CS3350B Computer Organization
Chapter 1: CPU and Memory
Part 1: The CPU

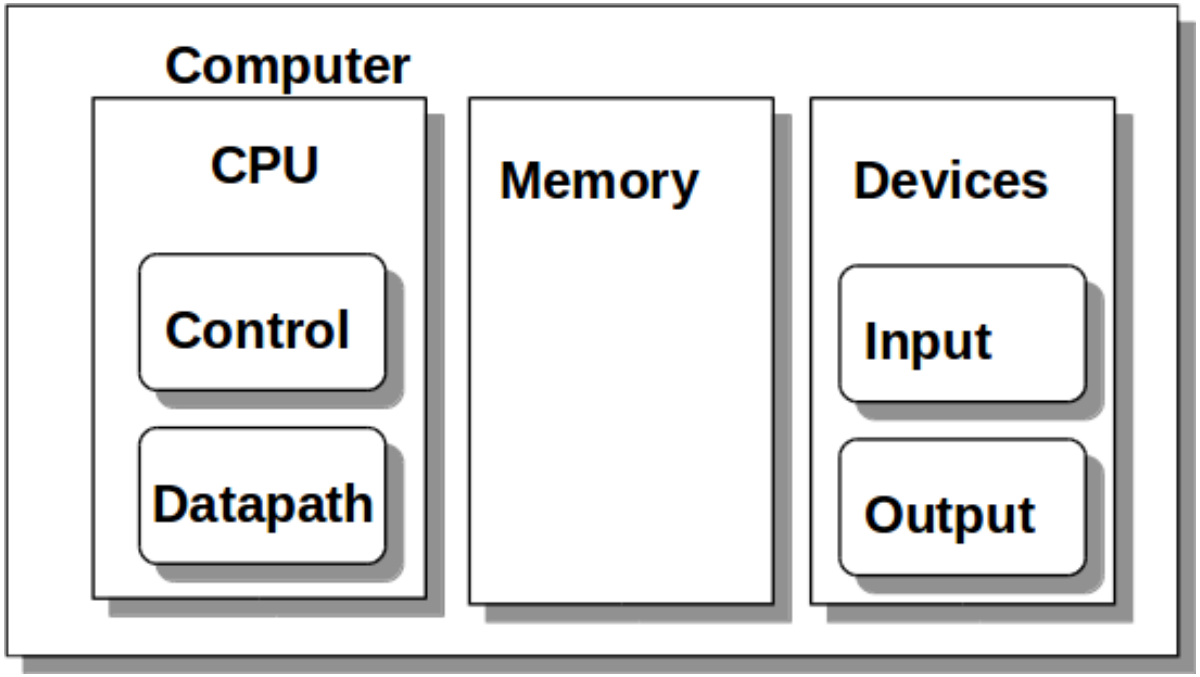
Iqra Batool

Department of Computer Science
University of Western Ontario, Canada

Monday January 9, 2024

Outline

Components of a computer



Micro-architecture: the internals of a CPU (control and datapath) will come later in the course. For now, we look at the CPU as a whole.

CPU and memory are highly coupled, especially when we look at performance. This will be seen by the end of this chapter.

CPUs and Memory: Terminology Review

CPU – Central Processing Unit

Processor – Sometimes the CPU, sometimes the actual circuitry doing the processing within the CPU.

Memory Word – The typical unit of memory that is stored, passed around, and operated on. Usually 32 or 64 bits in size.

RAM – Random Access **Memory**. A memory storage technology that can be either *static* or *dynamic*.

HDD – Hard Disk Drive.

SSD – Solid State Drive. Drives store data that persists loss of power.

Programmer's View of CPU Performance

At a basic level, the running time of some program on a CPU is determined by:

- The **clock** rate of the CPU (e.g. 3.4 GHz),
- The type of **instructions** being performed,
  - ↳ Addition/Subtraction faster than Multiplication/Division, etc.
  - ↳ Affects the *average* clock cycles per instruction (CPI)
- **Memory** access time.
  - ↳ Recall *processor-memory gap*

Assuming CPU clock rate is fixed, programmers can influence program performance by changing the type of instructions they use as well as how their code accesses memory.

Aside: Changing Instructions for Performance

On 6th-generation Intel CPUs

- 32-bit integer division ~26 clock cycles vs. logical bit shift ~1 clock cycle
  - `int i = 1234567; i /= 2;`
  - `int i = 1234567; i >>= 1;`

- Floating point division ~14 clock cycles vs. multiplication ~5 clock cycles
  - `float x = 1.2f; x /= 2.0f;`
  - `float x = 1.2f; x *= 0.5f;`

Fast Inverse Square Root thanks to *Quake*:
`https://en.wikipedia.org/wiki/Fast_inverse_square_root`

Understanding and Analyzing Performance

- **Algorithmic analysis:** Estimating the complexity of algorithms in some abstract, idealized way. In reality, two different $O(n^2)$ algorithms can have wildly different running times.
- **Programming language, compiler, architecture:**
  - ↳ Programming language and corresponding compiler determine the actual machine instructions the CPU will perform.
  - ↳ Resulting number and type of machine instructions vary by compiler and language and therefore resulting performance varies.
- **Processor and Memory:** Determines how fast instructions are executed and how fast data moves to and from the processor.
- **I/O system (including OS):** Determines how fast I/O operations are executed

Need for Performance Metrics

- Purchasing Perspective. For a collection of machine which one has the
  - ↳ "best" cost?
  - ↳ "best" cost relative to performance?
- Design perspective. Given many design options and directions which one has the
  - ↳ "best" performance improvement?
  - ↳ "best" cost relative to performance improvement?

In either case we need: (i) a basis for comparison, (ii) metrics for evaluation.

Our goal is to understand what factors in the architecture contribute to the overall system performance and the relative importance (and cost) of these factors.

CPU Performance: Latency

CPU performance is largely measured by **latency**, **throughput**, **clock frequency**.

- Want reduced response time (aka execution time, aka latency) – the time between the start and the completion of a task.
  - Important to general PC users.
- To maximize performance of some code segment (program) X, we need to minimize execution time.

$$\text{performance}_X = 1/\text{execution\_time}_X$$

- If X is n times faster than Y, then

$$\frac{\text{performance}_X}{\text{performance}_Y} = \frac{\text{execution\_time}_Y}{\text{execution\_time}_X} = n$$

- *Note:* Can also compare latency of the same single *instruction* on different CPUs.

CPU Performance: Throughput

- Want increased *throughput* - the total amount of work done in a given unit of time.
  - Important to users like data center managers.
- Again, throughput depends on the code segment being executed. Different instructions result in different throughput measures.
- Decreasing response time usually improves throughput, but other factors are important (task scheduling, memory bandwidth, etc.).

CPU Performance: Clock Frequency

Clock Frequency is a code-agnostic measure of CPU performance.

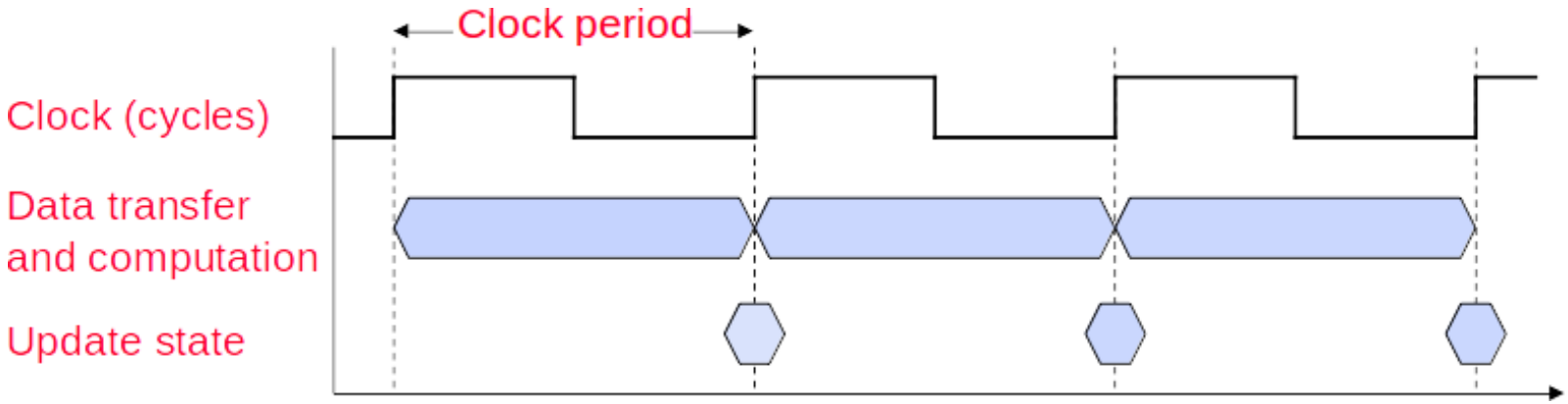
- Typically, a faster clock (higher frequency) yields a higher performing CPU.
- **But** the *micro-architecture* and the *instruction set architecture* play a large role.
  - ↳ They influence how much work the CPU does per cycle (i.e. efficiency)
- **Example:**

  CPU A runs at 3 GHz and a division takes 20 cycles.
  CPU B runs at 2 GHz and a division takes 10 cycles.

$$20 \text{ cycles} \;/\; 3\ GHz = 20 \;/\; 3000000000\ Hz = 6.66ns$$
$$10 \text{ cycles} \;/\; 2\ GHz = 10 \;/\; 2000000000\ Hz = 5.00ns$$

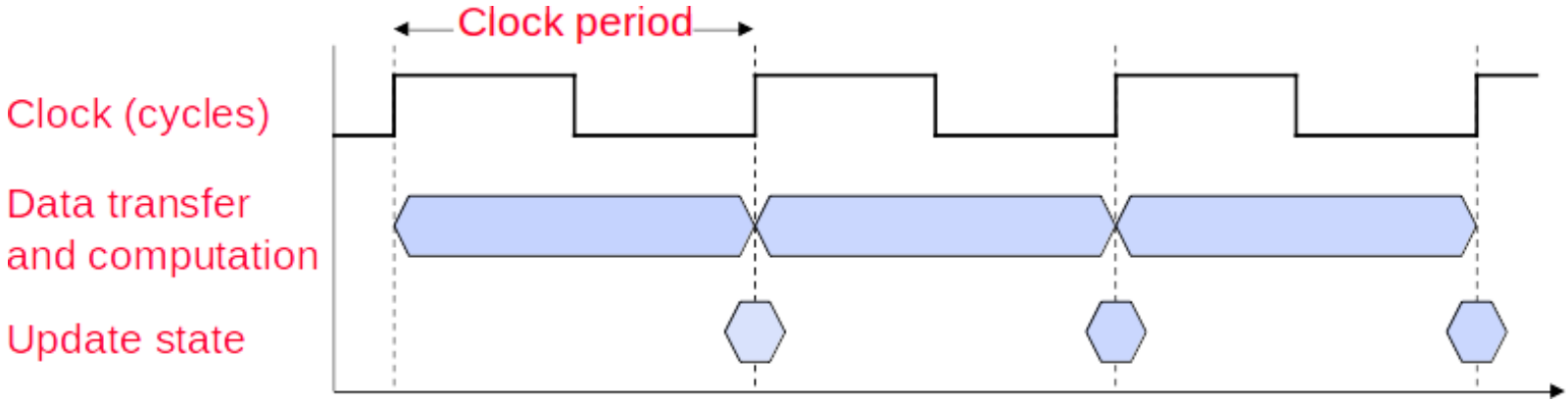
CPU Clocking

Synchronous digital systems (e.g. a CPU) are governed and controlled by a clock. This clock synchronizes internal circuits, memory states, and data movement.



Length of clock period determined by internal circuits and micro-architecture design. We will look at this with CPU datapaths and pipelining.

CPU Clocking



Clock period (cycle): duration of a clock cycle (CC)

- determines the *speed* of a computer processor
- Caveat: again, not necessarily latency or throughput though
- e.g., 250ps = 0.25ns = $250 \times 10^{-12} s$

Clock frequency or rate (CR): cycles per second

- the inverse of the clock period
- e.g., 3.0GHz = 3000MHz = 3.0×10^{9}Hz

CR = 1 / CC.

CPU Time

- It is important to distinguish *elapsed time* and the *time spent on your task*.
  - ↳ **Wall time** vs. **CPU time**
  - ↳ CPU time does not include time waiting for I/O or time spent on other processes

$$\underset{\text{for a program}}{\text{CPU execution time}} = \underset{\text{for a program}}{\#\text{CPU clock cycles}} \times \text{clockcycle}$$

or

$$\underset{\text{for a program}}{\text{CPU execution time}} = \underset{\text{for a program}}{\#\text{CPU clock cycles}} \; / \; \text{clockrate}$$

- We can improve performance by reducing either the *length of the clock cycle* or the **number of clock cycles required for a program**.

Outline

Instruction Performance

$$\begin{array}{c} \#\text{CPU clock cycles} \\ \text{for a program} \end{array} = \begin{array}{c} \#\text{Instructions} \\ \text{for a program} \end{array} \times \begin{array}{c} \text{Average \# of clock cycles} \\ \text{per instruction} \end{array}$$

- **Clock cycles per instruction (CPI)** - the average number of clock cycles each instruction takes to execute.
  - ↳ Different instructions may take different amounts of time depending on what they do.
  - ↳ A way to compare two different implementations (micro-architectures) of the same ISA.
  - ↳ Calculated by a simple averaging of each instruction type in a program and their corresponding number of cycles.
- Average CPI and Effective CPI mean the same thing and are usually shortened to just CPI.
- The CPI for a particular instruction type is usually denoted CPI_i

The Classic Performance Equation

$$\begin{aligned} \text{CPU time} &= \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle} \\ &\text{or} \\ \text{CPU time} &= \text{Instruction\_count} \times \text{CPI} / \text{clock\_rate} \end{aligned}$$

- Keep in mind that the only complete and reliable measure of computer performance is **time**.
- For example, redesigning the hardware implementation of an instruction set to lower the instruction count may lead to an organization with
  - ↳ a slower clock cycle time or,
  - ↳ higher CPI,

  that offsets the improvement in instruction count.
- *Note:* CPI depends on the type of instruction executed, so the code which executes the fewest instructions may not be the fastest.

A Simple Example (1/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i) \;/\; \text{IC}$$

Op	Inst. Freq	CPI_i	Freq $\times$ CPI_i	(1)
ALU	50%	1	.5	.5
Load	20%	5	1.0	
Store	10%	3	.3	.3
Branch	20%	2	.4	.4
			$\sum = 2.2$	

(1) How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?

A Simple Example (1/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i) \;/\; \text{IC}$$

Op	Inst. Freq	CPI_i	Freq × CPI_i	(1)
ALU	50%	1	.5	.5
Load	20%	5	1.0	.4
Store	10%	3	.3	.3
Branch	20%	2	.4	.4
			$\sum = 2.2$	1.6

(1) How much faster would the machine be if a better data cache reduced the average load time to 2 cycles?
CPU time new = 1.6 × IC × CC; so 2.2 versus 1.6 which means 37.5% faster

A Simple Example (2/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i) \;/\; \text{IC}$$

Op	Freq	CPI_i	Freq × CPI_i	(2)	(3)
ALU	50%	1	.5	.5	
Load	20%	5	1.0	1.0	1.0
Store	10%	3	.3	.3	.3
Branch	20%	2	.4		.4
			$\sum = 2.2$		

(2) How does this CPI compare with using branch prediction to save a cycle off the branch time?

(3) What if two ALU instructions could be executed at once?

A Simple Example (2/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i) \;/\; \text{IC}$$

Op	Freq	CPI_i	Freq × CPI_i	(2)	(3)
ALU	50%	1	.5	.5	
Load	20%	5	1.0	1.0	1.0
Store	10%	3	.3	.3	.3
Branch	20%	2	.4	.2	.4
			$\sum = 2.2$	2.0	

(2) How does this CPI compare with using branch prediction to save a cycle off the branch time?
CPU time new = 2.0 × IC × CC so 2.2 versus 2.0 means 10% faster

(3) What if two ALU instructions could be executed at once?

A Simple Example (2/2)

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{IC}_i) \;/\; \text{IC}$$

Op	Freq	CPI_i	Freq × CPI_i	(2)	(3)
ALU	50%	1	.5	.5	.25
Load	20%	5	1.0	1.0	1.0
Store	10%	3	.3	.3	.3
Branch	20%	2	.4	.2	.4
			$\sum = 2.2$	2.0	1.95

(2) How does this CPI compare with using branch prediction to save a cycle off the branch time?
CPU time new = 2.0 × IC × CC so 2.2 versus 2.0 means 10% faster

(3) What if two ALU instructions could be executed at once?
CPU time new = 1.95 × IC × CC so 2.2 versus 1.95 means 12.8% faster

Understanding Program Performance

$$\text{CPU Time} = \text{Instruction\_count} \times \text{CPI} \times \text{clock\_cycle}$$

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

- The performance of a program depends on the algorithm, the language, the compiler, the architecture, and the actual hardware.

	Instruction_count	CPI	clock_cycle
Algorithm	X	X	
Programming language	X	X	
Compiler	X	X	
ISA	X	X	X
Processor organization		X	X

Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by a factor of 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below:

a. $\frac{15 \times 0.6}{1.1} = 8.2$ sec
b. $15 \times 0.6 \times 1.1 = 9.9$ sec
c. $\frac{15 \times 1.1}{0.6} = 27.5$ sec

Check Yourself

A given application written in Java runs 15 seconds on a desktop processor. A new Java compiler is released that requires only 0.6 as many instructions as the old compiler. Unfortunately, it increases the CPI by a factor of 1.1. How fast can we expect the application to run using this new compiler? Pick the right answer from the three choices below:

a. $\frac{15 \times 0.6}{1.1} = 8.2$ sec
b. $15 \times 0.6 \times 1.1 = 9.9$ sec
c. $\frac{15 \times 1.1}{0.6} = 27.5$ sec

b!

$$
\begin{aligned}
Time_1 &= IC_1 \times CPI_1 \times CC_1 \\
Time_2 &= IC_2 \times CPI_2 \times CC_2 \\
&= (IC_1 \times 0.6) \times (CPI_1 \times 1.1) \times CC_1 \\
&= Time_1 \times 0.6 \times 1.1 \\
&= 15 \times 0.6 \times 1.1 = 9.9
\end{aligned}
$$

Outline

CPU Power Usage

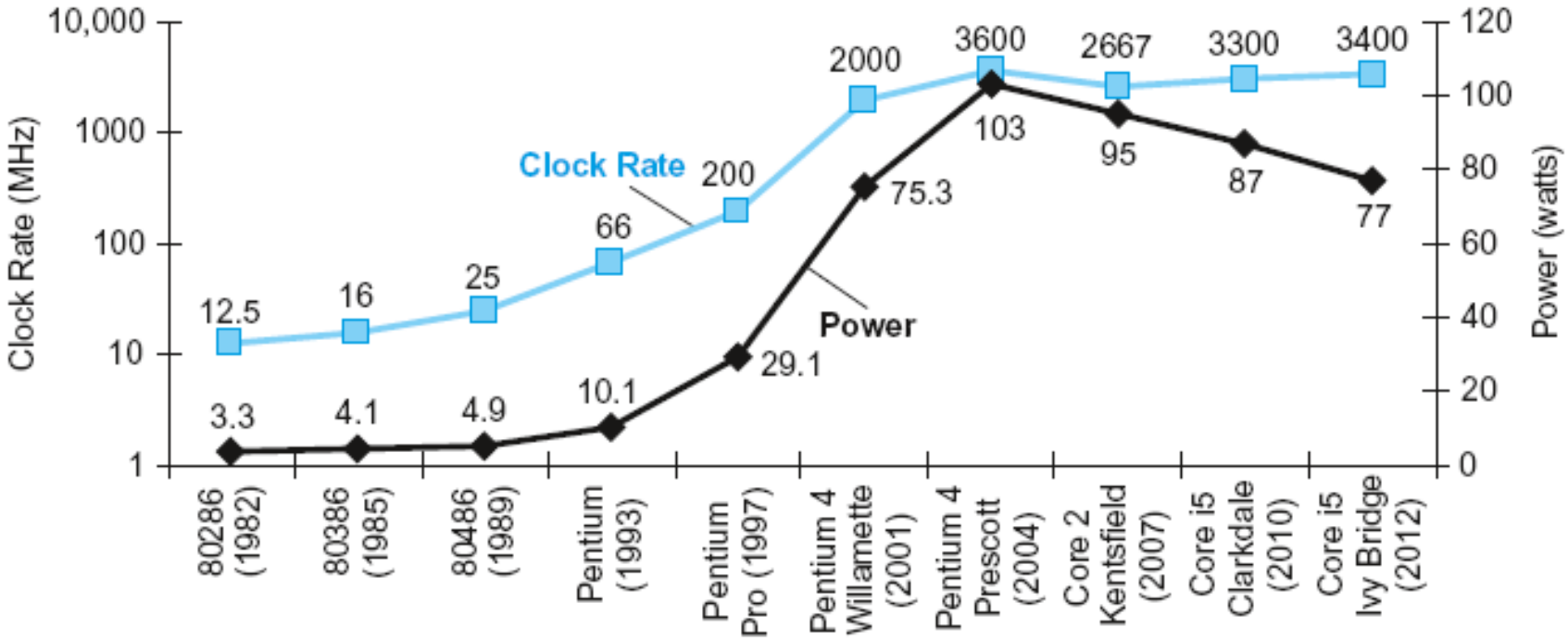
Depending on an architect's design goals they may want to look at metrics different from latency, throughput, time, or clock frequency.

$\uparrow$ Power Usage $\implies$ $\uparrow$ Temperature

$\uparrow$ Power Usage $\implies$ $\downarrow$ Battery Life

Ultrabooks are so (not) hot right now

Power Trends



Complementary metal oxide semiconductor (CMOS) integrated circuits, the technology which implements the physical circuity inside modern CPUs, has a (simplified) power equation:

$$\underset{(\times 30)}{\text{Power}} = \text{Capacitive load} \times \underset{(5V \rightarrow 1V)}{\text{Voltage}^2} \times \underset{(\times 1000)}{\text{Frequency switched}}$$

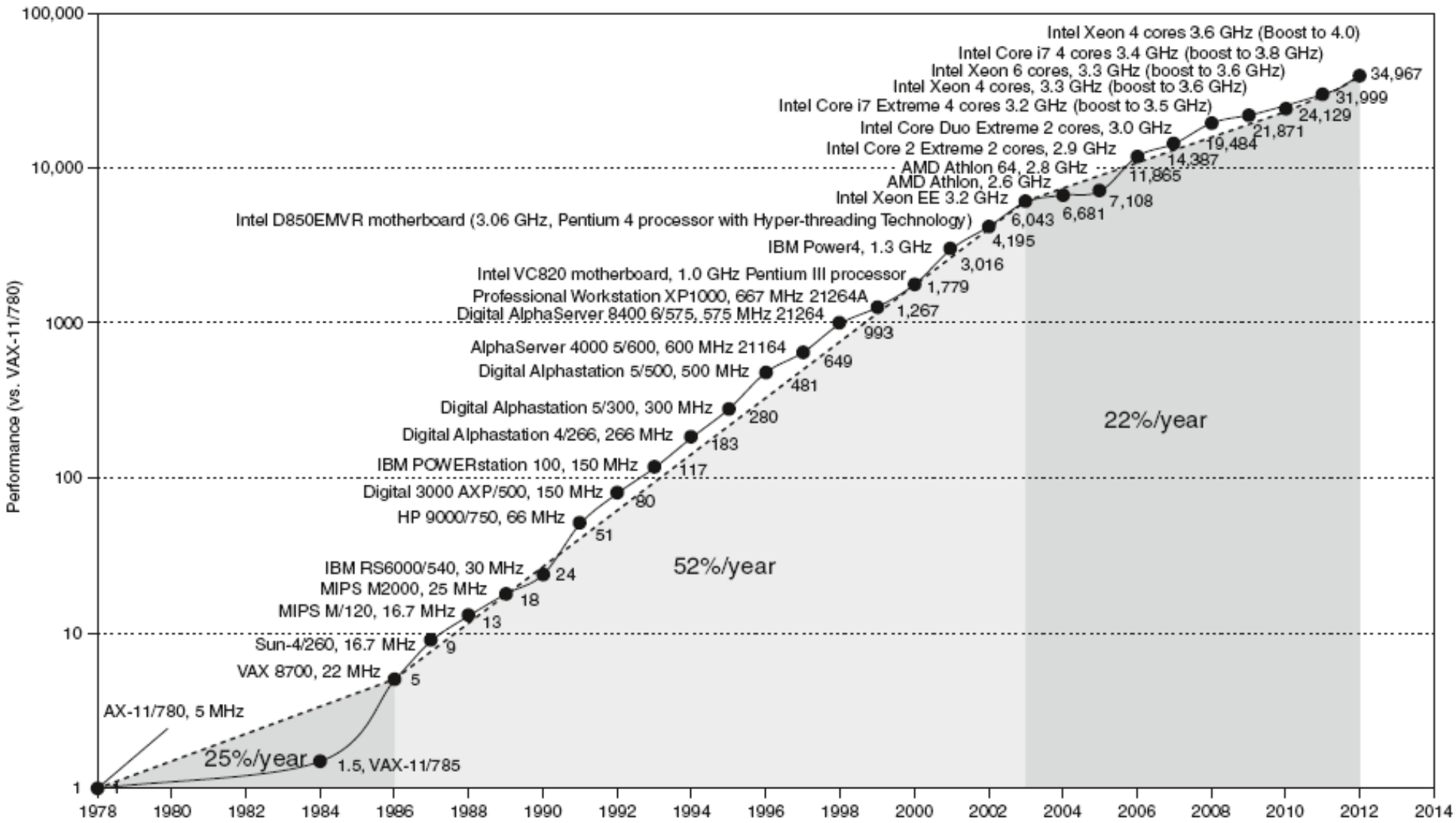
Reducing Power

- Suppose a new CPU has
  - 85% of capacitive load of old CPU
  - 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^2 \times F_{old}} = 0.85^4 = 0.52$$

- **The Power Wall**
  - Simply cannot consume any more power.
  - Decreasing transistor size $\Longrightarrow$ more transistors per chip $\Longrightarrow$ greater power density
  - Required input voltage may decrease but more transistors use more power.
  - Reducing voltage further very difficulty.
  - Removing extra heat very difficult.

CPU Performance: Relative Performance vs VAX-11



Multi-Processors to the Rescue

Moore's Law failing? Great idea #4: Performance via Parallelism.

- Multi-core processors!
  - ↳ More than one processor per chip
  - ↳ Huge benefits to OS and multiple processes.
- Within a single process requires explicit parallel programming
  - ↳ Compared with instruction level parallelism:
    - Hardware executes multiple instructions at once (pipelining/multi-issue)
    - Hidden from the programmer
  - ↳ Hard to do:
    - Programming for performance
    - Thread management, Load Balancing
    - Optimizing communication and synchronization

Outline

SPEC CPU Benchmark

- Programs used to measure performance
  - ↳ Supposedly typical of actual workload
- Standard Performance Evaluation Corp (SPEC)
  - ↳ Develops benchmarks for CPU, I/O, Web, ...
- SPEC CPU2006
  - ↳ Elapsed time to execute a selection of programs
    - Negligible I/O, so focuses on CPU performance
  - ↳ Normalize relative to reference machine
  - ↳ Summarize as geometric mean of performance ratios
    - CINT2006 (integer) and CFP2006 (floating-point)

$$\sqrt[n]{\prod_{i=1}^{n} \text{Execution time ratio}_i}$$

CINT2006 for Intel Core i7 920

Description	Name	Instruction Count x 10^9	CPI	Clock cycle time (seconds x 10^{-9})	Execution Time (seconds)	Reference Time (seconds)	SPECratio
Interpreted string processing	perl	2252	0.60	0.376	508	9770	19.2
Block-sorting compression	bzip2	2390	0.70	0.376	629	9650	15.4
GNU C compiler	gcc	794	1.20	0.376	358	8050	22.5
Combinatorial optimization	mcf	221	2.66	0.376	221	9120	41.2
Go game (AI)	go	1274	1.10	0.376	527	10490	19.9
Search gene sequence	hmmer	2616	0.60	0.376	590	9330	15.8
Chess game (AI)	sjeng	1948	0.80	0.376	586	12100	20.7
Quantum computer simulation	libquantum	659	0.44	0.376	109	20720	190.0
Video compression	h264avc	3793	0.50	0.376	713	22130	31.0
Discrete event simulation library	omnetpp	367	2.10	0.376	290	6250	21.5
Games/path finding	astar	1250	1.00	0.376	470	7020	14.9
XML parsing	xalancbmk	1045	0.70	0.376	275	6900	25.1
Geometric mean	–	–	–	–	–	–	25.7