Computer Abstractions and Technology

CSI3102-02: Architecture of Computers
(컴퓨터아키텍쳐)

Youngsok Kim (김영석)

DEPARTMENT OF
COMPUTER SCIENCE
YONSEI UNIVERSITY

Computer Architecture

- A set of rules and methods which describe the functionality, organization, and implementation of **computer systems**
- In other words, **computer architects** design the internals of computer systems!

- Computer systems
  - A complete **computer** including the hardware, the operating system, and peripheral equipment required for performing full operations

Computers

- Machines which can be instructed to **execute series of arithmetic or logic operations automatically** via **computer programming**

- Via computer programming
  - Perl/Python/Ruby, C/C++, Java, assembly, ...
- Execute arithmetic or logic operations
  - Instructions generated by compilers/interpreters
- Execute the operations **automatically**
  - This is what you will learn throughout this semester!

Traditional Classes of Computers

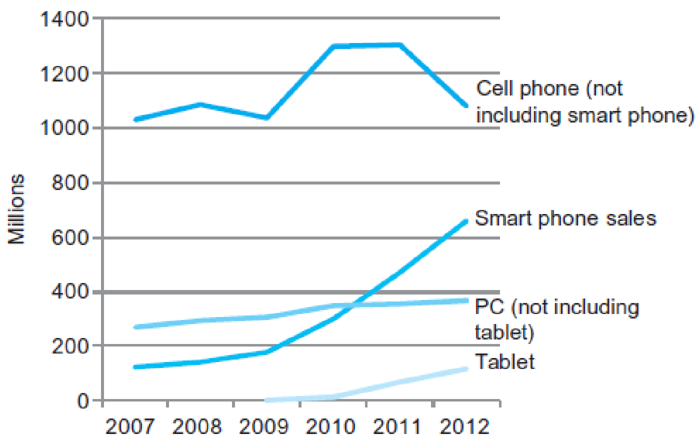
- Personal Computers (PCs)
  - Good performance to single users at low cost
- Servers
  - Greater computing, storage, and input/output capacity
- Supercomputers
  - The high-end extreme of servers
- Embedded Computers
  - Run one application or one set of related applications typically integrated with the hardware

Q: Why split computers to the classes in the first place?

A: Because the performance/cost requirements differ!

The Post-PC Era

- **Personal Mobile Devices (PMDs)** instead of PCs
  - Operated with a battery and wireless connections
  - Likely to rely on a touch-sensitive screen or speech input



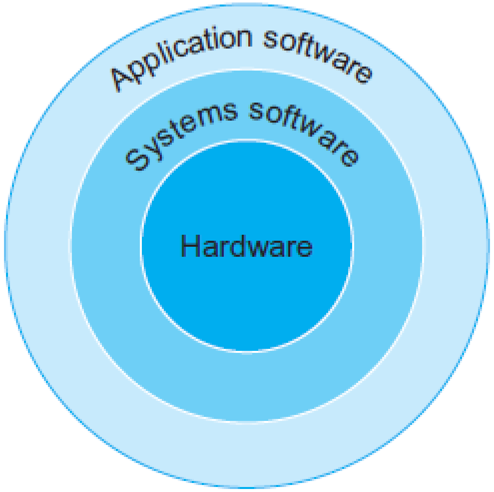
The manufactured # of PMDs (tablets, smart phones) now exceed PCs and traditional cell phones, reflecting the post-PC era.

- **Cloud Computing** instead of servers
  - a.k.a. Warehouse Scale Computers (WSCs)
  - Provide Software-as-a-Service (SaaS) via the cloud
    - e.g., Google App Engine, Amazon Web Services, Microsoft Azure

Below Your Program

Abstracting a Computer System

- **Application software**
  - High-level software running on top of the systems software
  - e.g., web browsers, games
- **Systems software**
  - Low-level software for executing the application software on the hardware
  - e.g., operating systems, compilers
- **Hardware**
  - The underlying hardware that executes operations from the systems software
  - e.g., CPUs, GPUs



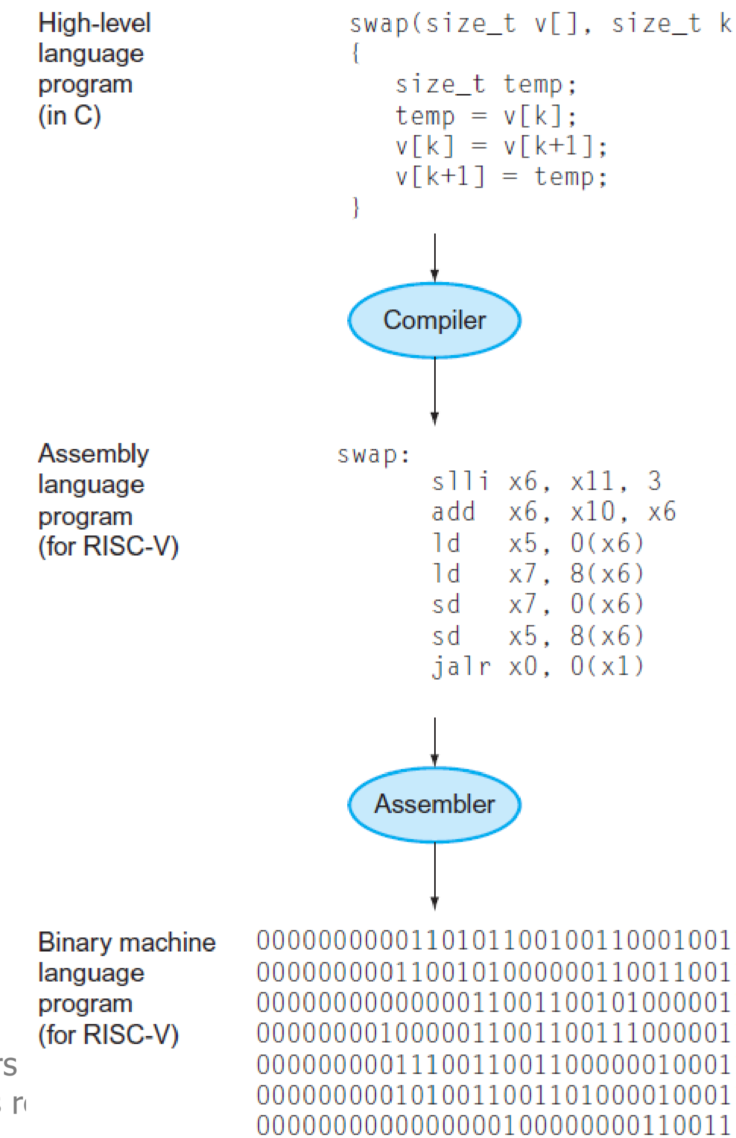
Operating Systems & Compilers

- Two types of systems software tightly coupled to the underlying hardware

- **Operating Systems (OSes)**
  - Interfaces between user applications and the hardware
  - Handle basic I/O operations, allocate memory/storage, ...
  - e.g., Windows, Linux, OS X
- **Compilers**
  - Translates a program written in a high-level language into instructions which the underlying hardware can execute
  - e.g., gcc/g++, LLVM

Compiling a User Program

- **Assembly** language
  - Utilizes symbolic notations to abstract the machine language
  - e.g., `add A, B` instead of `1001010100101110`
- **Binary machine** language
  - Only consists of bits (`0`s & `1`s)
  - **THE** language electronic hardware can understand



DEPARTMENT OF COMPUTER SCIENCE
YONSEI UNIVERSITY

Under the Covers & Building Processors and Memory

Components of a Computer

- **Datapath** & **Control** (the Processor)
  - Get instructions and data from memory, and execute the instructions using the data
- **Memory**
  - Serves as an intermediate storage for instructions and data between the processor and the I/O
  - Organized in a hierarchical fashion
    - e.g., two-level memory hierarchy with SRAM and DRAM
- **Input** & **Output** (I/O)
  - Write data to memory & read data from memory
  - e.g., keyboards, mouse, monitors, speakers

Inside Apple iPad 2

- Contains all the 5 components of a computer
  - Processor: Apple A5
  - Memory: 512 MiB
  - I/O: touchscreen, LCD, WiFi, ...

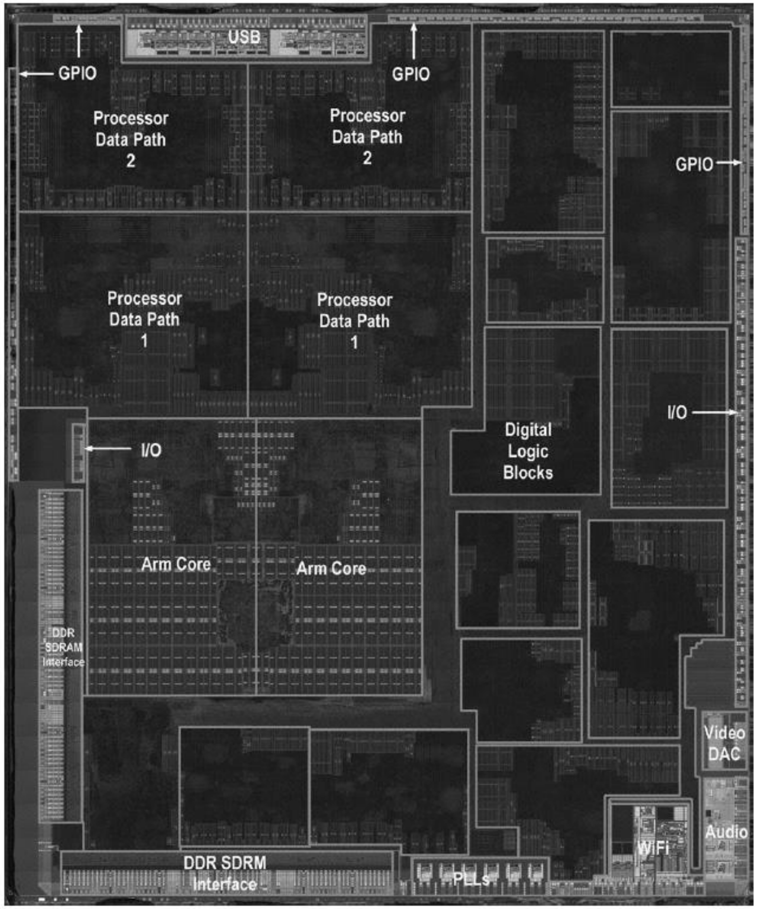
- Integrated Circuits (Chips)
  - 2x 1-GHz ARM processors
  - 2x 2-Gib DRAM chips

Components of Apple iPad 2

The logic board of Apple iPad 2

Apple iPad 2's Microprocessor

- Datapath
  - Perform arithmetic operations
  - e.g., two ARM processors, a PowerVR graphics processing unit
- Control
  - Instructs the datapath, memory, and I/O devices
  - e.g., interfaces to DRAM, I/O, WiFi, Audio



DEPARTMENT OF COMPUTER SCIENCE
YONSEI UNIVERSITY

The Hardware/Software Interface

- **Instruction Set Architectures** (ISAs)
  - The interface btw. the H/W and the lowest-level S/W
  - Include anything programmers need to know to make a binary machine language program work correctly
    - e.g., instructions, I/O devices

- **Application Binary Interfaces** (ABIs)
  - The basic instruction set & the OS interface provided for application programmers
  - Encapsulate the details of I/O, memory allocation, etc. (by the underlying OS)

How Do We Store Data?

- **Volatile** memory
  - Forget the data when it loses power
  - Used for holding data and programs inside the computer
    - e.g., Dynamic Random Access Memory (DRAM) chips
  - Fast, but small and expensive
  - a.k.a. **main/primary** memory

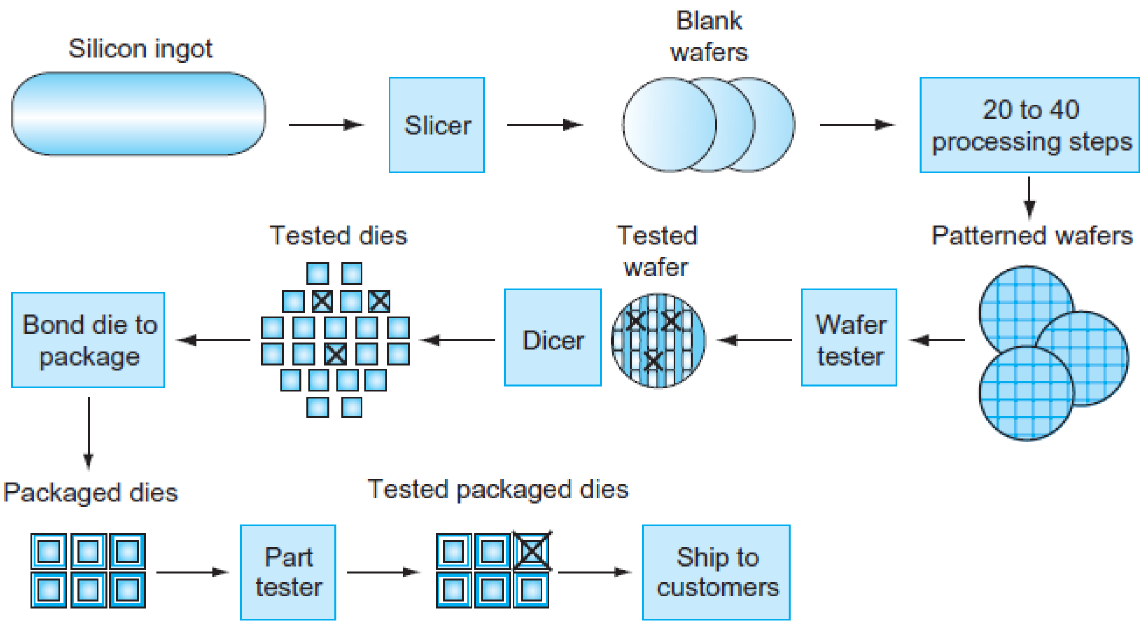
- **Non-volatile** memory
  - Maintain the data as is even though it loses power
    - e.g., Hard Disk Drives (HDDs), Solid-State Drives (SSDs)
  - Large and cheap, but slow
  - a.k.a. **secondary** memory

Manufacturing Integrated Circuits

- An Integrated Circuit (IC) consists of **transistors**.
  - On/off switches controlled by electricity

- Silicon-to-IC

  Silicon →
  Blank wafers →
  Patterned wafers →
  Tested dies →
  Packaged dies →
  Tested packaged dies (or **chips**)



Performance

Our Focus: "Make Programs Fast"

- Let's think of ourselves as **successful** programmers who are always concerned about the **performance** of our applications!

- To improve the performance, we must be aware of the primary performance **bottlenecks**.
  - In 1960s & 70s, it was the size of a computer's memory. → minimize memory space to make programs fast!
  - Now, the bottlenecks are the **parallel nature of processors** & the **hierarchical nature of memories**!
  - On PMDs and in the Cloud, we also need to consider **energy efficiency**!

Defining Performance

- When we say one computer is **faster** than another, what does it exactly mean?

- We need **performance metrics**!
  1. **Execution time**
     - The total time required for a computer to complete a task
     - Useful for single-user desktops
  2. **Throughput** (or bandwidth)
     - The total amount of work done in a given time
     - Useful for servers or WSCs which run various tasks concurrently

Comparing the Performance

- Let's say fast execution time == high performance.

$$\text{Performance}_X = \frac{1}{\text{Execution Time}_X}$$

- Now, we compare the performance of two computers:

$$\text{Performance}_X > \text{Performance}_Y$$

$$\frac{1}{\text{Execution Time}_X} > \frac{1}{\text{Execution Time}_Y}$$

$$\text{Execution Time}_Y > \text{Execution Time}_X$$

→ X is faster than Y ==
The execution time on Y is longer than that on X.

Quantifying the Performance

- We say "X is $\mathbf{n}$ **times faster than** Y." when:

$$\frac{\text{Performance}_\text{X}}{\text{Performance}_\text{Y}} = \text{n}$$

- That is,

$$\frac{\text{Performance}_\text{X}}{\text{Performance}_\text{Y}} = \frac{\text{Execution Time}_\text{Y}}{\text{Execution Time}_\text{X}} = \text{n}$$

- In the other direction, "Y is $\mathbf{n}$ **times slower than** X."

$$\frac{\text{Performance}_\text{X}}{\text{Performance}_\text{Y}} = \text{n} \qquad \text{Performance}_\text{Y} = \frac{\text{Performance}_\text{X}}{\text{n}}$$

Measuring the CPU Performance

- Is using the **wall clock time** the correct measure?
- Multiple tasks run in parallel even on single-user PCs.

→ Need to distinguish between the elapsed time & the time the CPU spent working on our tasks!

- **CPU execution time** (or **CPU time**)
  - The time the CPU spends computing for our tasks excluding time spent waiting for I/O or other tasks
  - Can be further divided into **user** & **system** CPU times
    - User CPU time: the CPU time spent in the program
    - System CPU time: the CPU time spent in the OS

Using Clock Cycles as the Measure

- Processors are **digital** circuits driven by a clock!
  - All events in a processor occur w.r.t. the clock signal.

- Use **the length of a clock period** as the measure
  - The time for a complete clock cycle
  - e.g., 1 ns for a 1-GHz clock, 250 ps for a 4-GHz clock

- Then, we get:

$$\text{CPU execution time} = \text{CPU clock cycles} \times \text{Clock cycle time}$$
$$= \text{CPU clock cycles} / \text{Clock rate}$$

Example #1

- Our program runs in 10 seconds on computer A having a 2-GHz clock.

- Q: How many clock cycles does the program take?

$$\text{CPU time}_\text{A} = {\text{CPU clock cycles}_\text{A}}\big/{\text{Clock rate}_\text{A}}$$

$$10 \text{ seconds} = {\text{CPU clock cycles}_\text{A}}\Big/{2 \times 10^9 \frac{\text{cycles}}{\text{second}}}$$

$$\text{CPU clock cycles}_\text{A} = 10 \text{ seconds} \times 2 \times 10^9 \frac{\text{cycles}}{\text{second}} = 20 \times 10^9 \text{cycles}$$

Example #2

- On computer B, our program takes 6 seconds, and 1.2 times as many clock cycles as computer A.

- Q: What is the clock rate/frequency of computer B?

$$\text{CPU time}_\text{B} = \text{CPU clock cycles}_\text{B} / \text{Clock rate}_\text{B}$$

$$6 \text{ seconds} = 1.2 \times \text{CPU clock cycles}_\text{A} / \text{Clock rate}_\text{B}$$

$$6 \text{ seconds} = 1.2 \times 20 \times 10^9 \text{cycles} / \text{Clock rate}_\text{B}$$

$$\text{Clock rate}_\text{B} = 24 \times 10^9 \text{cycles} / 6 \text{ seconds} = 4 \text{ GHz}$$

Adding Instructions to Performance

- A program consists of **instructions** generated by compilers and interpreters.

- Use **Cycles Per Instruction (CPI)** instead:

$$\text{CPU clock cycles} = \text{Instruction Count} \\ \times \text{Average clock cycles per instruction}$$

  - Instructions may take different amounts of time, so we use average CPI.

The Classic Performance Equation

- Using the CPI, the CPU time becomes:

$$\text{CPU time} = {}^{\text{CPU clock cycles}}/_{\text{Clock rate}}$$
$$= {}^{\text{Instruction count} \times \text{CPI}}/_{\text{Clock rate}}$$
$$= \text{Instruction count} \times \text{CPI} \times \text{Clock cycle time}$$

- Alternatively, we can write:

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

Eight Great Ideas in Computer Architecture

The Eight Great Ideas (1/2)

- **Moore's Law**
  - "Integrated circuit resources double every 18-24 months."
- **Abstraction**
  - Abstract/characterize the hardware design at different levels of representations
- **Make the Common Case Fast** (Amdahl's Law)
  - e.g., If functions A and B take 80% and 20% of the time, focus on function A rather than function B.
- **Parallelism**
  - Improve the performance of applications by executing operations in parallel

The Eight Great Ideas (2/2)

- **Pipelining**
  - e.g., divide laundry into three steps (e.g., wash, dry, fold), and fold clothes A while drying/washing clothes B/C
- **Prediction**
  - Guess & start executing next instructions instead of waiting
- **Hierarchy of Memories**
  - Place a faster, smaller, expensive memory near processors, and a slower, larger, and cheaper memory farther
- **Dependability via Redundancy**
  - Redundantly execute operations on redundant hardware, and compare the outcomes from the redundant hardware

Make the Common Case Fast

- "Improving one aspect of a computer would increase the overall performance of the computer!"
  - e.g., Improving A of a CPU by 20%
    improves the performance of the CPU by 20%.

- **Amdahl's Law**

$$\text{Execution time after improvement} = \frac{\text{Execution time affected by improvement}}{\text{Amount of improvement}} + \text{Execution time unaffected}$$

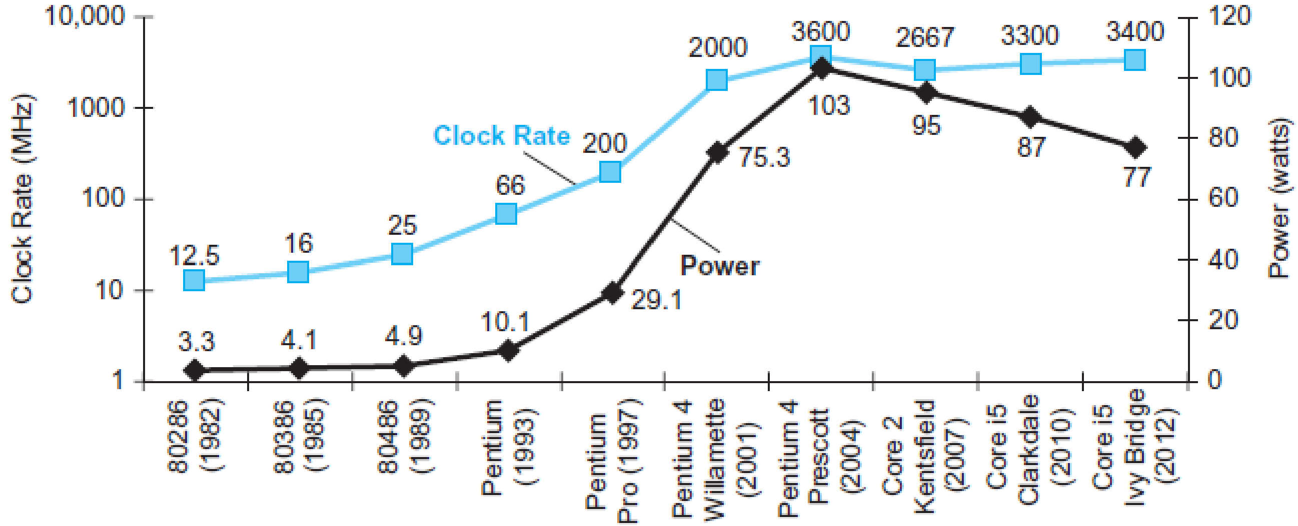
  - e.g., Making 20% of a 5-sec. application 1.2 times faster:

$$\text{Time} = \frac{0.2\times 5}{1.2} + (0.8 \times 5) = 4.83$$ ← only 0.17 seconds faster!

The Power Wall & The Switch to Multiprocessors

Trends in Clock Rate and Power

- Clock rate and power increased rapidly for decades. Then, they flattened off recently.



We ran into the practical power limit for **cooling**!

Clock Rate – Power Relationship

- ICs are typically made with **CMOS**.
  - Complementary Metal-Oxide Semiconductor (CMOS)
- In CMOS, **dynamic energy** is the primary source of energy consumption!
  - Depends on the **capacitive loading** of each transistor and the **supply voltage**:

$$Energy \propto Capacitive\ load \times Voltage^2$$

  - The energy of a single transition (0→1 and 1→0) is:

$$Energy \propto {}^{1}/_{2} \times Capacitive\ load \times Voltage^2$$

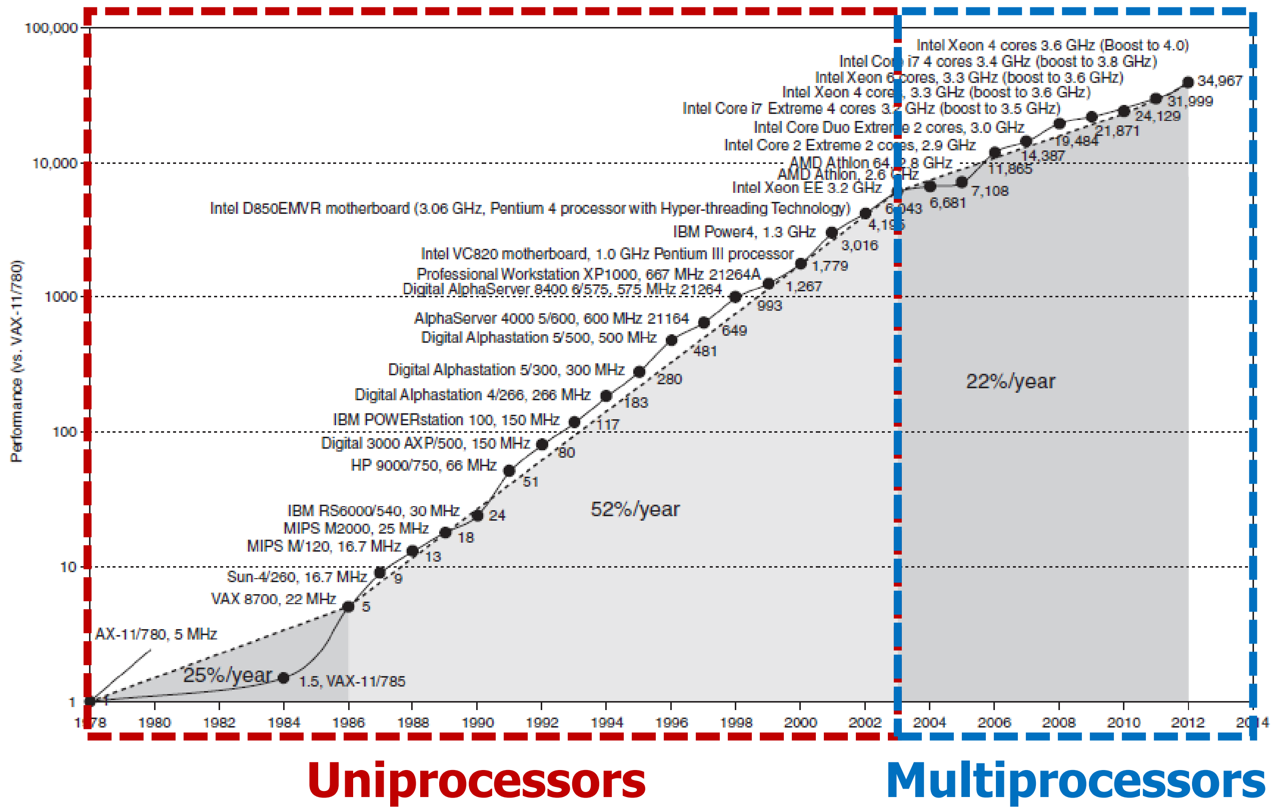
  - Then, the power required per transistor becomes:

$$Power \propto {}^{1}/_{2} \times Capacitive\ load \times Voltage^2 \times Frequency\ switched$$

where *Frequency switched* is a function of **the clock rate**.

The Switch to Multiprocessors

- Can’t increase the clock rate due to the power wall
- Okay, let’s add more **cores** instead!



The Era of Parallel Programming

- The free lunch is now over :(
  - Your program doesn't get faster anymore as the CPU's clock rate doesn't increase.

- Need to exploit **parallelism** for high performance; however, it has been & still is difficult.
  - Parallel programming increases programming difficulty.
  - Programmers must divide an application so that each processor has roughly the same amount of work.
  - Programmers must minimize inter-processor communication and synchronization.