Chapter 1 — Computer Abstractions and Technology

1.1 Introduction

1.3 Below your program

1.4 Under the covers

1.5 Technologies for building processors and memory

1.6 Performance

1.7 The power wall

1.8 The sea change: the switch from uniprocessors to multiprocessors

1.1 Introduction: What is computer organization and what is computer architecture?

Computer architecture

Encompasses those aspects of a computing system that are visible or important to the programmer.

- Instruction set - assembly language instructions that can be executed by the processor.
- CPU registers - storage within the processor; how many, names, sizes, usage limitations.
- Data representation - signed and unsigned integers, characters, real numbers, etc; sizes.
- I/O - memory mapped (devices accessed by reading and writing memory), ports (special instructions).
- Memory - how much, address space, segmentation, cache levels.
- Instruction addressing modes - where does data come from and where does it go when an instr is exec'd.

Examples of computer architecture related questions.

- Does our machine have a multiply or divide instruction?
- How do we write assembly language code to execute instructions from the instruction set?
- How long does it take to execute each instruction?
- How many bytes of memory does each instruction take?

Computer organization

Encompasses how to design and build the hardware to implement the architecture (architecture comes first).

Focus on computer performance, subject to constraints.

- Size - of chips and components (generally smaller is better); feature size.
- Cost - to manufacture integrated circuits.
- Power consumption - especially important in mobile systems.
- Backward compatibility - can we run programs designed for preceding generations of the architecture.
- Design to support future enhancements to the architecture.

Examples of computer organization related questions.

- How do we build the fastest hardware multiplier we can subject to various constraints.
- How can we design the circuits to meet a specific clock frequency goal.
- How do we design the circuits so power consumption is less than 25 watts (W).

1.3 Below your program

Machine language

Binary encoding of instructions and data.
The language of the machine.

Assembly language

Symbolic representation of instructions that maps directly to the architecture of the machine.
Instruction set architecture (ISA) - specific to each architectural version of a microprocessor.
An assembler is a tool that translates assembly language into machine language.

High-level languages (HLL)

Abstract away the hardware details of the machine to increase: programmer productivity, portability.
Productivity - allows us to write code that is closer to the problem domain.
Portability - source code can be modified as required and recompiled to run on a different machine.
HLL's are so advantageous that very little code is written in assembly today (hw specific, speed).
A compiler translates HLL code into assembly language code or directly to machine language.

1.4 Under the covers

The five classic components of all computer systems

1. Input devices.
2. Output devices.
3. Storage (memory).
   Primary memory. Today, volatile (RAM: SRAM[1], DRAM[2]). Future: nonvolatile (MRAM[3], FeRAM[4]).
   Secondary memory. Nonvolatile (ROM: Flash, hard disk, optical disc).

Units for measuring information[5]

Historically, metric prefixes (kilo, mega, giga, etc) have been variably used to represent either 2^{10x} ($x = 1$ for kilo, $x = 2$ for mega, $x = 3$ for giga, ...) or 10^{3x}, although only the latter is technically correct. It would not be so bad to use the power-of-2 representation if it were applied consistently but it has not been. For example, when discussing memory sizes, giga means 2^{30} so 1 GB of memory is 1,073,741,824 bytes. However, when discussing disk sizes, giga has historically meant 10^9 so a 1 GB disk stores 1,000,000,000 bytes. Because the U.S. possess the bulk of the world's lawyers, and because there are too many lawyers in the world, there have been some important legal disputes[6] regarding this discrepancy.

To address the issue, IEEE, ISO, and the International Electrotechnical Commission (IEC) established new prefixes (colloquially referred to as the IEC prefixes) in 1999. The new terms are formed from the first two letters of the metric terms, "ki" for kilo, and the first two letters of the word "binary"; hence a kibibyte is 2^{10} bytes whereas a kilobyte is 10^3 bytes. Other units:

Kilobyte (KB)	10^3	Kibibyte (KiB)	$2^{10} = 1024$
Megabyte (MB)	10^6	Mebibyte (MiB)	$2^{20} = 1024^2$
Gigabyte (GB)	10^9	Gibibyte (GiB)	$2^{30} = 1024^3$
Terabyte (TB)	10^{12}	Tebibyte (TiB)	$2^{40} = 1024^4$
Petabyte (PB)	10^{15}	Pebibyte (PiB)	$2^{50} = 1024^5$
Exabyte (EB)	10^{18}	Exbibyte (EiB)	$2^{60} = 1024^6$
Zettabyte (ZB)	10^{21}	Zebibyte (ZiB)	$2^{70} = 1024^7$
Yottabyte (YB	10^{24}	Yobibyte (YiB)	$2^{80} = 1024^8$

4. Datapath - hardware component that manipulates bits to execute instructions.
5. Control - hardware component that coordinates the activities of the datapath.

Central Processing Unit (CPU) = datapath + control (historically)
CPU = datapath + control + memory (modern)

1 Static RAM. Generally, SRAM is faster and more expensive than DRAM. It is primarily used to build cache memory (see Chapter 5).
2 Dynamic RAM. Cheaper and slower than SRAM. Primary memory in a system. Stores bits representing instructions and data of executing programs.
3 Magnetoresistive RAM.
4 Ferroelectric RAM.
5 http://en.wikipedia.org/wiki/Binary_prefix
6 http://en.wikipedia.org/wiki/Binary_prefix#Legal_disputes

1.5 Technologies for building processors and memory

Fig 1.10

Year	Technology used in computers	Relative performance/unit cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit	900
1995	Very large-scale integrated circuit	2,400,000
2013	Ultra large-scale integrated circuit	250,000,000,000

Vacuum tubes and electromechanical relays - 1930's to early 1960's.

Transistor - 1950's to mid 1960's.

Integrated circuits (IC's) - mid 1960's to today.

Small scale integration (SSI) - tens of transistors on a single die.
Medium scale integration (MSI) - hundreds of transistors.
Large scale integration (LSI) - thousands of transistors[7]
Very large scale integration (VLSI) - hundreds of thousands of transistors.
Ultra large scale integration (ULSI) - millions of transistors.
IFCSI[8] - sometime off in the future.

Moore's Law (Gordon Moore)

Number of transistors crammed onto an IC doubles approximately every two years.

If a microprocessor contains 250,000,000 transistors today, then in two years, we will be able to squeeze 500,000,000 transistors onto the same-sized IC. In two more years (a total of four years), 1,000,000,000. This is exponential (power-of-2) progress.

Corollary: Performance doubles every 18-24 months.

It is unknown how long this trend can continue. It could be 10 yrs, 20 yrs, or ...[9]. In 2012, Purdue University researchers published a paper describing how they constructed a working transistor from a single phosphorous atom[10]. They are continuing to make progress in this area. Subatomic transistors seem inevitable.

7 The first microprocessor was the 1971 Intel 4004 which consisted of 2300 transistors. The highest transistor count in a commercially available CPU is 4.3 billion (the Intel Xeon IvyBridge-EX), which could contain almost 1.9 million 4004's.
8 Insanely Frickin' Crazy Scale Integration.
9 http://www.techradar.com/news/computing/moore-s-law-how-long-will-it-last--1226772/2
10 http://www.purdue.edu/newsroom/research/2012/120219KlimeckAtom.html

1.6 Performance

Defining performance (Fig 1.14)

Airplane	Passenger capacity	Cruising range (miles)	Cruising speed (m.p.h.)	Passenger throughput (passengers x m.p.h.)
Boeing 777	375	4630	610	228,750
Boeing 747	470	4150	610	286,700
BAC/Sud Concorde	132	4000	1350	178,200
Douglas DC-8-50	146	8720	544	79,424

Which airplane has the best performance? That depends on how we define performance:

Carries the most passengers from point A to point B? Boeing 747
Moves from point A to point B the fastest? Concorde
Flies the farthest without having to land and refuel? Douglas DC-8-50

Conclusion: It is not so easy to compare things having different characteristics. The same holds true for computer systems: it is not necessarily true that a 2 GHz system will execute software faster than a 1.8 GHz system, i.e., there are numerous architectural and organizational characteristics that affect performance.

Two computer system performance metrics

Execution Time (response time) - Time to complete a program or task.
Throughput (bandwidth) - Number of programs or tasks completed per time unit.

Individual computer users are generally interested in execution time. Data center and mainframe managers are generally interested in throughput.

The initial focus of the book will be on execution time. Chapter 6 (Parallel Processors...) focuses more on throughput.

To maximize performance of a program P on computer system X, we minimize the execution time of P on X

$perf_{X(P)} = 1 \ / \ t_{exec,X(P)}$

Relative performance (n) of computer X compared to computer Y (on program P)

$perf_{rel,X(P),Y(P)} = perf_{X(P)} \ / \ perf_{Y(P)}$

Case 1: $perf_{rel,X(P),Y(P)} = 1$
Performance of X and Y are identical.

Case 2: $perf_{rel,X(P),Y(P)} > 1$
$perf_{X(P)} > perf_{Y(P)} == t_{execX(P)} < t_{execY(P)}$
We say X is $perf_{rel,X(P),Y(P)}$ times faster than Y (or Y is $1/perf_{rel,X(P),Y(P)}$ times as fast as X).

Case 3: $perf_{rel,X(P),Y(P)} < 1$
$perf_{X(P)} < perf_{Y(P)} == t_{execX(P)} > t_{execY(P)}$
We say X is $1/perf_{rel,X(P),Y(P)}$ times as fast as Y (or Y is $perf_{rel,X(P),Y(P)}$ times faster than X).

Example:

Computer A runs a program P in 10 secs, B runs P in 15 secs.

$t_{\text{execA(P)}} = 10$ secs, $perf_{\text{A(P)}} =$ _____.
$t_{\text{execB(P)}} = 15$ secs, $perf_{\text{B(P)}} =$ _____.

Relative performance of A compared to B (on P).

$perf_{\text{rel,A(P),B(P)}} = perf_{\text{A(P)}} \ / \ perf_{\text{B(P)}} =$ _____.
$perf_{\text{rel,A(P),B(P)}} > 1$: A is _____ times faster than B.

Relative performance of B compared to A.

$perf_{\text{rel,B(P),A(P)}} = perf_{\text{B(P)}} \ / \ perf_{\text{A(P)}} =$ _____.
$perf_{\text{rel,B(P),A(P)}} < 1$: B is _____ as fast as A.

Measuring performance

Execution Time (t_{exec}; also referred to as *elapsed time, wall time, real time*, *response time*): Elapsed seconds from start to finish. Includes time for:

1. Executing instructions (this is called *CPU Time*, see below).
2. Disk accesses - processor cannot execute instructions when waiting for data to arrive.
3. Memory accesses - fetching instructions, reading and writing operands (data).
4. I/O activities - accessing input and output devices.
5. OS overhead - system calls, context switches in a multitasking system.

CPU Time (t_{cpu}; also referred to as *CPU execution time*): Only counts the time the CPU spends executing instructions. Excludes time for items 2-5, above.

CPU time can be partitioned into:

User CPU Time ($t_{\text{cpu(user)}}$): Seconds the CPU spends executing the program's instructions.

System CPU Time ($t_{\text{cpu(sys)}}$): Seconds the CPU spends executing OS instructions on behalf of the program

For example, the Unix command `time` will run a program and at the end, display three numbers which are the execution time (response time), user CPU time, and system CPU time. On my Linux computer, running `time ls -R /usr` displayed (this command displays all the files in the directory tree rooted at */usr*),

`real`	`0m17.420s`	this number is t_{exec}
`user`	`0m1.269s`	this number is $t_{\text{cpu(user)}}$
`system`	`0m2.253s`	this number is $t_{\text{cpu(sys)}}$

`real` is the *execution time*, i.e., this program appeared to require 17.42 seconds to complete; `user` is the *user CPU time*, i.e., the CPU spent 1.269 seconds executing the instructions of my program; `system` is the *system CPU time*, i.e., the CPU spend 2.253 executing OS code. The *CPU time* is the sum of *user CPU time* and *system CPU time*, i.e., $t_{\text{cpu}} = t_{\text{cpu(user)}} + t_{\text{cpu(sys)}} = 1.269$ s+ 2.253 s = 3.522 s. And what was the CPU doing during the 17.42 s - 3.522 s = 13.898 s it was not executing instructions? See items 2-5 above. In particular, this `ls` command is an example of an I/O-intensive program: it spent most of its time waiting around for disk accesses and prints to the terminal to complete.

There is a distinction between performance based on *execution time* and performance based on *CPU time*.

System Performance

Execution time on an unloaded single-tasking system: $perf_{sys} = 1 \ / \ t_{exec}$.

Includes OS overhead, so system performance is highly dependent on the efficiency of the OS.

CPU Performance

The inverse of *user CPU time*, i.e., $perf_{cpu(user)} = 1 \ / \ t_{cpu(user)}$.

Excludes OS overhead, so CPU performance is highly dependent on the CPU organization.

To increase CPU performance we must design the system to decrease user CPU time.

Bottleneck

Something which restricts performance.

The canonical computer system bottleneck is the CPU to main memory access time.

To increase performance, bottlenecks must be reduced or eliminated.

All modern processors are based on synchronous digital logic which means that events occur at specific times.

System Clock: Device which generates the master clock signal. Coordinates the activities of the system.

Clock Cycle (*clock tick, tick, clock period, clocks, cycles*).

Time from low-to-high (high-to-low) clock edge to next low-to-high (high-to-low) edge.

Clock Period (denoted by p; other terms: *clock cycle time, cycle time*): Time for one complete clock cycle.

Units for measuring time:

Milliseconds (ms)	10^{-3} sec
Microseconds (μs)	10^{-6} sec
Nanoseconds (ns)	10^{-9} sec
Picoseconds (ps)	10^{-12} sec

Clock frequency (f; other terms: *clock rate*, *tick rate*): The inverse of the clock period, i.e., $f_{clock} = 1 \ / \ p_{clock}$.

Units for measuring frequency:

Hertz (Hz)	1 time per second
Kilohertz (KHz)	10^{3} times per second
Megahertz (MHz)	10^{6} times per second
Gigahertz (GHz)	10^{9} times per second

CPU performance and its factors

Relationship between CPU performance (user CPU time) and the system clock,

$t_{cpu(user)} = prog_{clocks} \times p_{clock}$ Eqn 1

$t_{cpu(user)} = prog_{clocks} \ / \ f_{clock}$

To decrease user CPU time (increase CPU performance) we can:

1. Decrease $prog_{clocks}$, and/or
2. Decrease p_{clock} (which is the same as increasing f_{clock}). For the first 65-70 years of computing, increasing the system clock was the easiest way to increase performace. That trend has stopped for reasons we will discuss in a bit.

Instruction performance

There are two measures of *instruction count*:

Static instruction count ($icount_{\text{stat}}$) is the number of machine language instructions in the program.

Dynamic instruction count ($icount_{\text{dyn}}$) is the number of machine language instructions that are executed. While the static instruction count is a fixed value (unless the high-level language program is modified and recompiled) the dynamic instruction will vary from run to run as different sequences of instructions will be executed when the program input varies.

The number of clocks to execute one instruction:

Depends on the particular instruction and the organization of the CPU.

It may be a fixed number for each instruction, e.g., 1 (or 2, or 3, ...). Or, it may vary from 1, 2, 3, ..., among the instruction set, i.e., some instructions might take 1 clock, some might take 2, others might take 3, and so on.

Clocks Per Instruction (*CPI*).

The average number of clock cycles each instruction of a program or program fragment takes to execute. Calculated as the sum of the clock cycles for the instructions divided by the dynamic instruction count:

$CPI = prog_{\text{clocks}} \ / \ icount_{\text{dyn}}$

Knowing $icount_{\text{dyn}}$ and CPI we have $prog_{\text{clocks}} = icount_{\text{dyn}} \times CPI$ Eqn 2

The inverse of CPI is *Instructions Per Clock*: $IPC = 1 \ / \ CPI = icount_{\text{dyn}} \ / \ prog_{\text{clocks}}$.

The classic CPU performance equation

$t_{\text{cpu(user)}} = prog_{\text{clocks}} \times p_{\text{clock}}$	Eqn 1
$prog_{\text{clocks}} = icount_{\text{dyn}} \times CPI$	Eqn 2
$\boldsymbol{t_{\text{cpu(user)}} = icount_{\text{dyn}} \times CPI \times p_{\text{clock}}}$	Eqn 3 (combining Eqn 1 and Eqn 2)
$\boldsymbol{t_{\text{cpu(user)}} = (icount_{\text{dyn}} \times CPI) \ / \ f_{\text{clock}}}$	Since $f_{\text{clock}} = 1 \ / \ p_{\text{clock}}$

Eqn 3 (which is a function of three variables) tells us that to improve CPU performance—decreasing user CPU time—we must:

1. Decrease $icount_{\text{dyn}}$, and/or
2. Decrease CPI, and/or
3. Decrease p_{clock} (which is the same as increasing f_{clock}).

These are not mutually exclusive:

1. Altering the design to reduce either the static or dynamic instruction count could increase CPI or p_{clock}.

   Reducing the static or dynamic instruction count for a program generally means that more complicated instructions have to be implemented in hardware. For example, a processor may have an instruction that loads a value from memory into a processor register (LOAD), an instruction that adds a constant to a value in a register (ADDC), and an instruction that writes the value in a register to memory (STORE). To implement a C `++var;` statement would require a LOAD instruction to move the value of *var* from memory to a register, an ADDC instruction to add 1 to the value in the register, and a STORE instruction to move the incremented value from the register back to memory. In C, C++, and Java, `++var;` is a very common statement, so suppose the designers of the system decide to implement an instruction that will perform this statement in one machine language

instruction: INCM (increment variable in memory). For every occurrence of `++var;` in a C program, the new design would reduce both the static and dynamic instruction counts by a factor of 3, which seems like a good thing. However, the addition of the INCM instruction takes time to execute, and suppose the extra time is such that p_{clock} must be increased by x ns. These two changes will affect Eqn 3, as it depends on both $icount_{\text{dyn}}$ and p_{clock}. If the increase in p_{clock} is such that even though $icount_{\text{dyn}}$ is decreased, user CPU time increases, then there would be no advantage to implementing the INCM instruction.

2. Altering the design to decrease p_{clock} may increase *CPI*.

   In a **single-cycle** design—where every instruction completes in exactly one clock cycle—p_{clock} is basically dependent on the time it takes to execute the slowest instruction of the instruction set. In the single-cycle MIPS microprocessor design we will study in Chapter 4, this is the `lw` instruction (load word from memory into a register). Suppose $p_{clock} = 200$ ns (f_{clock} would be 50 MHz) and that `lw` completes in $t_{\text{ld}} = 180$ ns $= 1$ clock cycle (p_{clock} must be a bit longer than t_{ld} to account for signal propagation delays, rise and fall times, etc). Now, suppose the designers of the system decide, for whatever reason[11], that p_{clock} must be decreased to 150 ns (increasing f_{clock} to 66.67 MHz). Without making any other changes in the design, t_{ld} will still be 180 ns, which means it will now take two clock cycles to execute (this is not a necessarily a problem; a design where instructions complete in a variable number of clock cycles is called a **multi-cycle** design). This will increase the number of clocks it takes for a program to execute, so consequently, *CPI* will increase.

3. Altering the design to decrease *CPI* requires either $prog_{\text{clocks}}$ to decrease (because it appears in the numerator of the CPI equation) or $icount_{\text{dyn}}$ to increase (because it appears in the denominator) or a concomitant decrease and increase in both, respectively.

   In general, decreasing *CPI* is a good thing. Without altering the clocks that it takes each instruction to execute, increasing $icount_{\text{dyn}}$ will certainly decrease *CPI* but from Eqn 3 this will increase user CPU time—which we are trying to decrease. Increasing $icount_{\text{dyn}}$ *would* be a successful strategy for decreasing *CPI* as long as there is a concomitant decrease in the number of clocks those instructions take to execute. In fact, as we shall see in Chapter 4, this was the driving force behind the RISC design philosophy.

It is important to understand that *CPI* depends on the particular instructions that are executed, Minimizing $icount_{\text{dyn}}$ in Eqn 3 by executing a different sequence of instructions may increase *CPI* enough that $t_{\text{cpu(user)}}$ would actually increase. Example:

A system has three classes of instructions: *A*, *B*, and *C*. The *CPI* for *A* class instructions is 1, for *B* is 2, and for *C* is 3. Suppose two different code sequences perform the same operation using different instruction classes. Code sequence 1: A B C A C; code sequence 2: A A B C A A.

Which code sequence executes the most instructions?

$icount_{dyn,1} = 5$ instr.

$icount_{\text{dyn},2} = 6$ instr.

11 Because the geniuses in marketing want to advertise that their computer system is faster than their competitor's computer.

How many clocks does each sequence take?

$prog_{\text{clocks},1}$ = _____.

$prog_{\text{clocks},2}$ = _____.

What is the *CPI* for each sequence?

$CPI_1 = prog_{\text{clocks},1} \ / \ icount_{\text{dyn},1}$ = _____.

$CPI_2 = prog_{\text{clocks},2} \ / \ icount_{\text{dyn},2}$ = _____.

If p_{clock}= 2 ns then

$t_{\text{cpu(user)},1}$ = _____.

$t_{\text{cpu(user)},2}$ = _____.

Therefore, the _____ code sequence actually takes more time to execute.

Measuring performance in practice

Execution time, user CPU time, and system CPU time can be measured by running the program and measuring the time from begin to end (the Linux `time` command will do this). Or, many programming language libraries contain time-related functions which permit the programmer to measure time.

The user CPU time equation is a function of dynamic instruction count, average CPI, and the clock period. The clock period (clock frequency) is known—although modern processors will vary the clock frequency to conserve power (in general, slowing down the clock will conserve power and speeding it up will consume power). The dynamic instruction count can be measured by executing the program on a CPU simulator, or, because performance is so important, modern CPU's now include dedicated hardware for collecting run time information, such as dynamic instruction count, CPI, and so on. Intel, in their Core architecture, refers to this hardware as the Performance Monitoring Unit[12] (PMU).

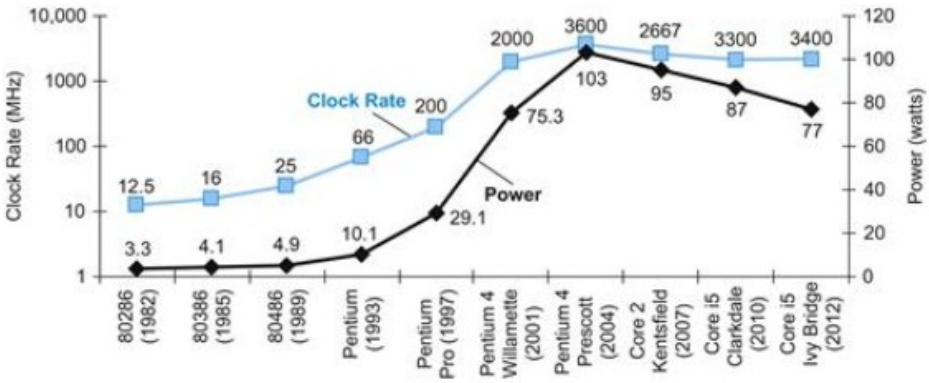
Increasing performance is of such importance that entire books and multiple courses could be dedicated to the topic. This course just gives us a high-level view of performance.

12 See Chapter 18 of the Intel 64 and IA-32 Architectures Software Developers Manual (the document is on the course website).

1.7 The power wall (why not just crank up the clock frequency?)

The general trend for many years was to increase the system clock frequency to improve performance. That trend reached a limit in the early 2000's with the end of the Pentium 4. Why?

Fig. 1.16



The system clock frequency controls the switching frequency of CMOS[13] transistors and CMOS transistors primarily consume power (referred to as *dynamic energy*) when switching from 0-to-1 or 1-to-0.

$$\textit{CMOS transistor dynamic energy (0-to-1 or 1-to-0)} \propto 1/2 \times \textit{capacitive load} \times V^2$$

Capacitive load is dependent on the fanout of the transistor and the device technology.

$$\begin{aligned}\textit{CMOS transistor power} &\propto \textit{CMOS transistor dynamic energy} \times \textit{switching frequency} \\ &\propto 1/2 \times \textit{capacitive load} \times V^2 \times \textit{switching frequency}\end{aligned}$$

Therefore increasing the system clock frequency increases power consumption, with a concurrent need to dissipate more heat.

How to reduce power consumption?
1. Decrease the system clock frequency (switching frequency), and/or,
2. Decrease the capacitive load, and/or,
3. Decrease the voltage.

In general, from Eqn 3, decreasing the system clock frequency is undesirable because that increases user CPU time (reduces performance).

Decreasing capacitive load is not simple as it depends on the transistor technology, e.g., CMOS, and the *fanout* (the number of transistors connected to an output).

Decreasing voltage is most feasible and that has been the trend in microprocessor design over the last 20 years. We have moved from 5V down to 3.3V down to 1.7V, and maybe even lower, I'm not sure. Unfortunately, in MOSFET transistors, as voltage decreases, transistors begin to leak current (aptly named leakage current[14]) and consequently, further reductions in voltage seem to be impossible due to the unacceptable increases in power consumption.

13 Complementary Metal Oxide Semiconductor.
14 http://en.wikipedia.org/wiki/Leakage_(electronics)#In_semiconductors

1.8 The sea change: the switch from uniprocessors to multiprocessors

We started the section on performance by stating that we could measure performance by execution time or response time. We then proceeded to discuss execution time which led us to the power wall.

What if we focus on throughput?

Problem: we need to make a lot of toast.

Solution 1: Get a single slot toaster and make one piece of toast at a time. Slow.
Solution 2: Get two single slot toasters and make two pieces of toast at a time.
Solution 3: Get four single slot toasters and make four pieces of toast at a time.
...
Solution n: Get n single slot toasters and make n pieces of toast at a time. Fast.

A multicore microprocessor is a microprocessor that contains multiple processors (cores).

Problem: we need to run a lot of programs.

Solution 1: Get a single microprocessor and start running one program at a time. Slow.
Solution 2: Get two microprocessors and start running two programs at a time.
Solution 3: Get four microprocessors and start running four programs at a time.
...
Solution n: Get n microprocessors and start n programs at a time. Fast.

Note that *a multiprocessor does not necessarily decrease the execution time of any one program but it certainly can increase throughput* (the average execution times of multiple programs). To decrease the execution time of an individual program we could write the code in such a way that some of the instructions are executed by one core while the other instructions are executed by a second core.

Unfortunately, *that* is not so easy because it requires changes to the way we design, write, debug, and test programs. For the last 70 years we have essentially been writing sequential computer programs and the switch to parallel programming substantially increases the complexity of programming.

How to do this is an important and active area of research.