Lecture 1 notes

Computer science at an inflection point:

- Billion-transistor processors possible but…
  - Power and thermal problems prevent us from just using 1 billion active transistors on a chip
- Instruction-level parallelism has driven processor design for many years but…
  - Techniques for extracting more ILP from programs show diminishing returns in performance
- Memory latency has been improving at a much slower rate than instruction execution latency
  - This trend has not changed in many years…
- Computer system designers are turning their interest to multi-processors, or more specifically, processors with multiple cores
  - This has been attempted before in the 80's but not with great success. Uni-processor performance improved rapidly (2X every 1.5 years), while multi-processor designers faced serious challenges while attempting to scale their systems to achieve high performance at low cost
  - Now things may be different: Multi-processing appears to be the only viable alternative to achieve higher performance in next-generation computer systems
  - Multi-processing is also now embraced by all major microprocessor vendors
- Why is this an inflection point?
  - In the 80's and 90's microprocessor performance improved at an exponential rate (2X per 1.5 years) thanks to both technology and architectural innovation and techniques for extracting more instruction-level parallelism out of programs (both hardware-based and software-based, e.g. compilers)
    - Clock speed was very strongly correlated with processor performance
    - Instruction sets remain stable (almost unchanged) across processor generations
    - Improvements in technology and architecture were available at "no cost" for software developers, e.g. you would buy a processor every other year and you would have a guarantee that the processor will be almost twice as fast as the previous one you had
  - With multi-processors, transparent performance scaling is no longer possible
    - Multi-processors can improve performance only if software can be "parallelized" so that the workload is "distributed" between multiple processors

- Hardware and compilers can not parallelize software automatically (at least not yet)
- Parallelizing software is tedious, even for the best programmers (e.g. a PhD student at a Top Computer Science Program)
- Parallelization is a hard process: Changes required in algorithms, programming languages, compilers, operating systems, and runtime libraries in order to get a program (eg. Sort) to run efficiently on a multi-processor

The role of instruction sets:

- Instruction sets are the interface between high-level programming languages and processors.
- Instruction sets provide convenient "abstractions" of the low-level functions implemented in the hardware
- Instruction sets (like any abstraction) need to be both convenient and amenable to an efficient implementation
- These days the arguments regarding instruction sets are more or less settled: most processors implement a small (reduced) instruction set, including three fundamental classes of simple instructions: arithmetic/logical, memory access (load/store) and control (branches, jumps). Instructions are of fixed length, have on of few (e.g. 2-3) formats, offer a few ways to address memory (e.g. based and offset, direct,…) and have specific semantics for detecting and handling exceptions
- In the old days of CS, "instruction set" was synonyms to computer architecture. These days instruction sets have converged to a common paradigm (RISC), and computer architecture carries the responsibility of implementing this paradigm more efficiently, across processor generations. Computer architecture evolves a lot faster than instruction sets.
- Computer architecture is now an end-to-end design process. Architects are concerned about the functionality of the entire system (processor, memory, disks, network, other external devices…), and customize their designs to the needs of applications

Computer architecture is a creative process

- Architects combine intuition with rigorous experimental work. Experimental computer architecture involves:
    - Searching the design space of a computer system (components, parameters, capacities), to find the optimal design (or more typically, a good compromise that serves the target application domain of the computer system well)
    - Running and characterizing workloads (applications), in order to understand where the cycles go, what are the bottlenecks in the

system and how the system can be modified to execute the same task in less time

Basic principles of modern computer architecture:

- Parallelism
- Locality
- Making the common case fast
- Amdahl's law, focusing optimizations on the major bottlenecks (the big challenges)
- Processor performance equation: seconds/program = instructions/program X cycles/instruction X seconds/cycle

Parallelism in computer architecture stems from:

- Multiple components (e.g. multiple cores on a single processor)
- Pipelining of activities
- Both have limits
- Pipelining is limited by: structural hazards (instructions requiring the same resource during the same cycle), data hazards (instructions dependent on data produced by earlier instructions and not available), and control hazards (instructions the execution of which is dependent on earlier control instructions that have not yet completed their execution)

Locality in computer architecture stems from program behavior:

- Spatial locality: Programs access words in adjacent memory locations, or, more formally, a program accesses a memory location X, shortly after it accesses a memory location X-d (d = small, architecture-dependent)
- Temporal locality: A program accesses a word in memory location X many times during a short time interval

Processor performance equation:

- #instructions in a program can be reduced by: algorithms, compilers (via optimizations, such as dead code elimination), and architecture (e.g. by using a string instruction that copies bytes from string a to string b, as opposed to individual loads and stores)
- cycles per instruction can be reduced by the instruction set (e.g. by using very simple instructions), by the architecture (e.g. by using pipelining of instruction execution), and by the compiler (e.g. by scheduling instructions to reduce delays between them)
- clock cycle can be reduced by the architecture (e.g. by using very simple instructions that can be implemented fast in hardware), and by technology (e.g. speed vs. power constraints)

Making the common case fast:

- What components of the system are used more frequently than others?

- What instructions are executed more frequently than others?
- What components of the system have the highest failure rate?

Amdahl's law:

- Given an optimization (e.g. improved system design), speedup is always bound by 1/(1-f), where f is the fraction of the program affected (accelerated) by the optimization