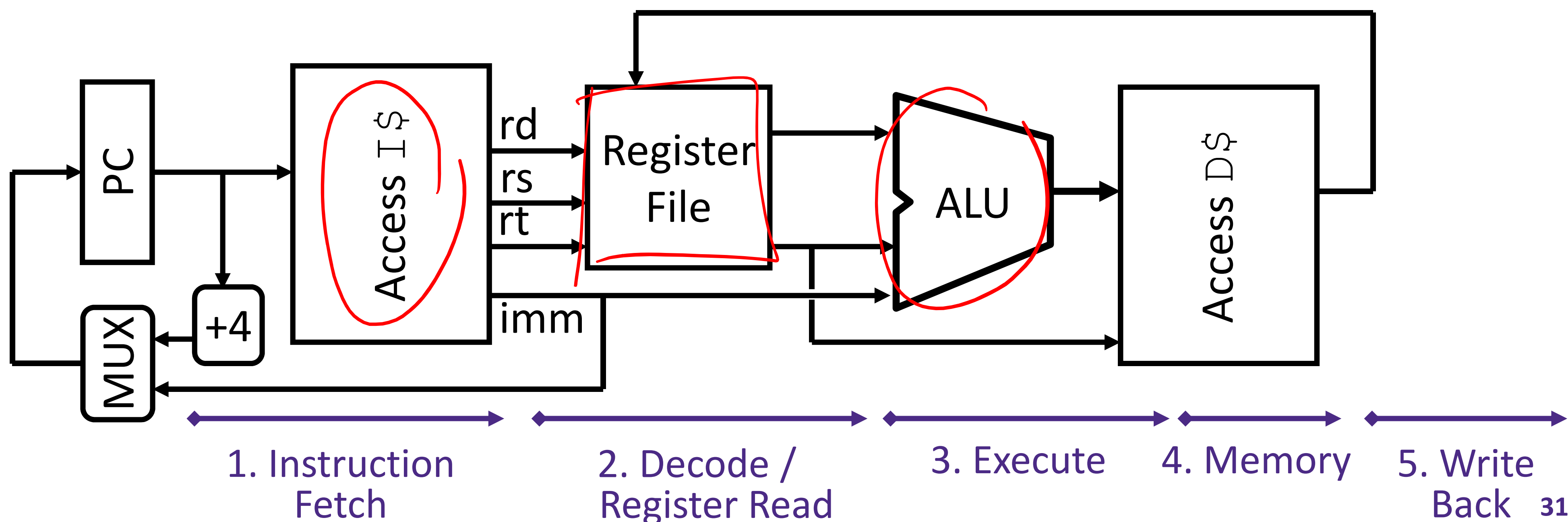
Executing an Instruction

- Very generally, what steps do you take to figure out the result of the next x86-64 instruction?



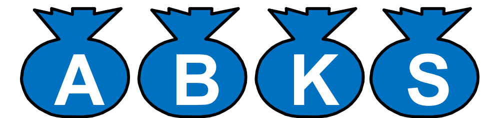
Datapath

- *Datapath*: part of the processor that contains the hardware necessary to perform operations required by the processor ("the brawn")
  - Each "stage" of instruction execution is roughly associated with a piece of the datapath
  - Each hardware piece can only perform one action at a time

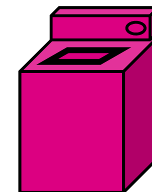


Analogy: Doing Laundry

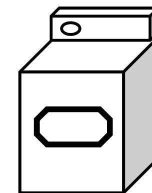
- **A**n, **B**rian, **K**ory, and **S**ophie each have one load of clothes to wash, dry, fold, and put away



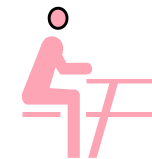
  - Washer takes 30 minutes



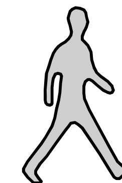
  - Dryer takes 30 minutes



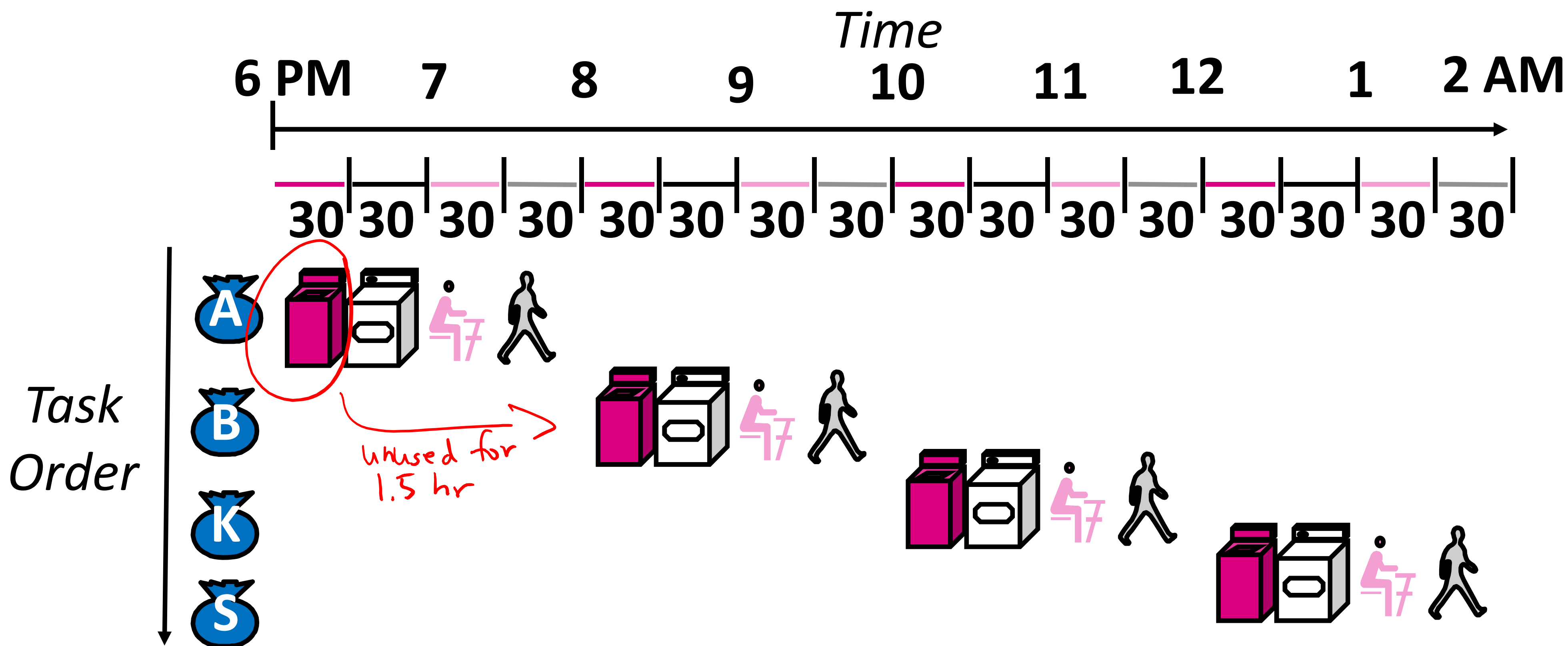
  - "Folder" takes 30 minutes



  - "Stasher" takes 30 minutes to put clothes into drawers

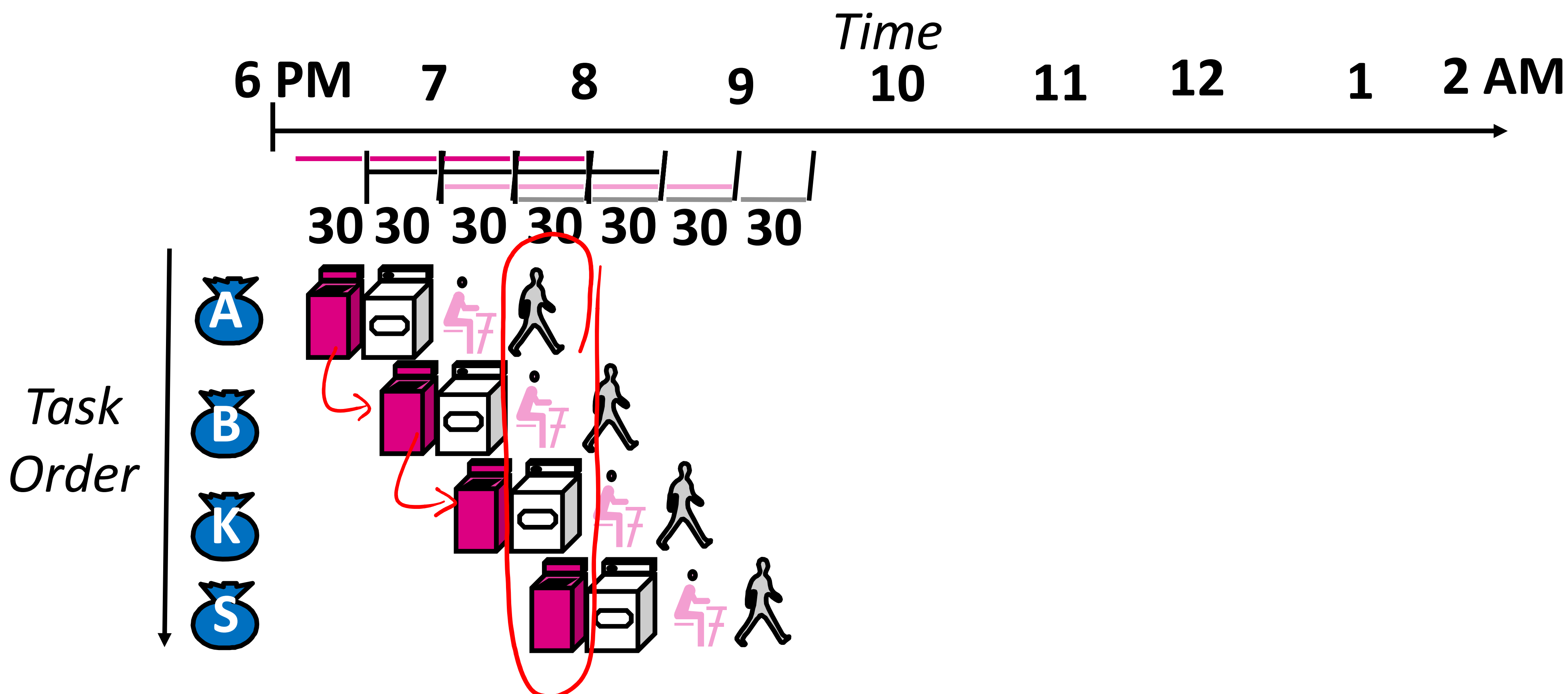


Sequential Laundry



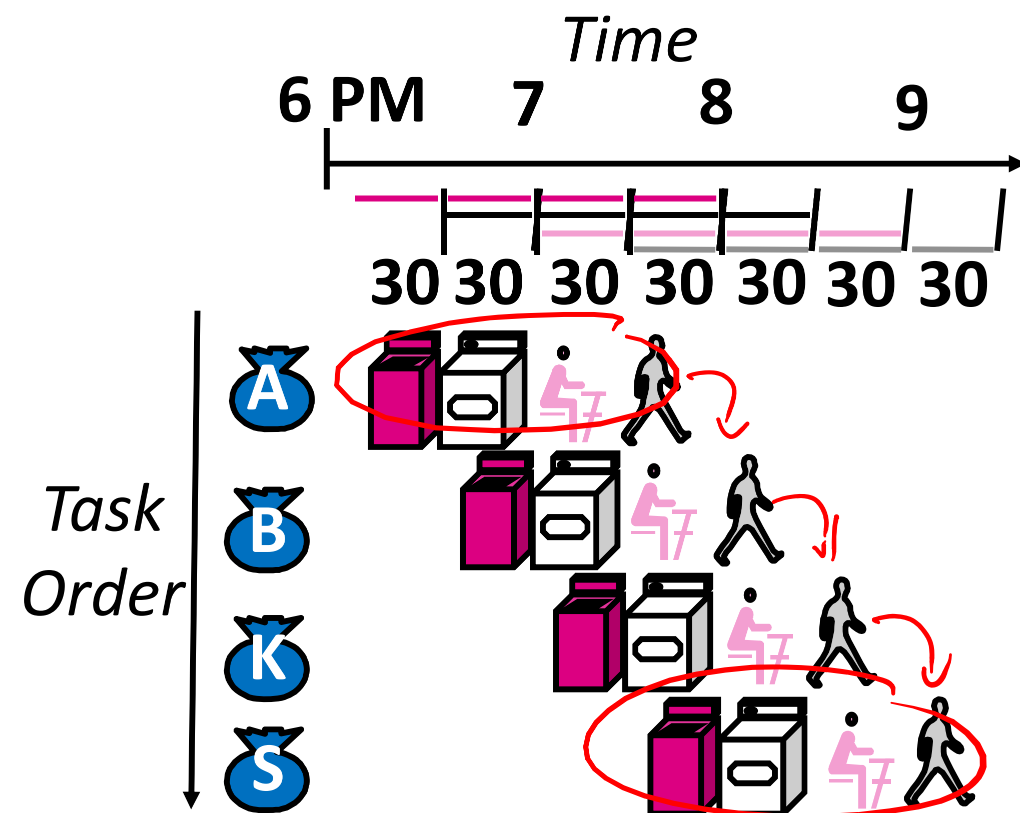
- Sequential laundry takes 8 hours for 4 loads

Pipelined Laundry



- Pipelined laundry takes 3.5 hours for 4 loads!

Pipelining Notes



- Pipelining doesn't help *latency* of single task, just *throughput* of entire workload
- *Multiple* tasks operating simultaneously using different resources
- Potential speedup = number of pipeline stages

- Pipelining allows us to execute parts of multiple instructions at the same time using the same hardware!
  - This is known as *instruction-level parallelism*

Multiple Issue

- With extra copy of main components of datapath, it's possible to issue multiple instructions simultaneously!
  - Need to make sure that simultaneously-executing instructions are dependent on each other

- A processor that can execute more than one instruction per clock cycle is called *superscalar*

- Even crazier: out-of-order execution

Summary

- In the pursuit of processing power, parallelism is the most promising path!
  - Requires specialized hardware and programming techniques
  - Lots of potential issues, so difficult to get right
- Many kinds of parallelism that can be used in conjunction with each other:
  - Thread-level parallelism (TLP)
  - Data-level parallelism (DLP)
  - Instruction-level parallelism (ILP)
  - … many other kinds not mentioned today!