Concurrency

Kelly Rivers and Stephanie Rosenthal

15-110 Fall 2019

Announcements

- Exam on Friday
- Homework 5 check-in due Monday

Big Picture

- Unit 1 - basic programming and how a computer works
- Unit 2 - organizing data, new algorithms based on those representations, and comparing/contrasting/analyzing the efficiency of the different algorithms
- Unit 3 – scaling up computing for larger tasks

Unit 3

- So far, we have assumed we have one processor (computer) executing algorithms to completion, and use one place to store our data
- This isn't really how things work today.
- In this unit, we'll answer questions like:
  - How does it work for you to run multiple applications on your computer at once?
  - How does your computer use multiple processors?
  - How does the internet work (so many computers working together)?
  - How does a company like google store all their data about websites?
  - How can you write programs that work on that much data?

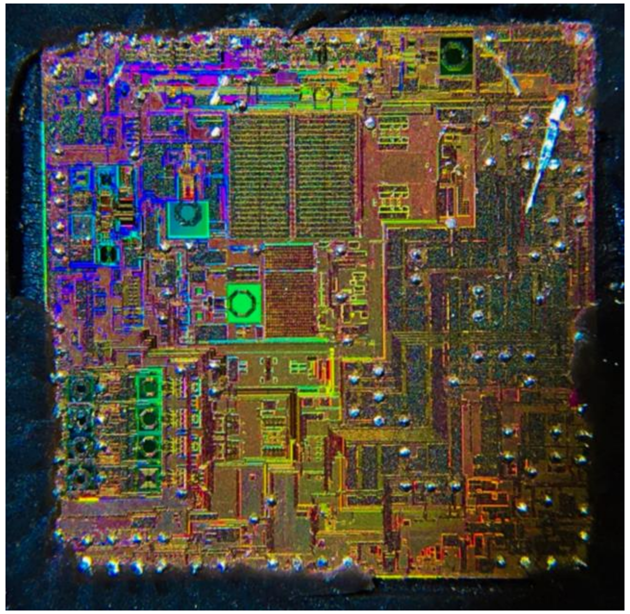
A Central Processing Unit (CPU)

In particular, the Central Processing Unit (CPU) or processor has a lot of circuits and runs your programs. It contains:

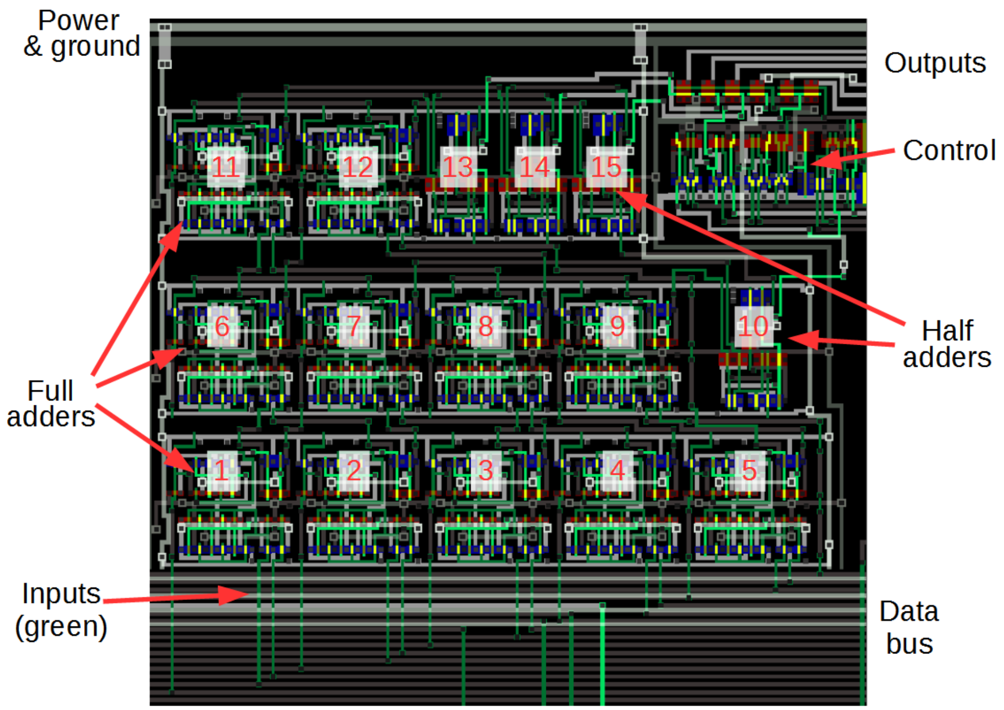
- Arithmetic Logic Unit (ALU) to perform computation (adders, multipliers, etc)
  - There isn't just one adder on a computer, there are hundreds or thousands of them and it doesn't matter which one does the addition
- Registers to hold information (small pieces of information)
  - Bytecode Instruction register (current instruction being executed)
  - Program counter (to hold location of next instruction in memory)
  - Accumulator (to hold computation result from ALU)
  - Data register(s) (to hold other important data for future use)
- Control unit to regulate flow of information and operations that are performed at each instruction step
  - The CPU has to get the data from the stack or the memory of the computer into the right gates and circuits and then send the results back again to memory (stores and loads, adds)

A Real CPU

There are many of each type of circuit in a CPU of a computer so that the computer can use each of them for different things at every time step. For example, there are many adders.



A CPU



Concurrency

Concurrency – doing multiple tasks at the same time (simultaneously)

We can accomplish computational concurrency

on a single computer using redundant circuits like adders

or by using multiple computers

Concurrency on a single CPU

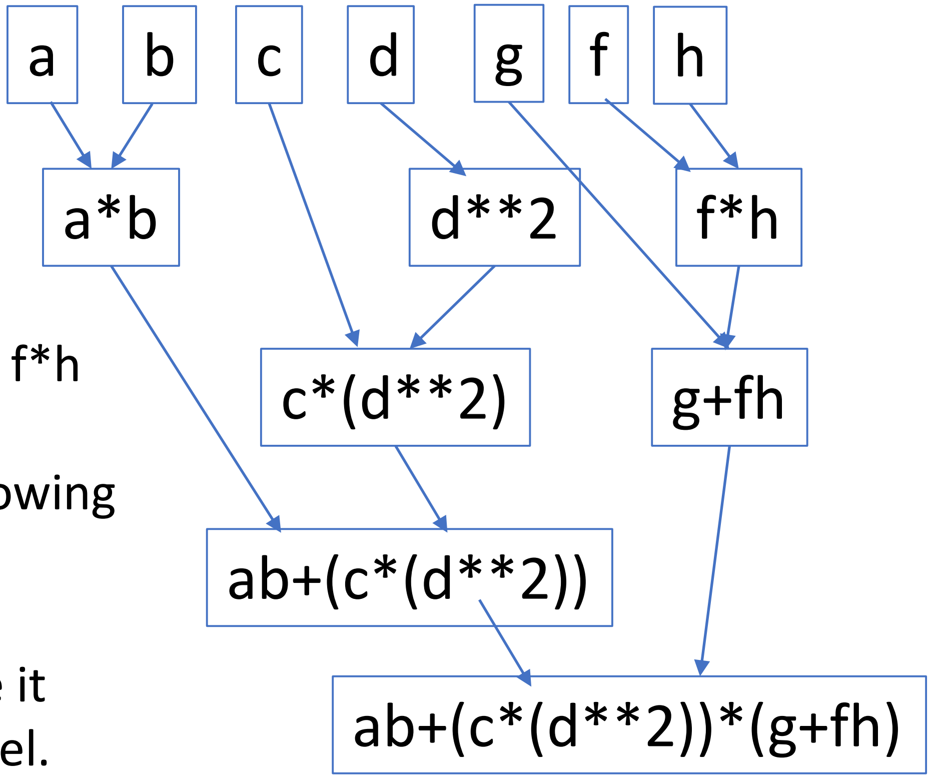
Because the CPU can direct different bits to different adders all at the same time, we no longer have to assume that only one addition happens at a time.

Concurrency happens on a CPU at a circuit-level. The computer can decide to add many things at the same time (and do many other tasks simultaneously). By doing tasks at the same time, it makes your computer appear faster

Concurrency Trees and Math Operations

A concurrency tree shows the computations that can be done simultaneously (at the same level) and which are needed together to get a new result for the next level.

e.g. $(ab+cd^2)(g+fh)$



a*b can be computed at the same time as d**2 and f*h
Instead of taking 7 time-steps to compute (one per multiply or add), it only takes 4 time-steps when allowing for concurrency.

Each level cannot be computed any sooner, because it depends on at least one value from the previous level.

Resource Constraints

There are many parts of a computer that can run simultaneously because they each require different resources.

Getting data from memory

Adding/multiplying two numbers on the stack

Displaying on the screen

If two operations do not use the same resources, there is no reason they cannot be run concurrently.

Resources and Yielding

If two operations require the same resource, one part of the program has to yield to the other (stop running) so the other can use it first.

Example: Suppose two tasks want to print to the screen. One has to wait for the other to print so that you don’t get jumbled text:

Task1 wants to print "Hello World"

Task2 wants to print "15-110 is great"

If one doesn’t yield to the other, you could get "Hello 15-110 World is great"

Deadlock Example

Task1 wants to print "Hello World" and use Adder1 to add 2+2 and print the answer

Task2 wants to use Adder1 to add 3+3 and print the answer

Task1 gets to use the screen first. Task2 gets to use the Adder first

But now they also both need the other resource.

They are Deadlocked because Task1 cannot continue until it gets Adder1 and Task2 won’t give up Adder1 until it gets the screen, which Task1 has

Deadlock

Deadlock is the condition when

A) two or more tasks are all waiting for some shared resource, but no task actually has it to release

B) two or more tasks are each using a resource the other needs while also not giving up their own resource

All tasks wait forever without proceeding

Deadlock is like Gridlock

Gridlock is a deadlock:

The cars are each using the road and wont move back, but they are also blocked and can’t continue forward

Solving Deadlock

Task1 wants to print "Hello World" and use Adder1 to add 2+2 and print the answer

Task2 wants to use Adder1 to add 3+3 and print the answer

If the two tasks had been programmed to always first request the adder and then the screen, then one task will yield for both requests until the other is finished with both.

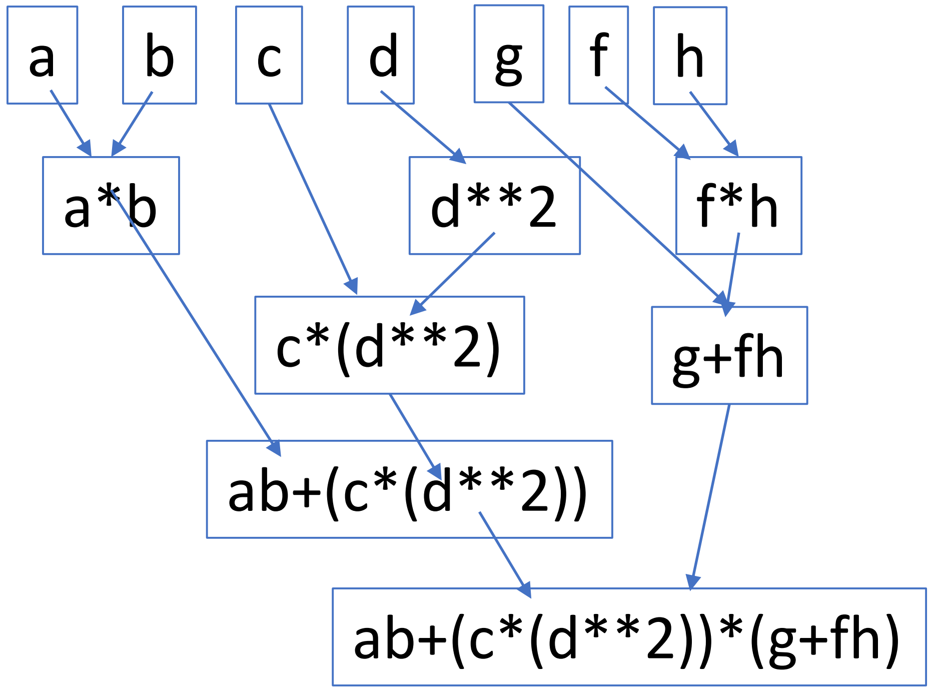
To reduce the chance of deadlock, the programmer decides an order of resources to request and ensures every task requests in that order.

Pipelining

We’ve thought about concurrency for doing a single task by breaking it into subtasks. What if that task is repeated?

e.g., if we had many values of a-h, do we have to compute the whole expression for one set of values before starting with the next set?

Can we model the amount of time repeated tasks would take when the subtasks have dependencies?



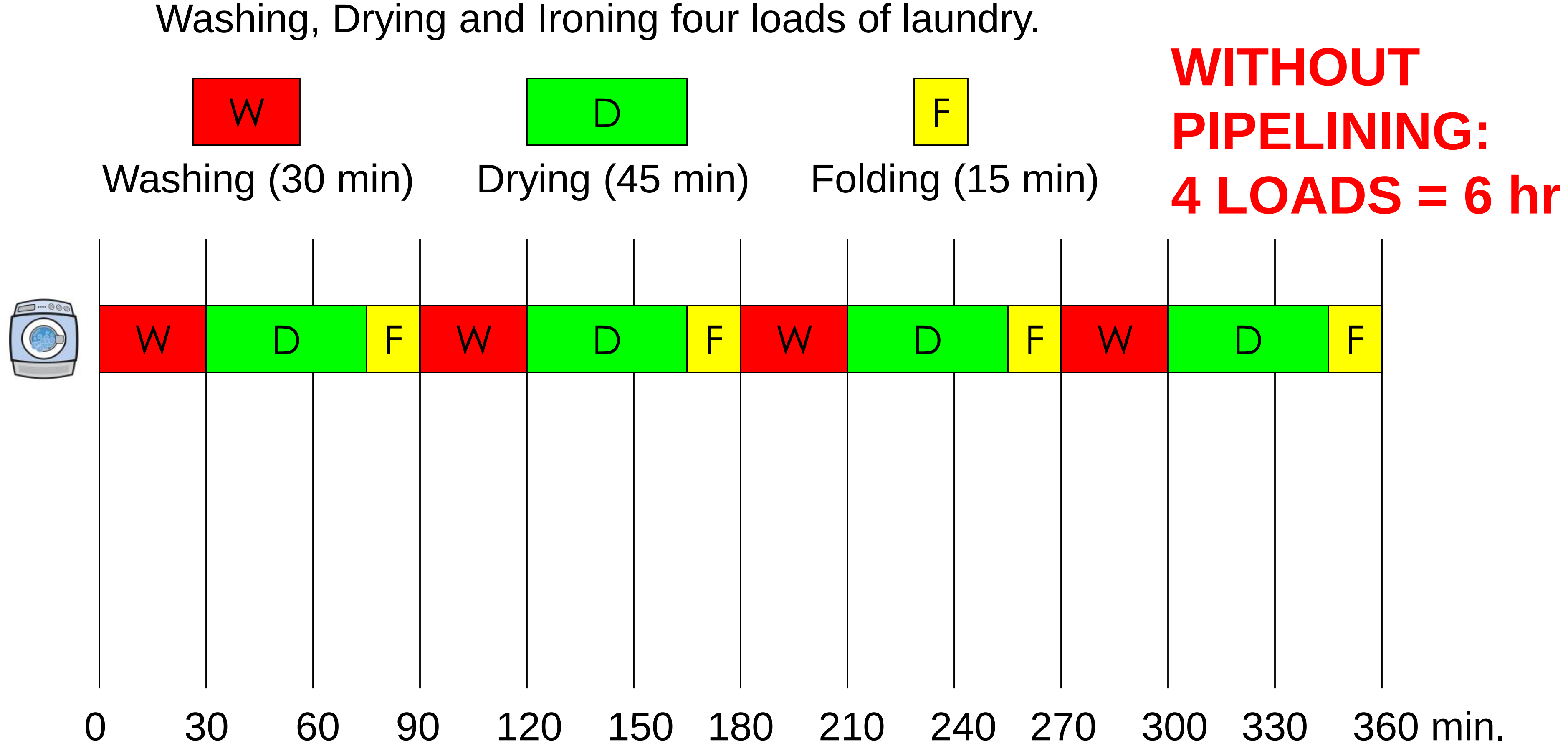
Pipelining

Pipelining is the process of splitting a task into a series of subtasks where each task relies on data modified by the task before it

- It's similar to an assembly line.

Instead of completing one object's computation before starting another, each computation is split into simpler sub-steps, and computations are started as others are in progress.

Example: Laundry without Pipelining



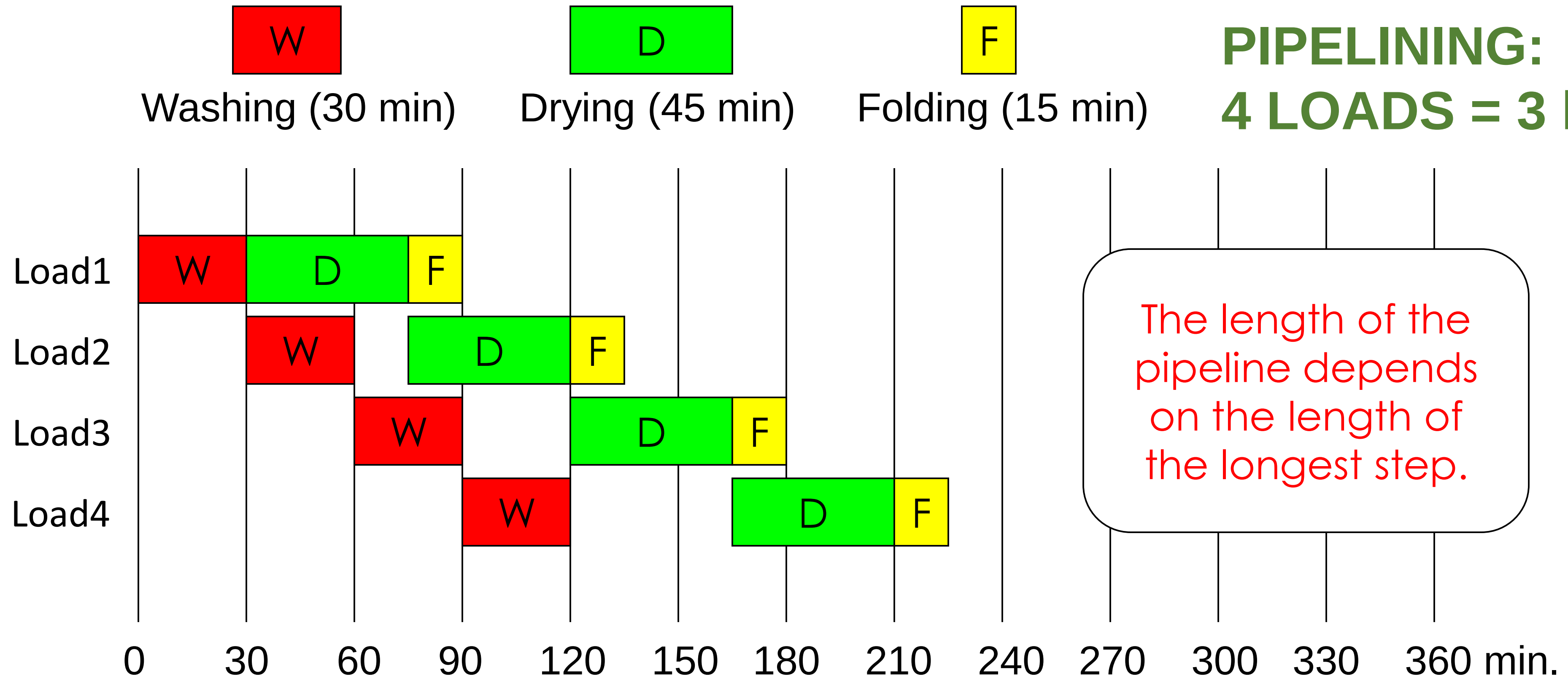
Washing, Drying and Ironing four loads of laundry.
W
Washing (30 min)
D
Drying (45 min)
F
Folding (15 min)
W D F W D F W D F W D F
0 30 60 90 120 150 180 210 240 270 300 330 360 min.

WITHOUT PIPELINING:
4 LOADS = 6 hr

Example: Laundry with Pipelining

Washing, Drying and Ironing four loads of laundry.

WITH PIPELINING:
4 LOADS = 3 hr 45 min

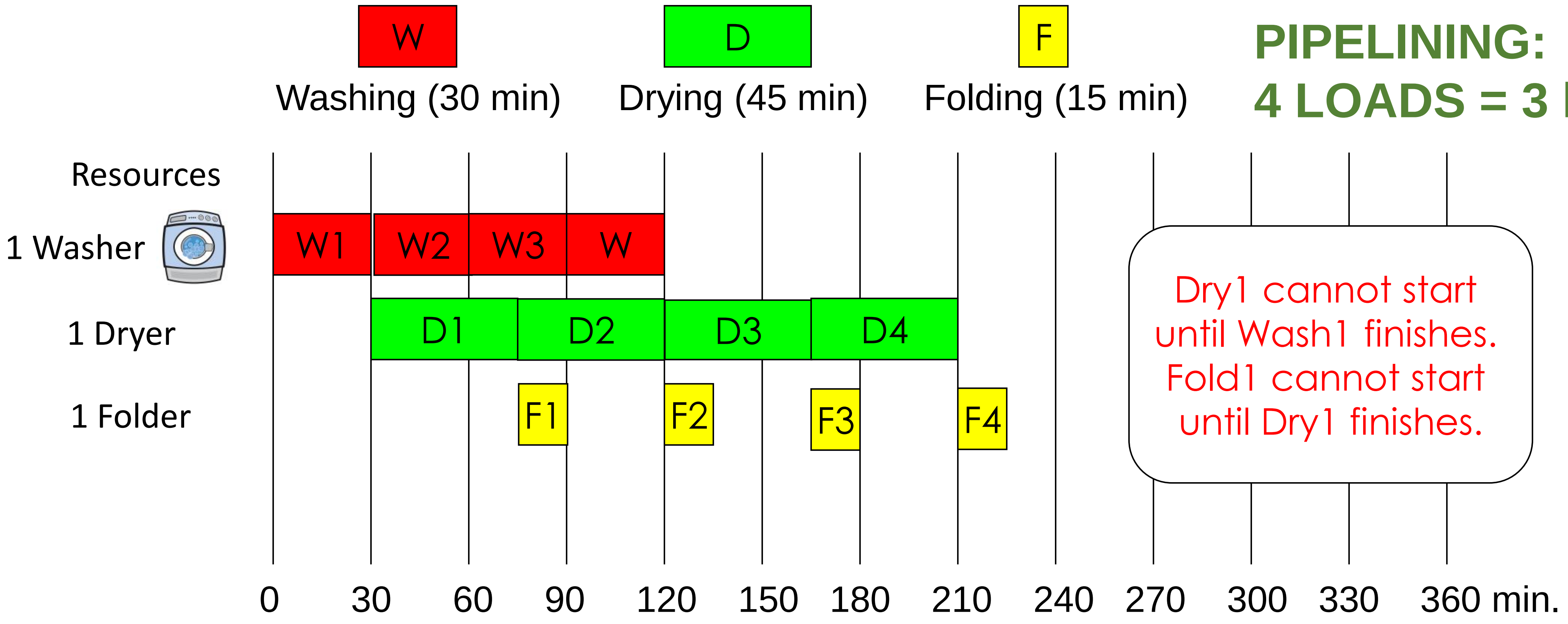


The length of the pipeline depends on the length of the longest step.

Example: Laundry with Pipelining Another Visual

Washing, Drying and Ironing four loads of laundry.

WITH PIPELINING:
4 LOADS = 3 hr 45 min



Dry1 cannot start until Wash1 finishes. Fold1 cannot start until Dry1 finishes.

Pipelining Takeaways

Pipelining helps us understand how sequences of subtasks can be repeated even if the subtasks depend on each other

- The second subtask in a sequence must start after the first ends
- More than one sequence can be in progress at the same time

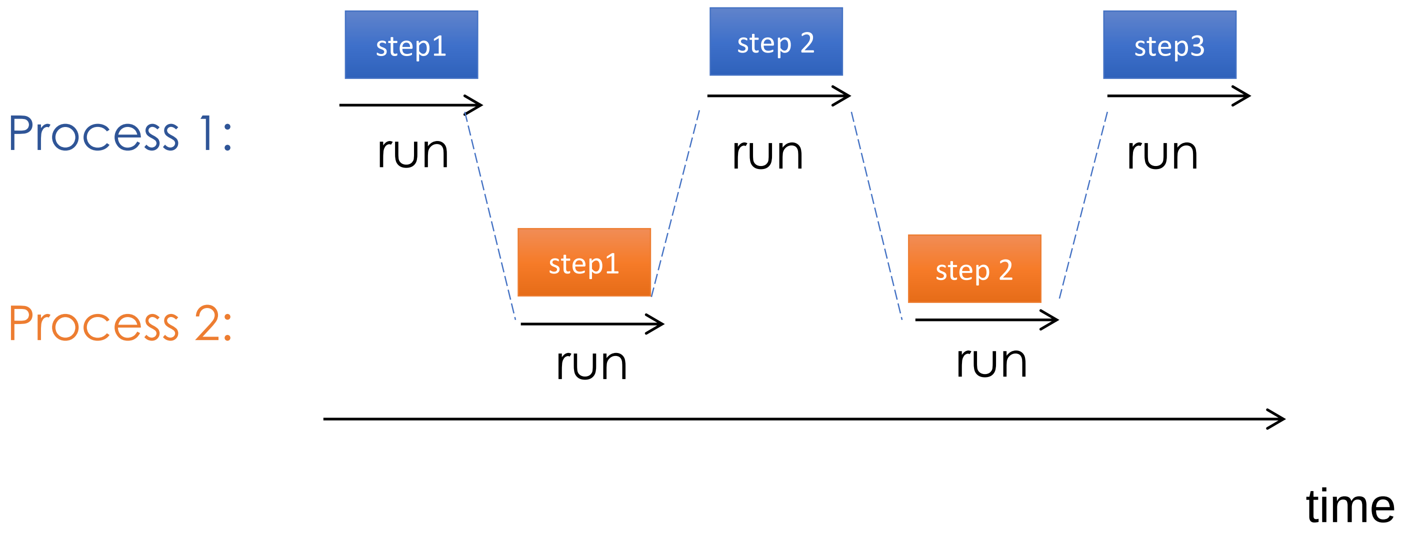
The longest subtask is what dictates how long the pipeline takes

Multitasking

Do different programs (Word, Firefox, Chrome) all run concurrently on your computer? Can you run them at the same time?

One CPU: Multitasking

If only one processor (CPU) is available, the only way to run multiple processes is by switching between them.

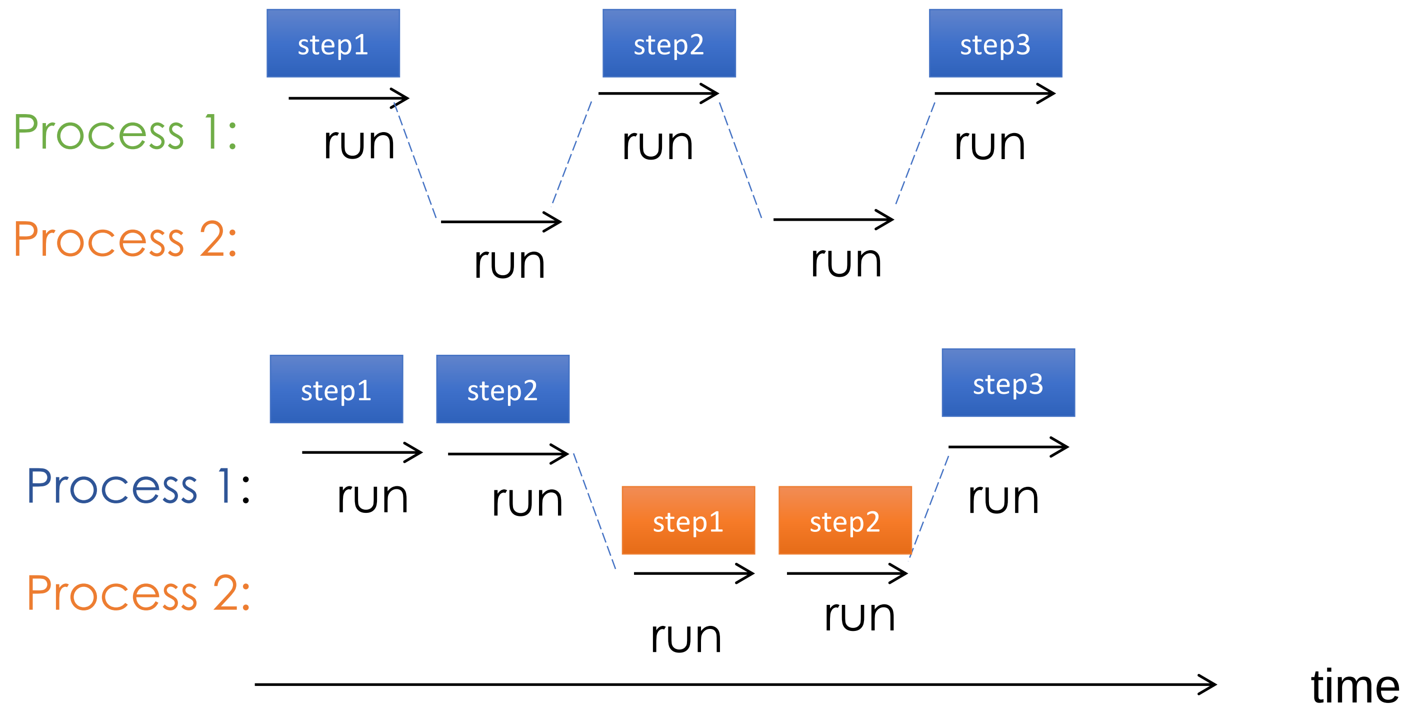


Only one process is using the CPU at a given time even though they look like they are running in parallel to an observer.

One CPU: Scheduling

The order in which the steps are run is determined by a scheduler.

There are many possibilities.



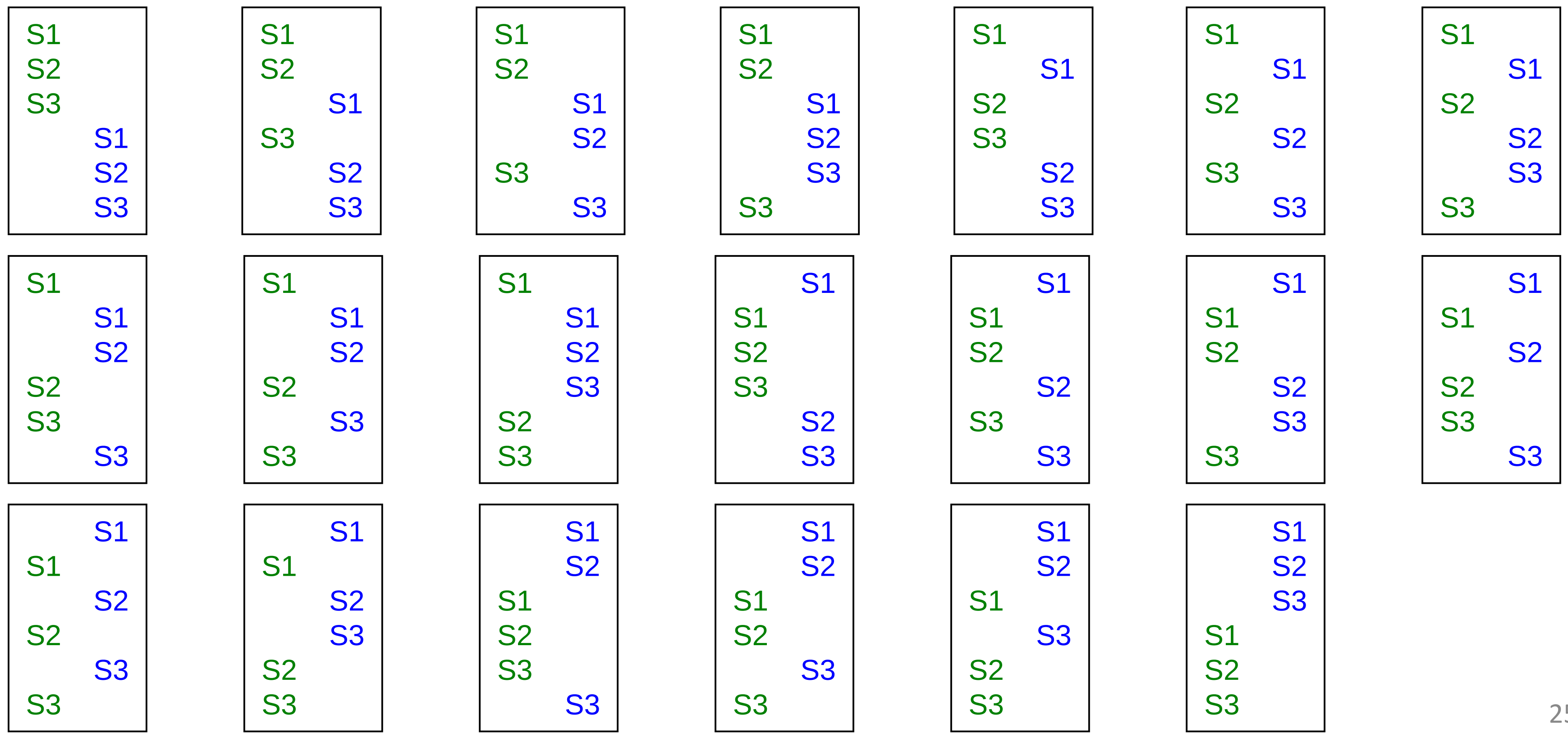
Many ways to multitask with two processes

The green process executes steps
S1 S2 S3 in the given order.
The blue process executes steps
S1 S2 S3 in the given order.

S1
S2
S3

S1
S2
S3

Several possible
interleavings of steps.



Transistors as the switches

A computer sends data to each of the adders (or any other part of a computer) using transistors

Think of transistors as train switches that change a train from one track to another. The transistor does the same thing with bits traveling through the wires.

The CPU decides which circuits different bits need to go to, uses transistors to move data to those circuits

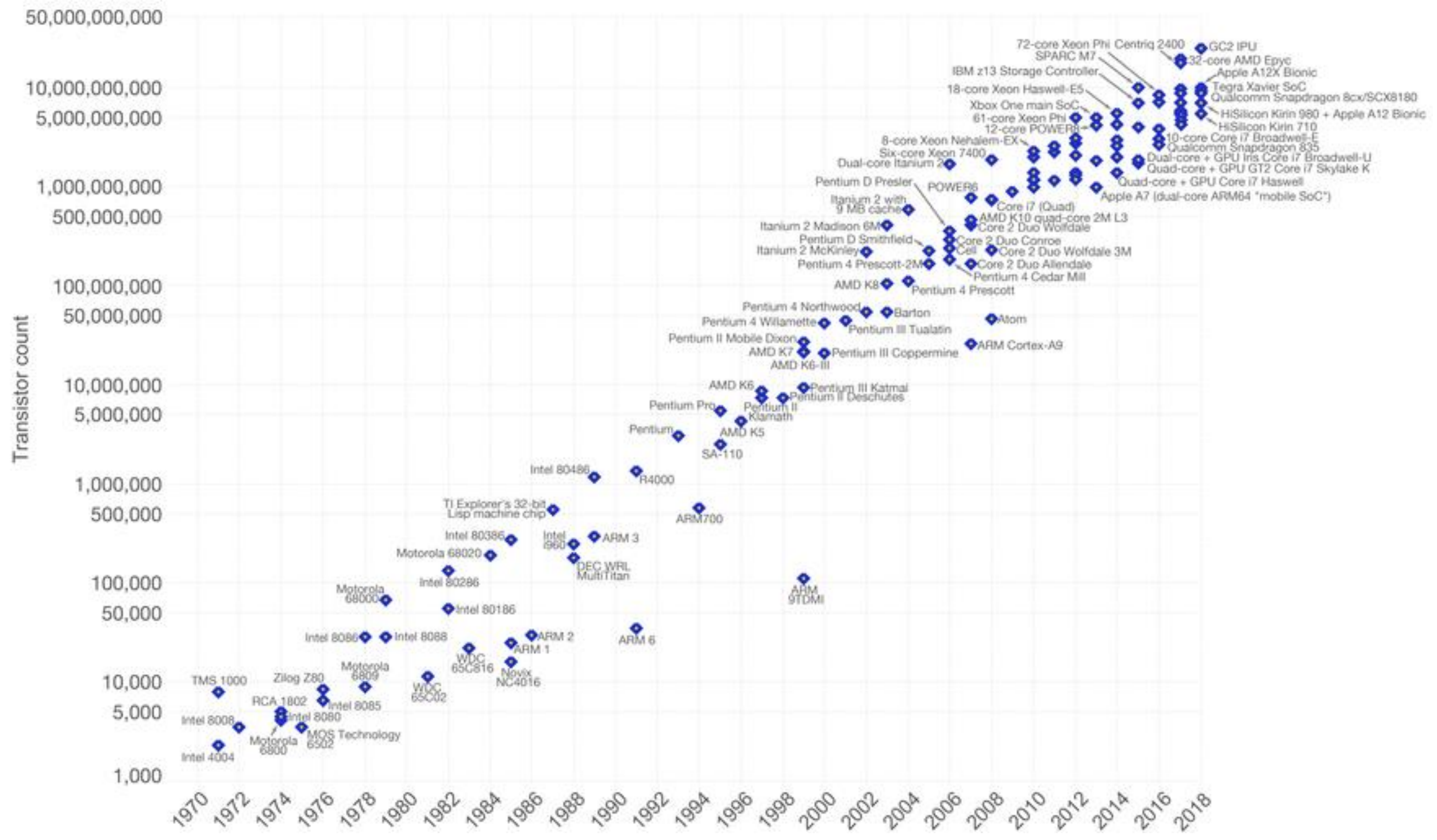
Transistors make computing faster…

Think of transistors as train switches that change a train from one track to another. The transistor does the same thing with bits traveling through the wires.

The CPU decides which circuits different bits need to go to, uses transistors to move data to those circuits. The more transistors, the more places the bits can go and the faster they can get there.

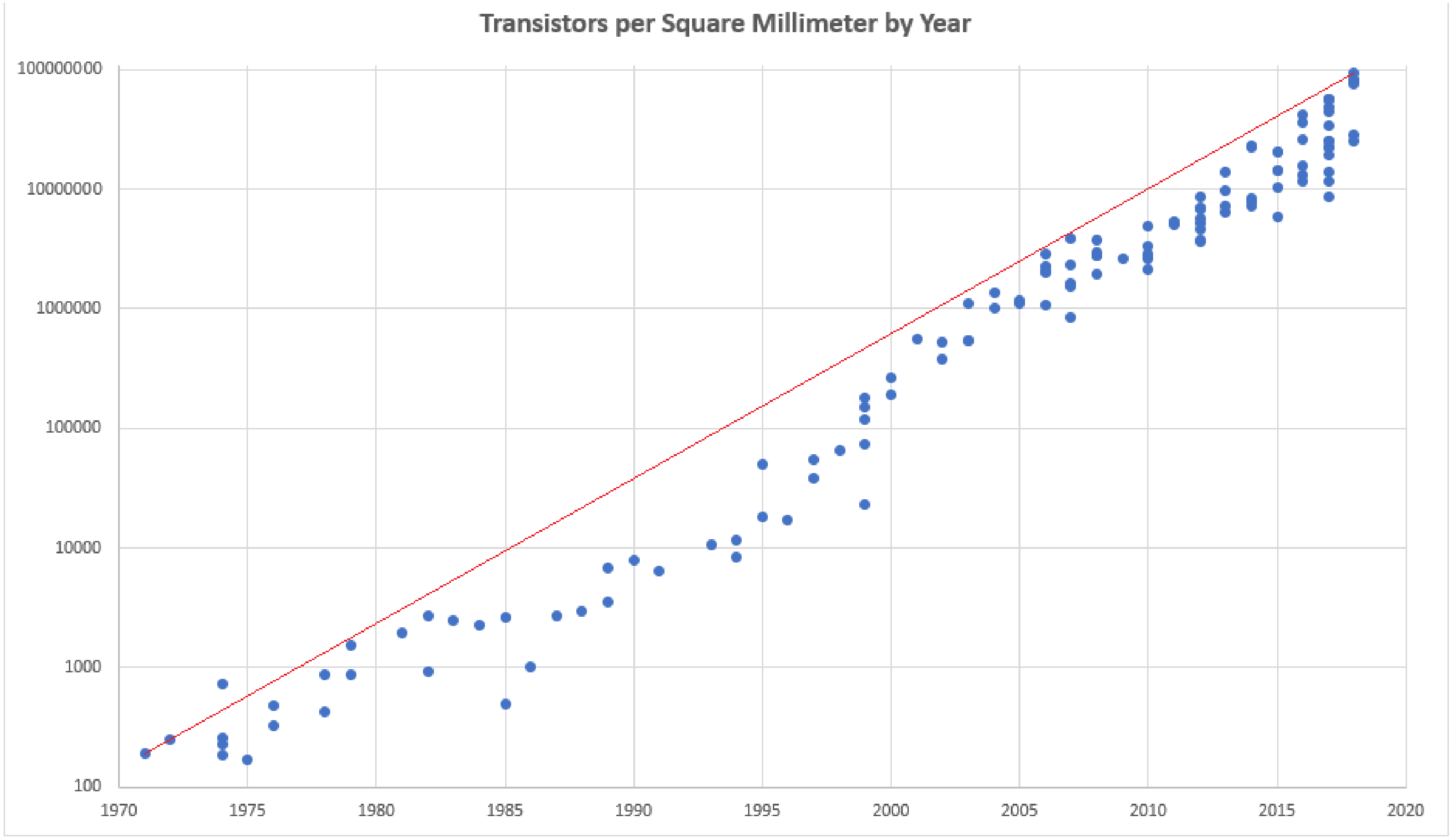
Moore’s Law

The number of transistor on a computer doubles roughly every 2 years



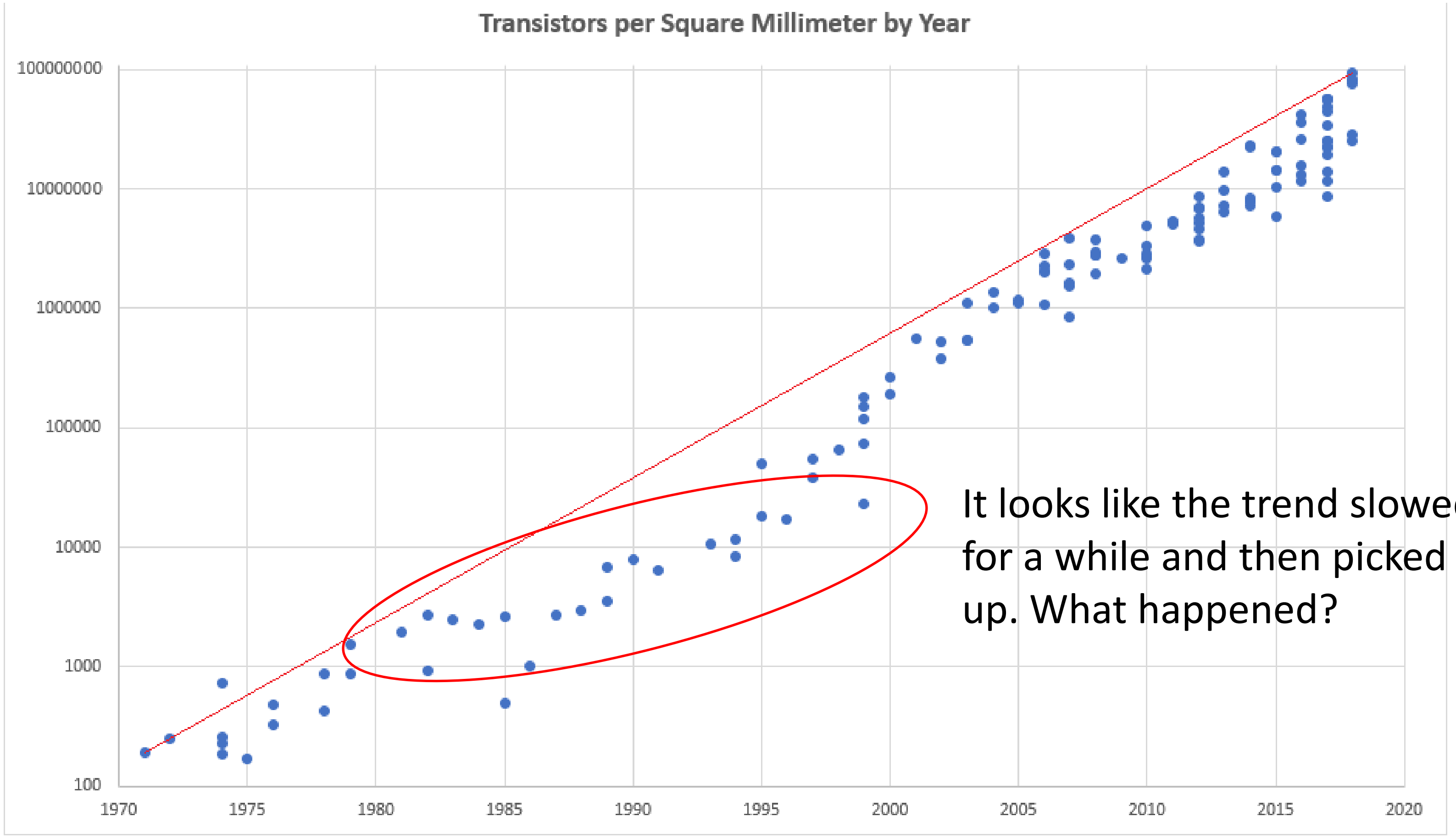
Moore's Law

The number of transistor on a computer doubles roughly every 2 years



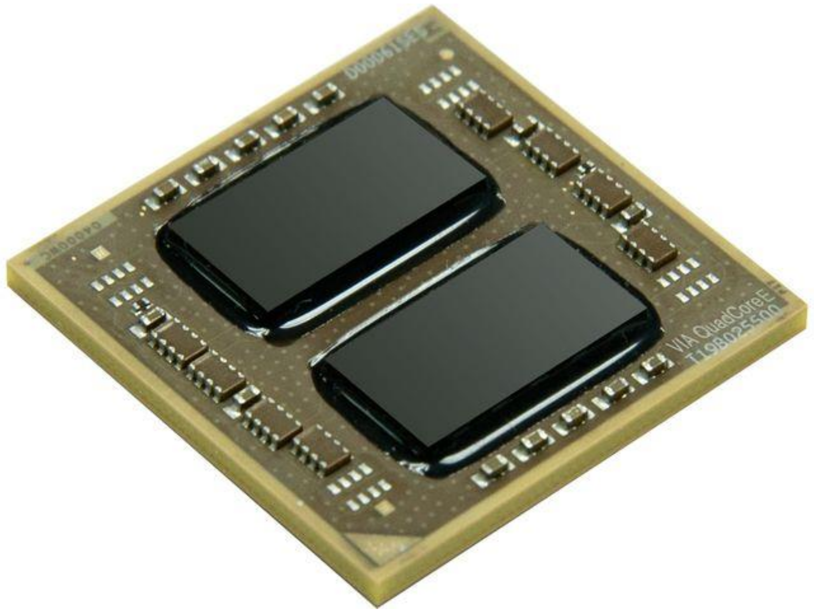
Moore’s Law

The number of transistor on a computer doubles roughly every 2 years



It looks like the trend slowed down for a while and then picked back up. What happened?

Multi-core Processors



Computers now have more than one "core" and each core can do separate computations. This made it possible to continue adding transistors by doubling the number of cores instead of adding more transistors to 1 core.

Concurrency Takeaways

- Concurrent programming splits programs into small tasks so that they can share time on CPUs (central processing units)
- Computers use multiple cores and schedulers to make programs run at the same time and/or share time on the same processor
- This can cause problems like deadlock, which we can solve by being consistent about which resources each tasks requests in what order