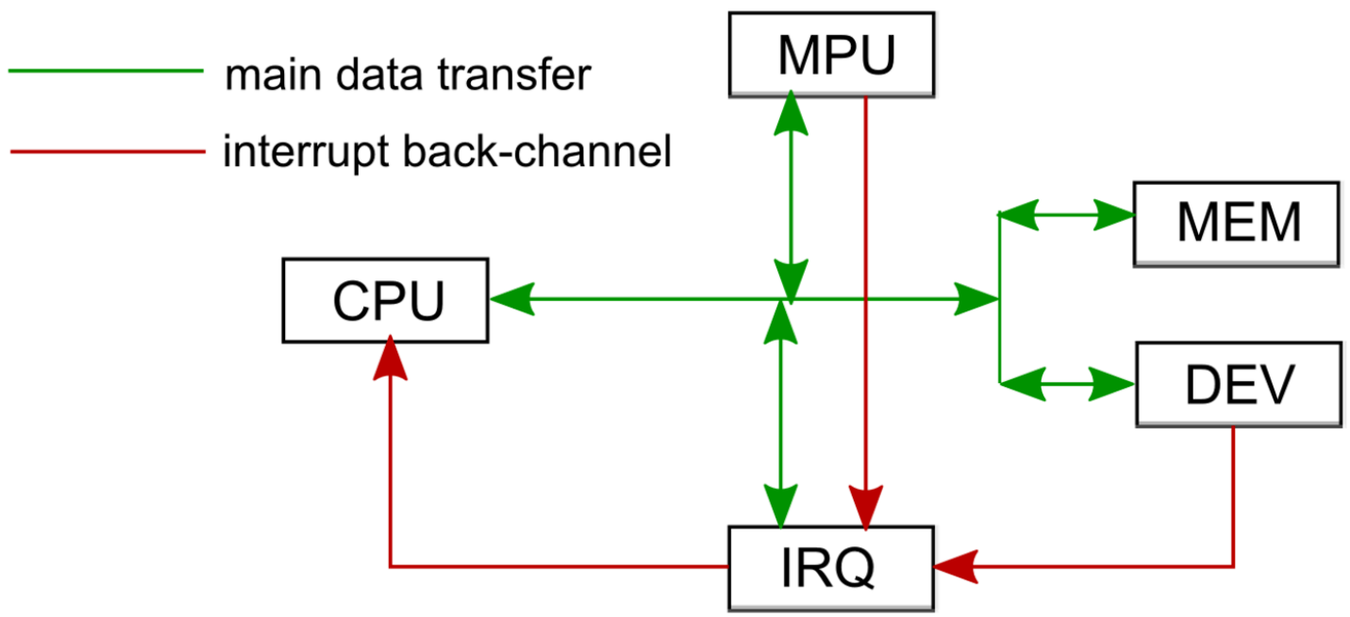
The life-time of a program

- A typical program alternates between times when it is:
  - Doing computation - lots of fast traffic between CPU and MEM
  - Doing Input/Output - short bursts of traffic between CPU and DEV with long waits in-between while **slow** I/O operations take place
- In early days, one program ran at a time
- The idea soon emerged to allow another program do its computation while the first one was waiting for I/O, and then another, and another, …
- The idea of a multi-user machine (a.k.a. "multiprogramming") was born
  - A concurrent system, requiring concurrent operating software.

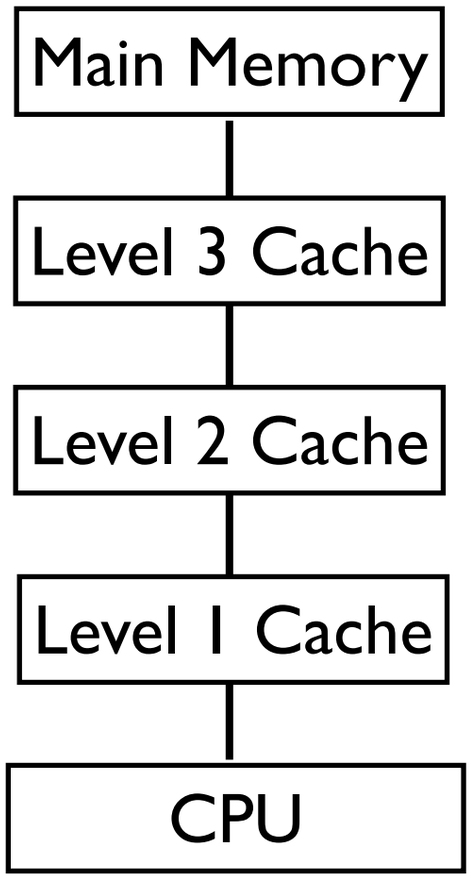
A Simple(?) Computer Architecture



For now, ignore the MPU and IRQ (we shall return to these)

CPU - central processing unit, executing instructions
MEM - fast memory, holding program and (live) data
DEV - hardware that interfaces to
(1) slow bulk storage devices (Hard Disks, etc
(2) outside world via I/O devices

Single Processor and Memory



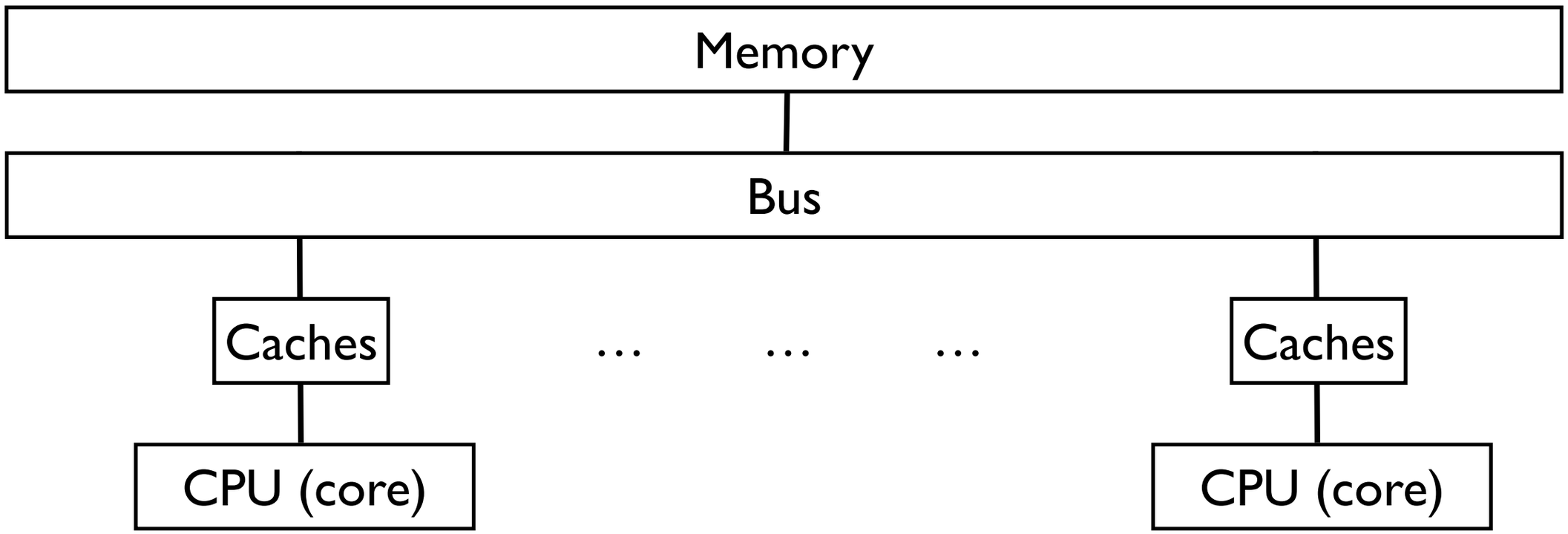
- Random Access Memory (RAM)
  - fairly slow, fairly cheap

- Hierarchy of caches between CPU and RAM.
  - Level 1 - smallest, fastest
  - ...
  - Level 3 largest, slowest.

- Register memory in the CPU:
  - very fast, very expensive
    - typically 400x faster than RAM !

(Part of) the so-called "Memory Hierarchy"

Multicore and Memory



Often the Level 3 cache is shared between all the CPUs

Shared Memory Machine

- Each processor has equal access to each main memory location – Uniform Memory Access (UMA).
  - Opposite: Non-Uniform Memory Access (NUMA)
- Each processor may have its own cache, or may share caches with others.
  - May have problems with cache issues, e.g. consistency, line-sharing, etc.

Sequential Program

- A ***Sequential Program*** has one site of program execution. At any time, there is only one site in the program where execution is under way.
- The ***Context*** of a sequential program is the set of all variables, processor registers (including hidden ones that can affect the program), stack values and, of course, the code, assumed to be fixed, that are current for the single site of execution.
- The combination of the context and the flow of program execution is often called [informally] a *thread*.

Concurrent Program

- A ***Concurrent*** *Program* has more than one site of execution. That is, if you took a snapshot of a concurrent program, you could understand it by examining the state of execution in a number of different sites in the program.
- Each site of execution has its own context—registers, stack values, etc.
- Thus, a concurrent program has *multiple threads* of execution.

Parallel Program

- A ***Parallel*** Program, like a concurrent program, has multiple threads of program execution.
- A key difference between *concurrent* and *parallel* is
  - In concurrent programs, only one execution agent is assumed to be active. Thus, while there are many execution sites, only one will be active at any time.
  - In parallel programs, multiple execution agents are assumed to be active simultaneously, so many execution sites will be active at any time.
- Another key difference:
  - Concurrency is a way of designing solution to problems by considering them to be composed of separate threads of execution that run independently, but are also able to interact when needed.
  - Parallelism is a way to speed up program execution when multiple computing cores are available by breaking a computation into pieces, each of which runs on its own core.

Interaction

- Most of the issues to do with threads (really, with any kind of concurrent/parallel processing) arise over unforeseen interaction sequences.
  - For example, if two threads attempt to increment the same variable

Unsafe Interaction Example

Thread 1	Thread 2	G
L=G (4)	–	4
L=L+10 (14)	–	4
	L=G (4)	4
G=L	L++ (5)	14
	G=L	5

G = Global Variable; L = Separate Local Variables

Interactions & Dependencies

- Interactions occur because of *dependencies*.
- Interactions can be made safe using a variety of techniques.
  - E.g. mutexes, condition variables, etc.
    - We have already had a look at them…
    - They tend to be expensive; sometimes *very* expensive.
- But, sometimes we can redesign the code to avoid dependencies.
- One of the biggest themes in this kind of programming is *dependency minimisation*.

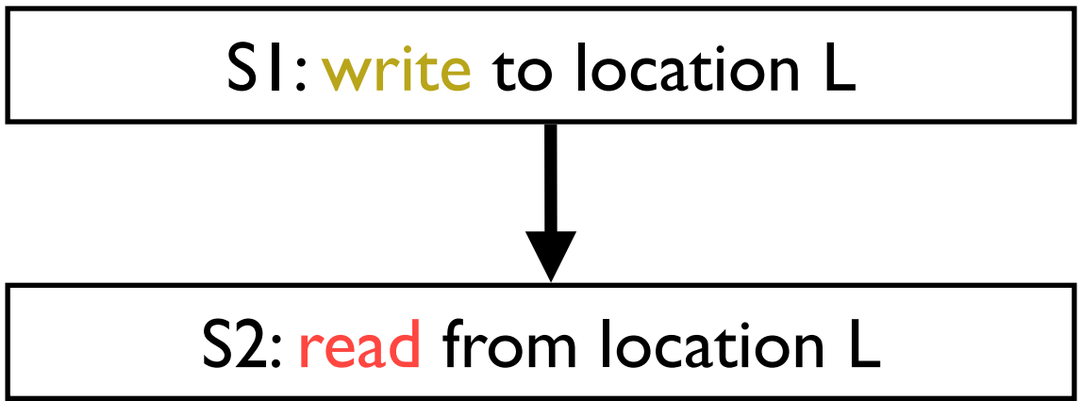
Why is it this difficult?

- It's not clear whether it *really* is more difficult to think about using multiple agents, e.g. multiple processors, to solve a problem:
  - Maybe we are conditioned into thinking about just one agent;
  - Maybe it's natural;
  - Maybe it's our notations and toolsets;
- Of course, maybe it really is trickier.
  - One possible culprit - often the simplest model we have of concurrent program behaviour is to assume that decision to switch between threads is purely non-deterministic (arbitrary, random).
    - This makes it hard to predict when things will happen.

Dependency

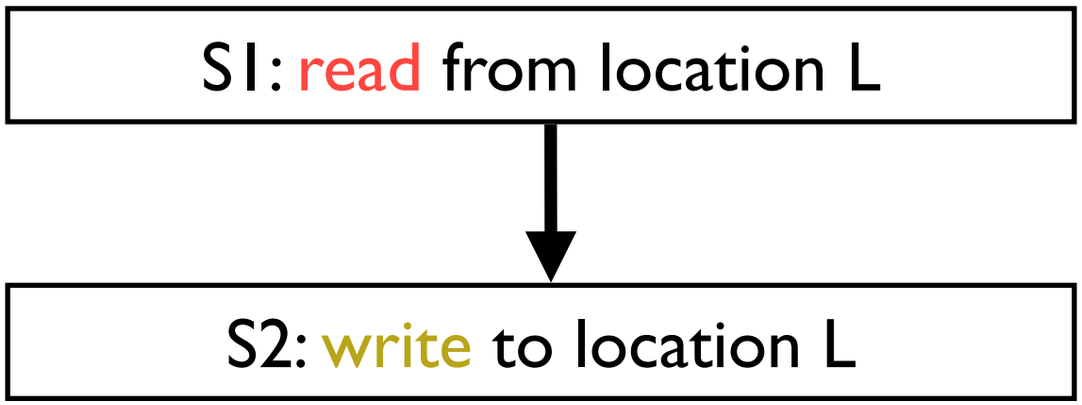
- Independent sections of code can run independently.
- We can analyse code for dependencies.
  - To preserve the meaning of the program;
  - To transform the program to reduce dependencies and improve parallelism.
- We typically define dependencies of various kinds between two program statements, (S1 & S2), that can be in the same thread, or in different ones.

1 – Flow Dependence



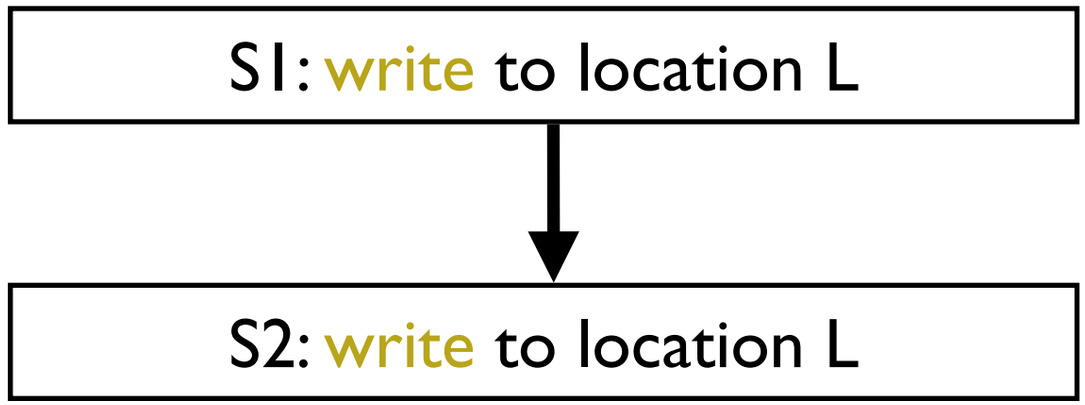
- Flow Dependence: S2 is flow dependent on S1 because S2 reads a location S1 writes to.
  - It must be written to (S1) before it's read from (S2)

2 – Antidependence



- Antidependence: S2 is antidependent on S1 because S2 writes to a location S1 reads from.
  - It must be read from (S1) before it can be written to (S2)

3 – Output Dependence



- Output dependence: S2 is output dependent on S1 because S2 writes to a location S1 writes to.
  - The value of L is affected by the order of S1 and S2.

Example