Computer Architecture I

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Modern computers are much more complicated than the LC-3.

B. Outcomes

After this lecture, you should understand

- The basic ideas behind memory caching, instruction pre-fetching, and instruction pipelining.
- The difference between multiprogramming/multitasking and multiprocessing.

C. Improving the Implementation of the ISA (Instruction Set Architecture)

- Improving primitive computers: Historically people worked to improve computers by (1) Augmenting their memory/peripherals/instruction set and (2) Making them run faster. To make them computers run faster, we can use faster transistors and clocks and we can try modifying the instruction cycle to get more instructions done.

D. Caching

- A **memory cache** is a small, fast memory that stores copies of values in main memory. When accessing memory, we check the cache first. If the cache contains often-used data, then our effective time to access memory decreases.
  - We may use different caches to hold data vs instructions because they can differ in pattern of uses.
  - It's not always the case that larger caches are better; the extra expense may not bring enough benefits.
  - People have found that multiple levels of cache can help: If a search of the very fast/small level 1 cache fails, we go to the somewhat slower but less

expensive level 2 cache before checking main memory. (We even see level 3 caches in some implementations.)

- Write-back policy: Writing a new value to memory actually changes the value in the cache; when should we actually update main memory? In a write-through cache, we update main memory every time a new value is stored; this keeps memory synchronized with the cache but is slower than in a write-back cache, where we only write the value to main memory when it's evicted from the cache (presumably to make room for some other piece of data).
  - Keeping the cache synchronized with memory ("**cache coherence**") is made harder by having I/O devices or other processes or CPUs that can update memory independently from the CPU.
- An **Instruction Cache** holds recently-used instructions. If the entire body of a frequently-executed loop fits into the cache, we can get excellent speedup; this is why we prefer programs with short loops over ones with long loops.
  - We can try to **pre-fetch instructions** (read the next instruction(s) into the cache from memory before we need them).
  - Modern CPUs have sophisticated algorithms for trying to predict which instructions might be used. Branch prediction: Which side of a conditional branch might we take? Keep info on what was taken in the recent past? Fetch both sides of the branch?
  - Once the instruction cache gets full, we need **eviction policies** to tell us which instructions to remove when we need room in the cache. E.g., **Least-Recently Used**

E. Instruction Pipelining

- **Instruction Pipelining** is where we overlap the processing of multiple instructions. We speed up decoding and increase the number of instruction executions per second by fetching instruction 5 while decoding instruction 4 while.... writing result for instruction 1.
  - Compare with memory caches, which speed up the Fetch instruction, Read Operands, and Write Results part of the instruction cycle.
- We may be able to complete more than one instruction per clock cycle.

  - A computer is superscalar, scalar, or subscalar depending on whether it can complete at most > 1, $= 1$, or < 1 instruction per clock cycle.
- **Latency vs Throughput**: Latency is the time to execute one instruction; throughput is the number of instructions completed per unit time. **Instruction pipelining aims to increase throughput, not decrease latency**.
  - One analogy uses **everyday plumbing**: Latency is the time it takes for water to come out of the faucet when you turn it on; throughput is the rate at which water comes out once it does start coming out. (Latency depends on the length of the pipe; throughput depends on the diameter.)
  - Another analogy uses a **bucket brigade**: The only person tossing water on the fire is the final person, but that person gets to toss more water on per unit time because other people are moving buckets through the line.
- A **pipeline stall** is when one part of an instruction pipeline has to wait for some other part. One cause is a data dependency (one instruction needs the result of another). To keep the pipeline full, modern computers use out-of-order execution: The control unit rearranges the order of instructions to reduce data dependency delays. When you toss conditional branches into the mix, you can find yourself speculatively executing an instruction (before you know you need it) but then having to un-execute it if your branch prediction was wrong.

F. Multiprogramming/Multitasking vs Multiprocessing

- In **multiprogramming (a.k.a. multitasking)**, the OS system runs multiple programs in parallel (on one CPU) by switching from one program to another. We execute programs **concurrently** (it looks like they're running in parallel, but they're actually each running individually for a short quantum of time).
  - By mixing the execution of processes that want the CPU vs processes that want I/O devices, we work toward having processes do less waiting (thereby decreasing latency) and keeping the CPU and I/O devices busier (thereby increasing throughput).
- In **multiprocessing**, we have multiple CPUs (either on the same chip or different chips), so we can perform operations truly in parallel. Each processor has its own control unit and arithmetic-logic units; they share memory (though each processor might have a bit of local memory too).

  - In **Asymmetric multiprocessing**, the processors are different. Often this is done so that specific processors can be specially optimized for a particular kind of work. (A CPU with a video processor on a graphics card uses asymmetric multiprocessing.)
  - In **Symmetric multiprocessing** (SMP), we have identical processors sharing memory. This makes it easy to move programs from one processor to another, so it's easier to balance loads across processors. A **multi-core** computer has multiple processors on the same chip. Usually they are identical processors, so we have symmetric multiprocessing.
  - When processors have individual caches there's another instance of the cache coherence problem (keeping them synchronized across processors); if processors share a cache, we have to be careful to arbitrate simultaneous requests.
- **Race conditions**: Whether we have multiple processors or multiple tasks running on one processor, if programs share a resource (typically variables in memory), we have to ensure that the overall computation is correct regardless of what order programs run in or how fast their processors are (if there are more than one of them). Otherwise we have a **race condition**: The correctness of our computation depends on the relative speed of execution of different parts of the computation. Race conditions and synchronization make it hard to write correct programs that run concurrently or in parallel.

Processes and Threads

- A **process** is a program running with its data[1]. An analogy: A program is like a script for a play — it's a set of instructions. **A process is like a performance of a play** — you have resources like actors, props, and a stage allocated, and the actors are doing things: Saying lines, interacting with props, etc.
- A **thread of execution** is an execution path through a program (along with the data for the thread). Going back to the play analogy, a thread is like an individual actor. A play performance involves the coordinated actions of all

[1] It's annoying, but running multiple processes on one CPU is **multiprogramming**. Having multiple processors is **multiprocessing**.

the actors. Execution of a process can involve coordinated action between threads.

- Contemporary programs are often **multi-threaded** (involve multiple threads). You might have one thread listening to the keyboard, another listening to the trackpad, another doing computation, and another displaying things to the monitor.) In the same way, a play typically has multiple actors, though you can have one-performer plays where one actor speaks, say, Ebenezer Scrooge's lines and then changes to Bob Cratchit's lines, etc.
- Multithreading is used partly because it can make the program **easier to write and understand**. It's also used because **changing threads within a process is cheaper than changing processes**. Threads share the same context (the stage and props), but processes don't. To change processes, you have to clear the stage and get everyone involved in the other play onto the stage.
- Running multiple threads: One possibility is to run different threads simultaneously on different cores or processors. (In the analogy, it's like having multiple actors saying lines simultaneously.) Another is to run multiple threads concurrently on one core. We can run each thread for a slice of time and switch (similar to multiprogramming). In Intel's hyperthreading approach, one core can simulate two cores by having two threads share the same instruction pipeline. Benchmarks indicate perhaps 20% increase in average computation speed.

Computer Architecture I

CS 350: Computer Organization & Assembler Language Programming

A. Why?

- Modern computers are much more complicated than the LC-3.

B. Outcomes

After this activity, you should be able to describe the basics of

- Caching, pipelining, multiprogramming, multiprocessing, race conditions, and processes and threads.

C. Questions

1. Are larger memory caches always better than smaller ones?
2. How do short loops in programs and instruction caches relate?
3. What is instruction pipelining? Does it decrease latency? Throughput?
4. What is the difference between multiprogramming and multiprocessing?
5. Do multicore computers use symmetric or asymmetric multiprocessing? What about a video card vs our general-purpose CPU?
6. What's a race condition?
7. How are programs and processes different?
8. What is a thread of execution?
9. Why do people write multi-threaded programs?

Solution

1. Surprisingly, no. There can be anomalous situations where enlarging a cache causes a slowdown.
2. Smaller loops have a better chance of having all their executable code fit inside the instruction cache, which speeds up execution.
3. In instruction pipelining we overlap the processing of multiple instructions. This increases throughput but not latency.
4. In multiprogramming, the OS system runs multiple programs concurrently, by quickly switching from one program to another. In multiprocessing, we have multiple CPUs, so both programs execute in true parallel fashion.
5. If the multiple cores are identical, we have symmetric multiprocessing. Video cards and general-purpose CPUs are asymmetric.
6. Say two programs execute concurrently or in parallel and share a resource. If the correctness of overall execution depends on the relative speed of execution of the programs, we have a race condition.
7. A program is a set of instructions; a process is the execution of that program (sitting in memory, with resources allocated, etc).
8. A thread of execution is an execution path through a program.
9. Multi-threaded programs can be easier to write and understand than monolithic single-threaded programs. If different threads execute on different CPUs or cores, then actual execution can be faster.

