Chapter 1

Fundamentals of Quantitative Design and Analysis

Acknowledgement: Slides are adapted from Morgan Kaufmann companion material

The Computer Revolution

- Progress in computer technology
  - Underpinned by Moore’s Law
- Makes novel applications feasible
  - Computers in automobiles
  - Cell phones
  - Human genome project
  - World Wide Web
  - Search Engines
- Computers are pervasive

Classes of Computers

- Internet of Things/Embedded computers
  - Hidden as components of systems
  - Stringent power/performance/cost constraints
- Personal Mobile Devices (PMDs)
  - Wireless devices with multimedia user interfaces such as cell phones, tablet computers, etc.
  - Stringent power/performance/cost constraints
- Desktop computers
  - General purpose, variety of software
  - Subject to cost/performance tradeoff
- Server computers
  - Network based
  - High capacity, throughput, reliability
- Clusters/Warehouse-Scale Computers
  - Collections of desktop computers or servers connected by a network to form a larger computer
  - Price-performance oriented

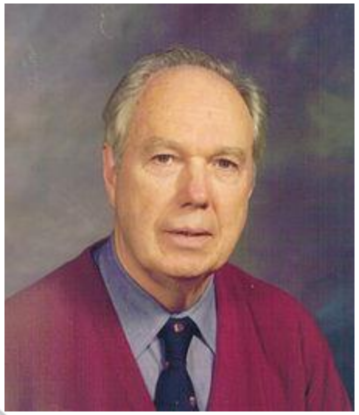
Improving Performance Through Exploiting Parallelism

- Exploiting parallelism is the main way of improving computer performance

- Types of parallelism
  - Data-level parallelism (DLP): Same operation, different data
  - Task-level parallelism (TLP): Divide work into tasks that can operate mostly in parallel

- Sources of parallelism in computers:
  - **Instruction-Level Parallelism (ILP):** Independent instructions can be executed in parallel
  - **Vector processing or Single Instruction Multiple Data (SIMD) parallelism:** A single instruction can be applied to multiple data elements at the same time.
  - **Thread-Level Parallelism (TLP):** Software can be organized as different threads that can operate in parallel (commonly threads communicate)
  - **Request-Level Parallelism:** In request-response server systems, software can process tasks in parallel (threads don't explicitly communicate but may access shared data structures)
  - **Memory-Level Parallelism (MLP):** Processor memory requests (loads and stores) can be issued in parallel to hide memory latencies

Flynn’s Taxonomy



- Single instruction stream, single data stream (SISD)
- Single instruction stream, multiple data streams (SIMD)
  - Vector architectures
  - Multimedia extensions to instruction sets (e.g., SSE)
  - Graphics processor units (GPUs)
- Multiple instruction streams, single data stream (MISD)
  - No commercial implementation
- Multiple instruction streams, multiple data streams (MIMD)
  - Tightly-coupled MIMD: multi-cores, many-cores
  - Loosely-coupled MIMD: clusters, data centers

WHAT IS COMPUTER ARCHITECTURE?

Classical Computer Architecture Definition: The ISA

- The Instruction Set Architecture (ISA)
  - The actual, programmer visible instructions set
  - In other words: The language you can use to get the computer to do stuff
  - We will focus on RISC-V, alternatives are x86, ARM, etc.

- ISA components and choices (more on this later):
  - Operand types (typically registers or memory locations)
  - Memory addressing (typically byte-addressing)
  - Addressing modes (e.g., register, immediate, displacement)
  - Types and sizes of operands (word, char, double word, etc.)
  - Operations (data transfer, logical, control or floating point)
  - Control flow instructions (conditional branches, unconditional jumps, etc.)
  - Encoding (fixed vs. variable length)

RISC-V Instruction Formats

31 … 25	24 … 20	19 … 15	14 … 12	11 … 7	6 … 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm [11:0]		rs1	funct3	rd	opcode	I-type
imm [11:5]	rs2	rs1	funct3	imm [4:0]	opcode	S-type
imm [12] imm [10:5]	rs2	rs1	funct3	imm [4:1|11]	opcode	B-type
imm [31:12]				rd	opcode	U-type
imm [20|10:1|11|19:12]				rd	opcode	J-type

Genuine Computer Architecture Definition

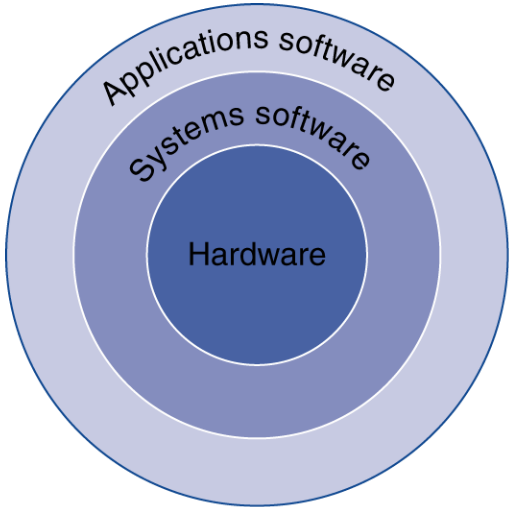
- Computer architecture objective: Design the *organization* and the *hardware* to meet objectives and functional requirements

- Computer organization:
  - High-level aspects of a computer's design
  - Examples: Memory system, interconnect, design of the CPU, etc.

- Computer hardware:
  - Detailed logic design and packaging of the computer

- Final definition: Computer architecture covers the ISA, the organization and the hardware.

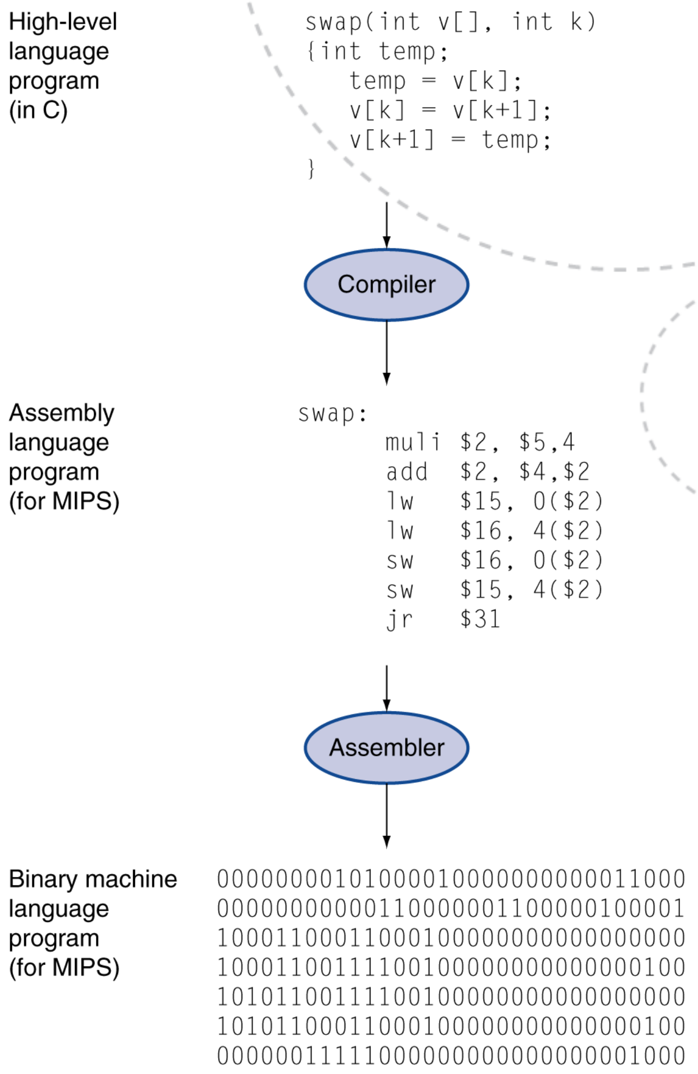
Below Your Program



- Application software
  - Written in High-Level Language (HLL)
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers

Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data


High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Inside the Processor (CPU)

- A processor capable of general computation needs to be able to do arithmetic, conditional branches and memory access

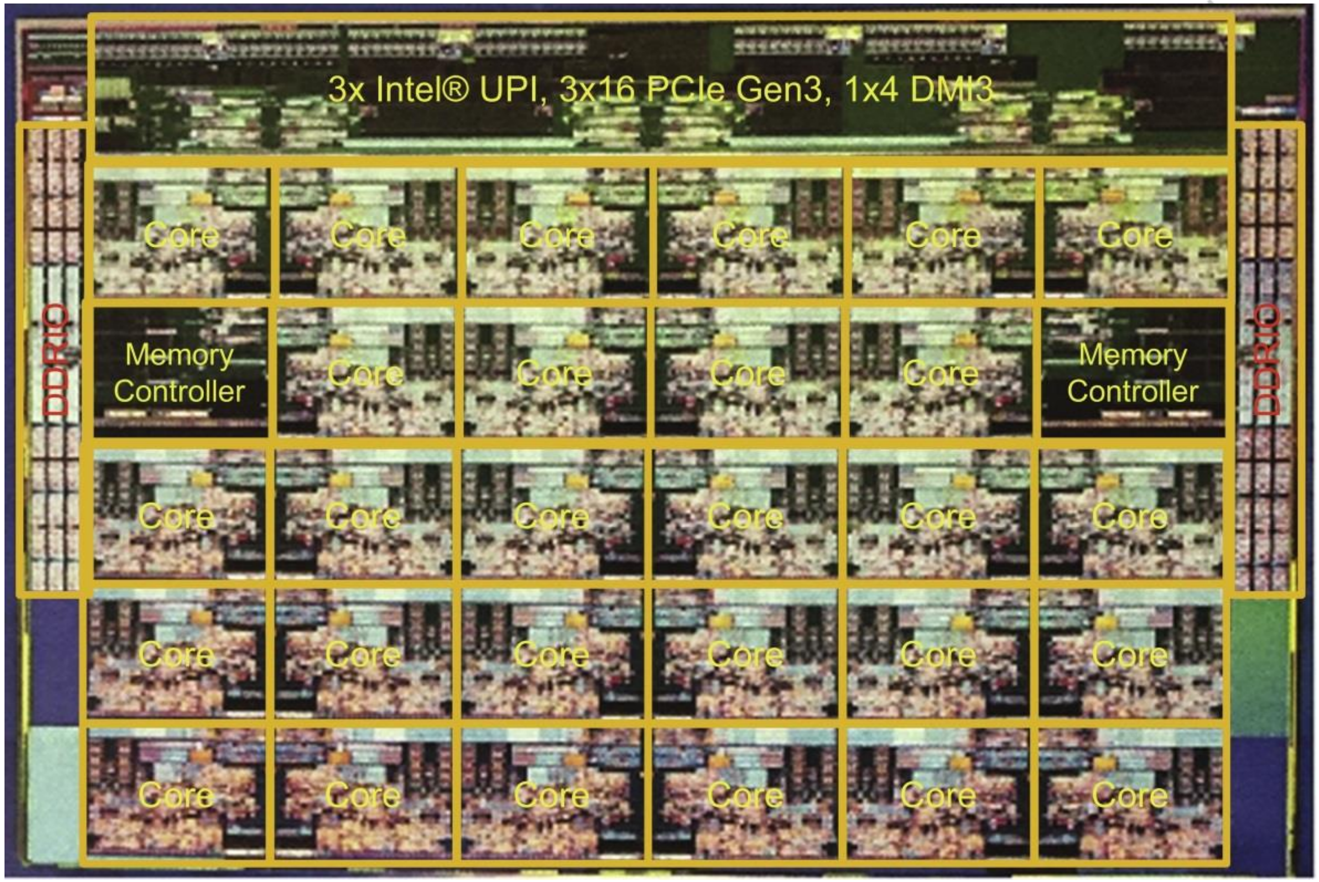
- Arithmetic
  - Datapath: performs operations on data (i.e. arithmetic)

- Conditional branch
  - Control: selects next instruction depending on branch outcome
  - Control: enables the parts of the datapath needed to execute a given instruction

- Memory access
  - Off-chip access is often slow
  - Solution: Cache (Small fast hardware-managed SRAM memory)

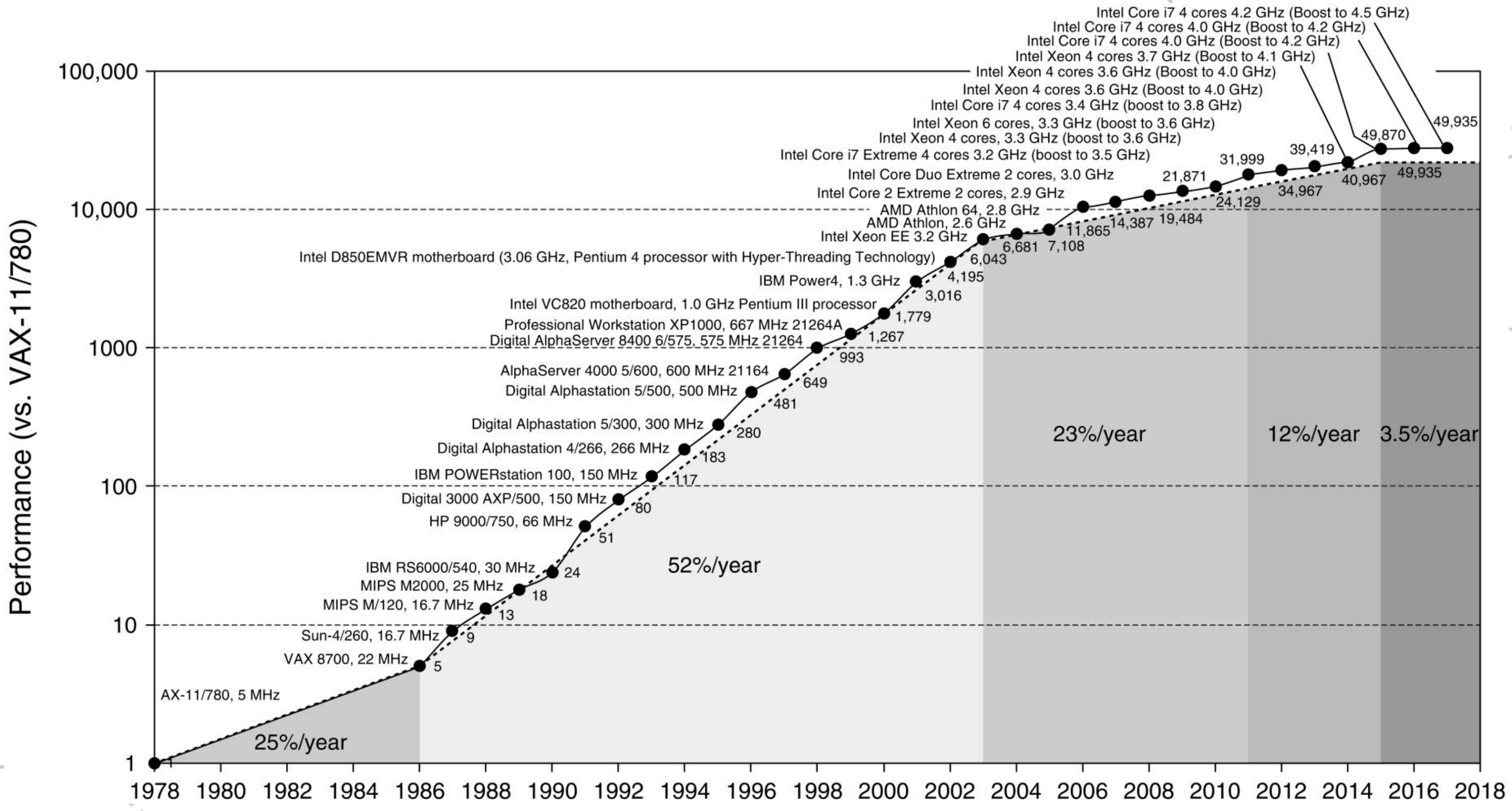
Multi-core Example

- Intel Skyelake Core i7 multi-core: 28 processor cores

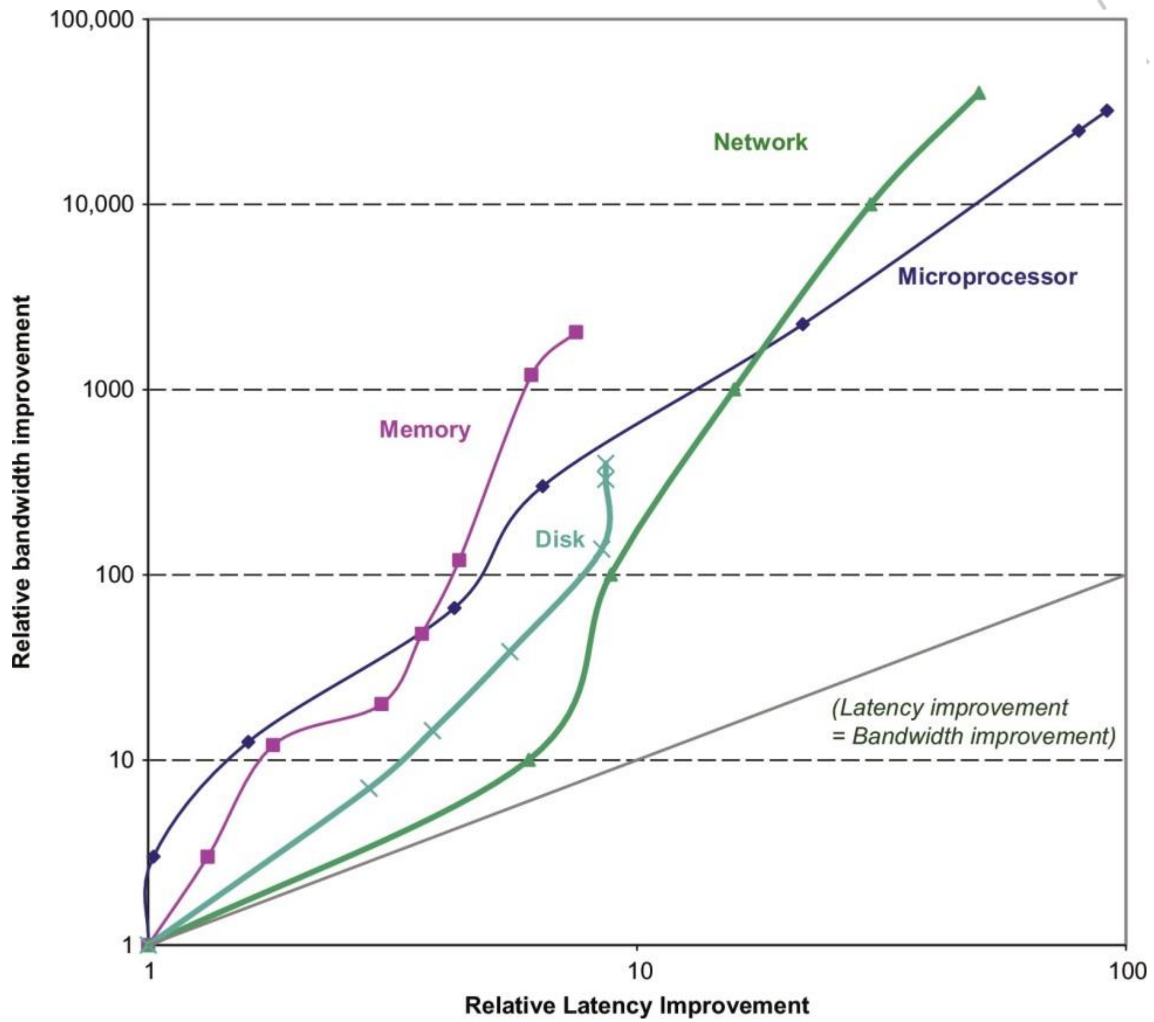


TECHNOLOGY TRENDS

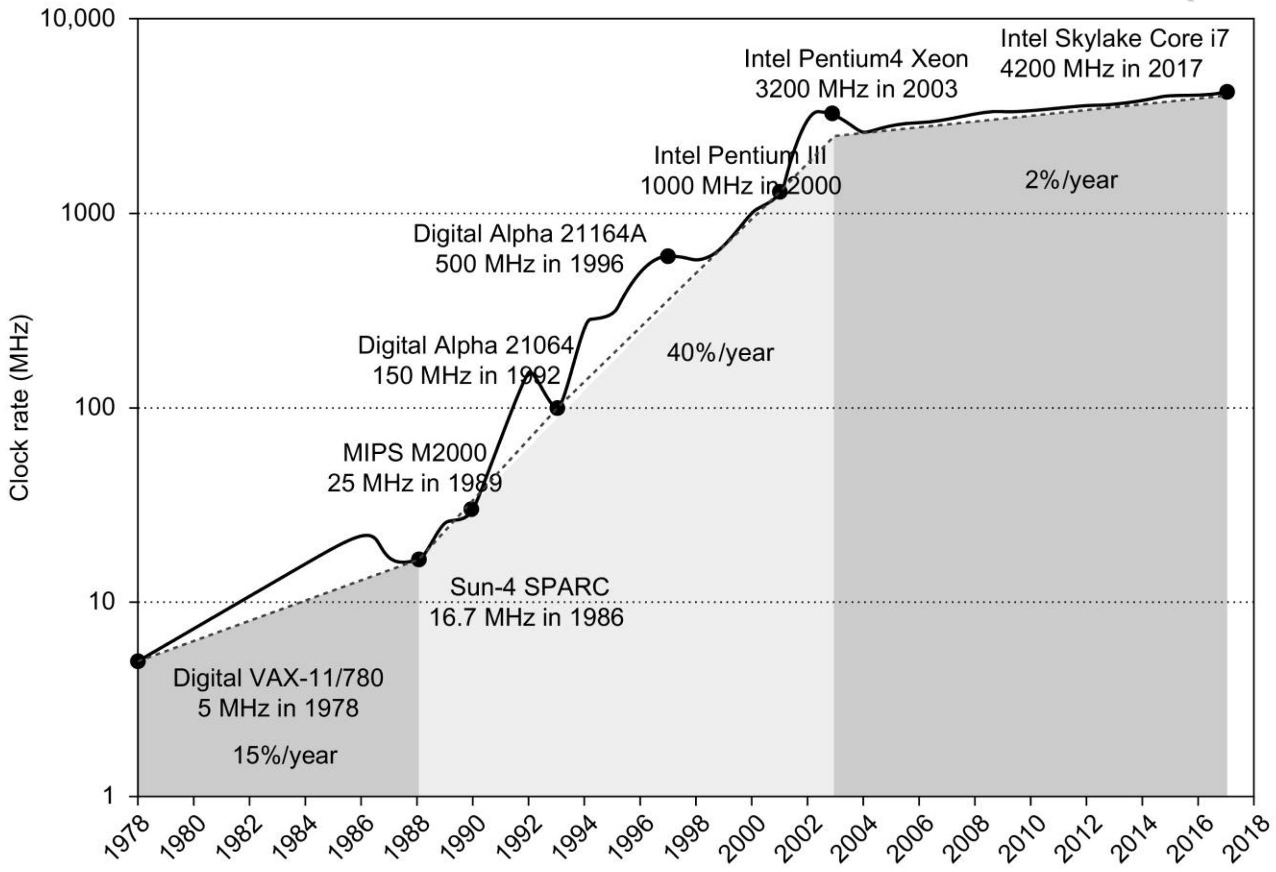
Single-Core Processor Performance Growth



Latency vs. Bandwidth



Clock Frequency



PRINCIPLES OF QUANTITATIVE DESIGN

Principles of Quantitative Design

- Take advantage of parallelism
  - Exploiting parallelism is the pervasive strategy for improving the performance of computers
- Leverage locality
  - Programs tend to reuse data and instructions – think of loops and arrays
  - This results in reuse over time (temporal locality) and in (memory) space (spatial locality)
- Focus on the common case
  - Computer architecture is full of trade-offs – favoring the frequent case over the infrequent case typically results in better performance
  - Acknowledge Amdahl's law: A massive improvement on an infrequent case gives negligible overall improvement

Pitfall: Amdahl's Law

- Improving one aspect of a computer and expecting a proportional improvement in overall performance

$$T_{\text{Improved}} = \frac{T_{\text{Affected}}}{\text{ImprovementFactor}} + T_{\text{Unaffected}}$$

- Example: Multiply accounts for 80s out of 100s total runtime
- How much improvement in multiply performance is necessary to get 5X overall speed-up (i.e., reduce runtime to 20s)?

$$\lim_{n \to \infty} f(n) = \frac{80}{\text{n}} + 20 = 20$$

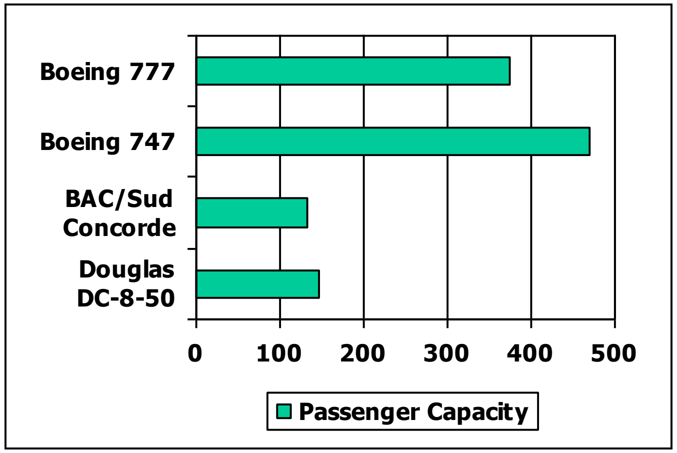
MEASURING PERFORMANCE

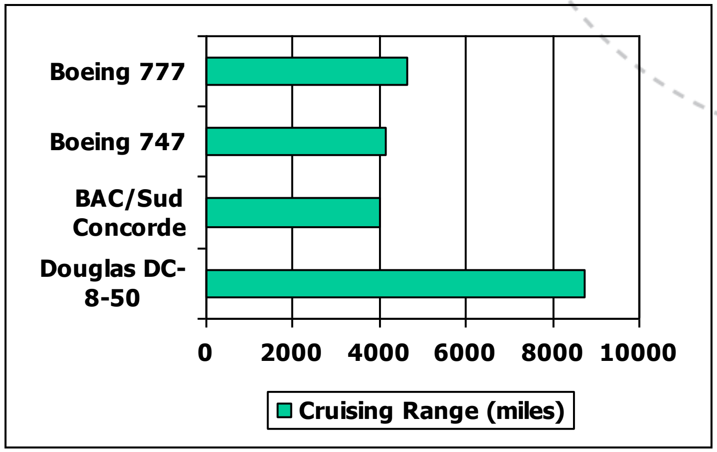
Understanding Performance

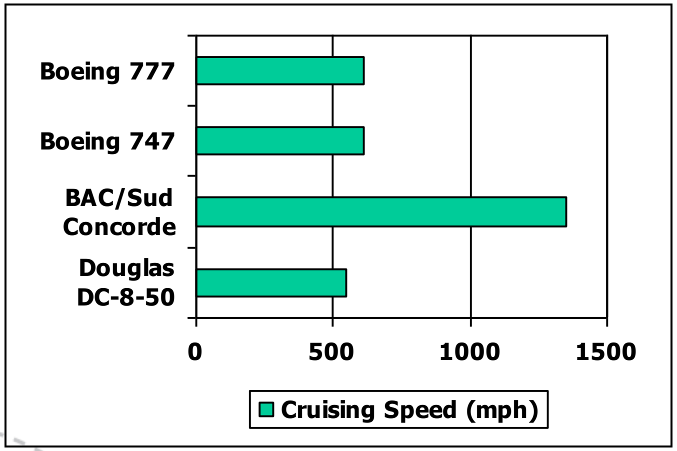
- Algorithm
  - Determines number of operations executed
- Programming language, compiler, and architecture
  - Determine number of machine instructions executed per operation
- Processor and memory system
  - Determine how fast instructions are executed
- I/O system (may include the Operating System (OS))
  - Determines how fast I/O operations are executed

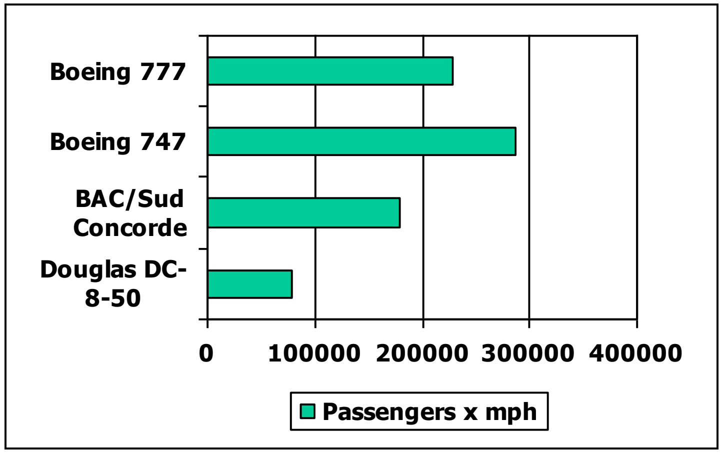
Defining Performance

- Which airplane has the best performance?









Response Time

- Time from issuing a command to its completion
  - This is often referred to as the *turn-around time*
- Another response time definition: Time from issue to first response
- *Execution time* is the time the processor is busy execution the program
  - Turn-around time includes the time the process waits to be executed, execution time does not
  - Also: user execution time vs. system execution time

Response Time and Throughput

- Throughput
  - Total work done per unit time
- How are response time and throughput affected by
  - Replacing the processor with a faster version?
  - Adding more processors?

Relative (Normalized) Performance

- Definition: Performance = 1/Execution Time
- “X is n times faster than Y”

$$\begin{aligned}&\text{Performance}_X/\text{Performance}_Y\\&=\text{Execution time}_Y/\text{Execution time}_X=n\end{aligned}$$

- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time$_B$ / Execution Time$_A$ = 15s / 10s = 1.5
  - So A is 1.5 times (or 50%) faster than B

Measuring Execution Time

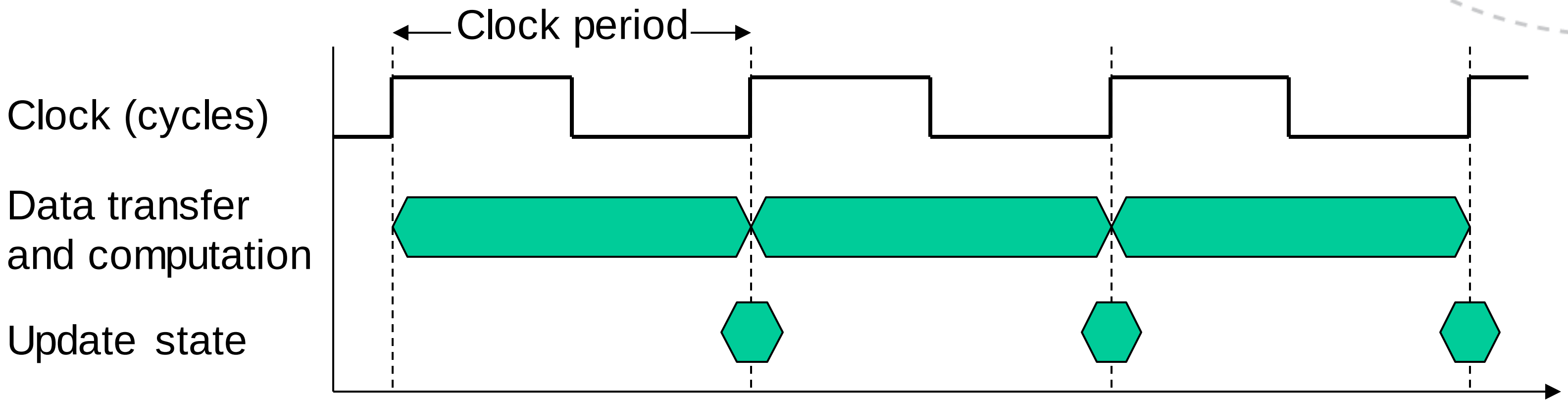
- Elapsed time/Wall clock time
  - Total turn-around time, including all aspects
    - Processing, I/O, OS overhead, idle time
  - Determines system performance
- CPU time
  - Time spent processing a given job
    - Discounts I/O time, other jobs' shares
  - Comprised of **user CPU time** and **system CPU time**
  - Different programs are affected differently by CPU and system performance
- Which programs to use?
  - Choose a representative set of real applications (i.e., benchmarks)

Benchmarks

- Must use real applications
  - Programs tend to consist of phases
  - The relative importance of each phase creates bias – different computer architecture techniques pay off in different phases
  - Recreating this bias synthetically is very difficult
- Must use a collection of real applications
  - A computer is general-purpose – it must perform decently across as many applications as possible
- Benchmark suites:
  - SPEC CPU, SPEC OMP, PARSEC, NAS, TPC, etc.
  - Each suite attempts to be representative for the workloads of a chosen domain
- Discredited benchmarking approaches:
  - Kernels, toy programs and synthetic benchmark
  - Do not accurately recreate the behavior of a collection of real programs
  - Another problem: It is too easy to cheat if the benchmarks are simple.

CPU Clocking

- Operation of digital hardware governed by a constant-rate clock



- Clock period: duration of a clock cycle
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- Clock frequency (rate): cycles per second
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$
$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate

- Hardware designer must often trade off clock rate against cycle count

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time on benchmark X
- Designing Computer B
  - Design Goal: 6s CPU time on benchmark X
  - Faster clock causes 20% increase in the number of clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program
  - Determined by program, ISA and compiler

- Average Cycles Per Instruction (CPI)
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix
  - Sometimes we report the inverse of CPI – called Instructions Per Cycle (IPC) since this is a higher-is-better metric

- Under which conditions is it valid to compare CPIs directly?

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA, program and compiler
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps} \leftarrow$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2 \leftarrow$$

…by this much

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
  - Clock Cycles = 2×1 + 1×2 + 2×3 = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock Cycles = 4×1 + 1×2 + 1×3 = 9
  - Avg. CPI = 9/6 = 1.5

The Iron Law

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, Clock cycle time

SUMMARY

Summary

- CPUs are pervasive and ubiquitous
  - Fully understanding how they are constructed is a good preparation for a career in ICT
- Performance of single-core processors have saturated
  - Future processors are likely to be heterogeneous
  - Understanding the basics of computer design is very helpful when trying to use these systems efficiently
- Key metric: Performance
  - Normally traded off against chip area, energy, power, etc.
  - Measured on a collection of programs (benchmarks) that are (hopefully) representative of a given domain.