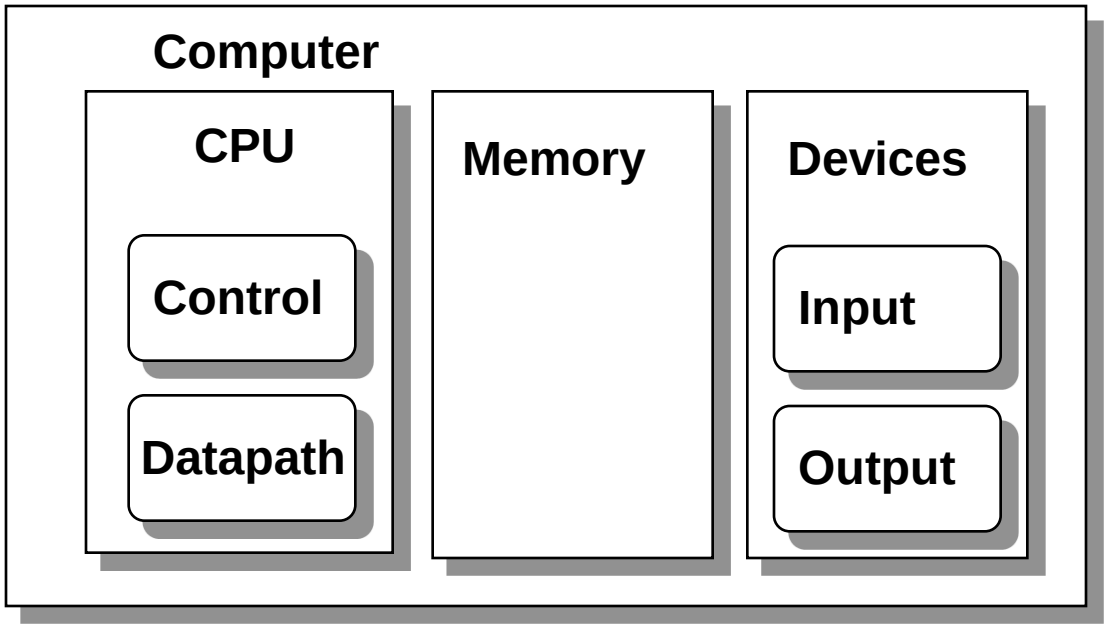
# CS3350B
# Computer Architecture
**Winter 2015**

# Performance Metrics I

Marc Moreno Maza

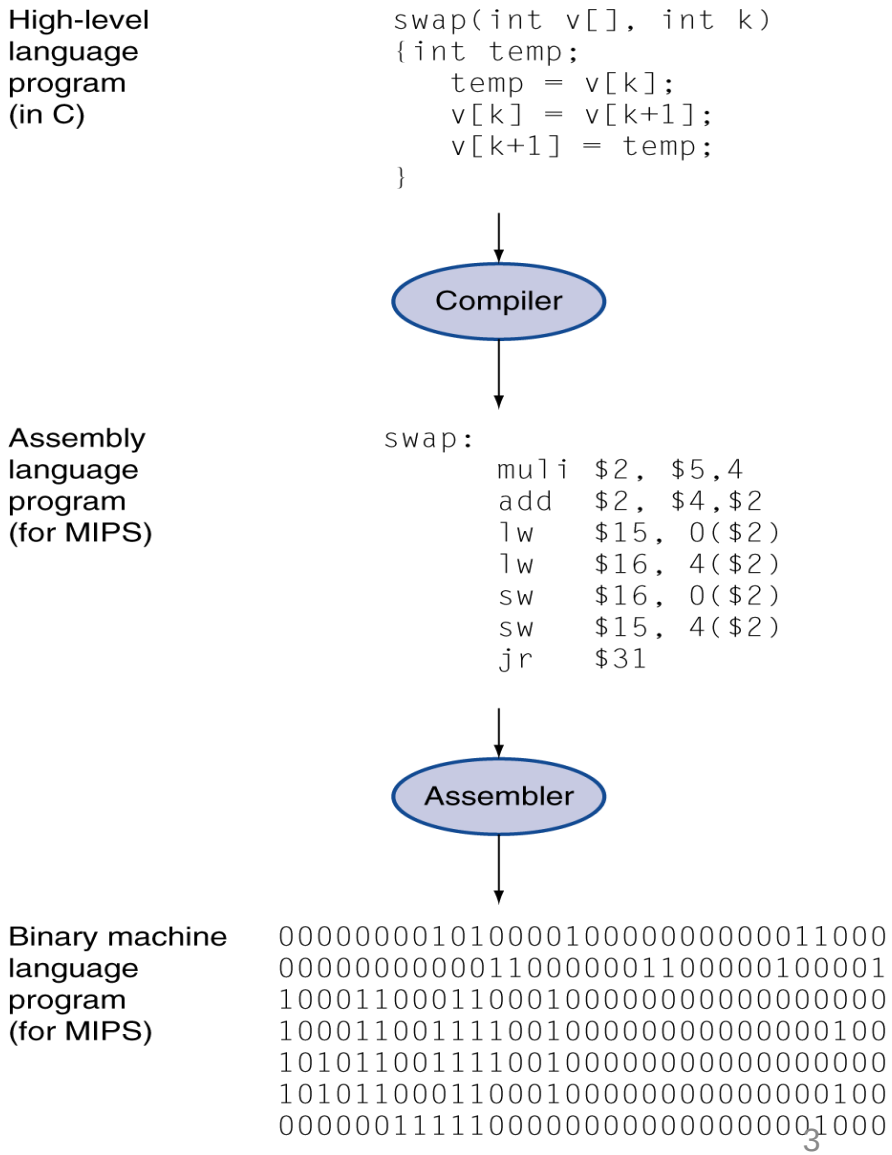
www.csd.uwo.ca/Courses/CS3350b

# Components of a Computer

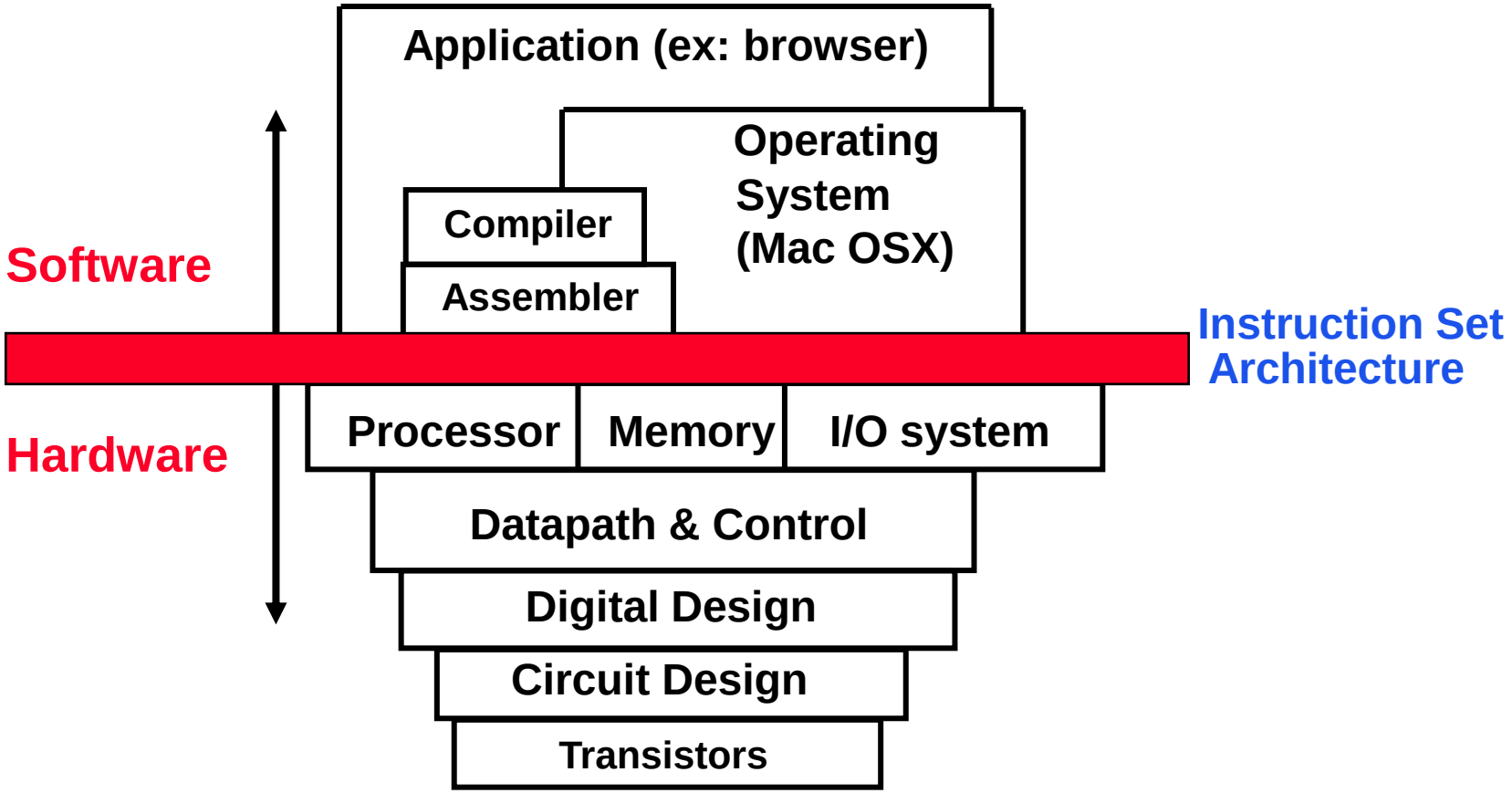


# Levels of Program Code

- **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



# Old School Machine Structures (Layers of Abstraction)



# New-School Machine Structures

*Software* *Hardware*

- Parallel Requests
  - Assigned to computer
  - e.g., Search “Katz”
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words
- Hardware descriptions
  - All gates working in parallel at same time

*Harness Parallelism & Achieve High Performance*



## Eight Great Ideas in Pursuing Performance

- Design for ***Moore's Law***
- Use ***abstraction*** to simplify design
- Make the ***common case fast***
- Performance *via* ***parallelism***
- Performance *via* ***pipelining***
- Performance *via* ***prediction***
- ***Hierarchy*** of memories
- ***Dependability*** *via* redundancy

# Abstractions

- Abstraction helps us deal with complexity
  - Hide lower-level detail
- Instruction set architecture (ISA)
  - The hardware/software interface
- Application binary interface
  - The ISA plus system software interface
- Implementation
  - The details underlying and interface

# Understanding Performance

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

# Performance Metrics

- Purchasing perspective
  - given a collection of machines, which has the
    - best performance ?
    - least cost ?
    - best cost/performance?
- Design perspective
  - faced with design options, which has the
    - best performance improvement ?
    - least cost ?
    - best cost/performance?
- Both require
  - basis for comparison
  - metric for evaluation
- **Our goal is to understand what factors in the architecture contribute to overall system performance and the relative importance (and cost) of these factors**

# CPU Performance

- Normally interested in reducing
  - Response time (aka execution time) – the time between the start and the completion of a task
    - Important to individual users
  - Thus, to maximize performance, need to minimize execution time

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

If X is n times faster than Y, then

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

- And increasing
  - Throughput – the total amount of work done in a given time
    - Important to data center managers
  - Decreasing response time almost always improves throughput

# Performance Factors

- Want to distinguish elapsed time and the time spent on our task
- CPU execution time (CPU time) – time the CPU spends working on a task
  - Does not include time waiting for I/O or running other programs

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

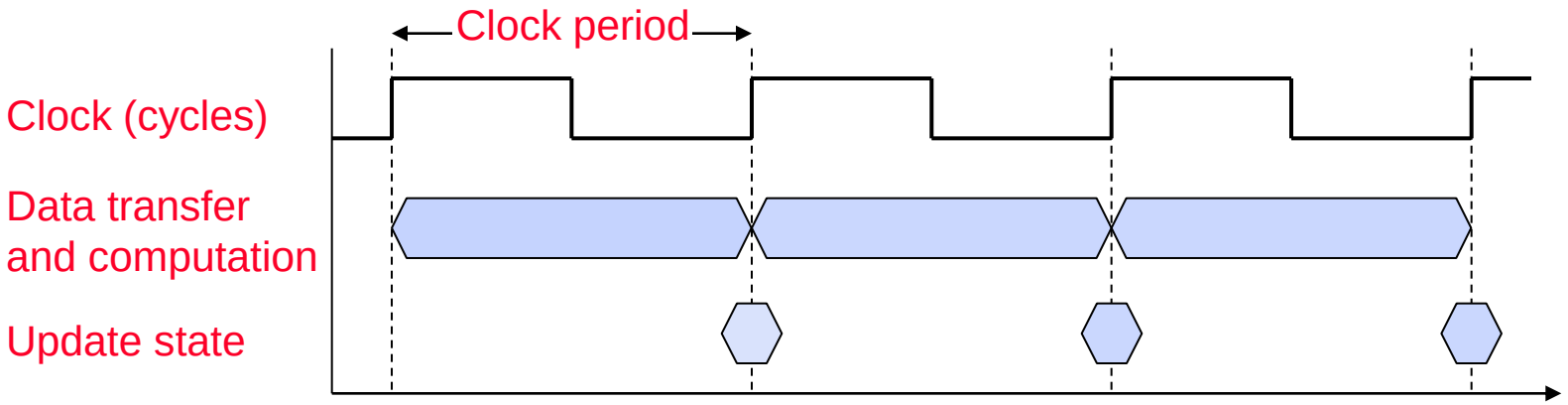
or

$$\text{CPU execution time for a program} = \frac{\text{\# CPU clock cycles for a program}}{\text{clock rate}}$$

- Can improve performance by reducing either the length of the clock cycle or the number of clock cycles required for a program

# CPU Clocking

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




- Clock period (cycle): duration of a clock cycle
  - e.g., 250ps = 0.25ns = $250 \times 10^{-12}$s
- Clock frequency (rate): cycles per second
  - e.g., 3.0GHz = 3000MHz = $3.0 \times 10^{9}$Hz
- CR = 1 / CC

# Clock Cycles per Instruction

- Not all instructions take the same amount of time to execute
  - One way to think about execution time is that it equals the number of instructions executed multiplied by the average time per instruction

| # CPU clock cycles for a program | = | # Instructions for a program | x | Average clock cycles per instruction |
|---|---|---|---|---|

- **Clock cycles per instruction (CPI)** – the average number of clock cycles each instruction takes to execute
  - A way to compare two different implementations of the same ISA

| | CPI for this instruction class | | |
|---|---|---|---|
| | A | B | C |
| CPI | 1 | 2 | 3 |

# Effective CPI

- Computing the overall effective CPI is done by looking at the different types of instructions and their individual cycle counts and averaging

$$\text{Overall effective CPI} = \sum_{i=1}^{n} (CPI_i \times IC_i)$$

  - Where $IC_i$ is the count (percentage) of the number of instructions of class i executed
  - $CPI_i$ is the (average) number of clock cycles per instruction for that instruction class
  - n is the number of instruction classes

- The overall effective CPI varies by instruction mix – a measure of the dynamic frequency of instructions across one or many programs

# THE Performance Equation

- Our basic performance equation is then

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

or

$$\text{CPU time} = \frac{\text{Instruction\_count} \times \text{CPI}}{\text{clock\_rate}}$$

- These equations separate the three key factors that affect performance
  - Can measure the CPU execution time by running the program
  - The clock rate is usually given
  - Can measure overall instruction count by using profilers/ simulators without knowing all of the implementation details
  - CPI varies by instruction type and ISA implementation for which we must know the implementation details

# Determinates of CPU Performance

CPU time = Instruction_count x CPI x clock_cycle

| | Instruction_ count | CPI | clock_cycle |
|---|---|---|---|
| Algorithm | | | |
| Programming language | | | |
| Compiler | | | |
| ISA | | | |
| Processor organization | | | |
| Technology | | | |

# Determinates of CPU Performance

CPU time = Instruction_count x CPI x clock_cycle

| | Instruction_ count | CPI | clock_cycle |
|---|---|---|---|
| Algorithm | X | X | |
| Programming language | X | X | |
| Compiler | X | X | |
| ISA | X | X | X |
| Processor organization | | X | X |
| Technology | | | X |

# A Simple Example

| Op | Freq | $CPI_i$ | Freq x $CPI_i$ | | | |
|---|---|---|---|---|---|---|
| ALU | 50% | 1 | .5 | .5 | .5 | .25 |
| Load | 20% | 5 | 1.0 | .4 | 1.0 | 1.0 |
| Store | 10% | 3 | .3 | .3 | .3 | .3 |
| Branch | 20% | 2 | .4 | .4 | .2 | .4 |
| | | | $\Sigma$ = 2.2 | 1.6 | 2.0 | 1.95 |

- 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 x IC x CC  so  2.2/1.6  means 37.5% faster

- How does this compare with using branch prediction to shave a cycle off the branch time?

  CPU time new = 2.0 x IC x CC  so  2.2/2.0  means 10% faster

- What if two ALU instructions could be executed at once?

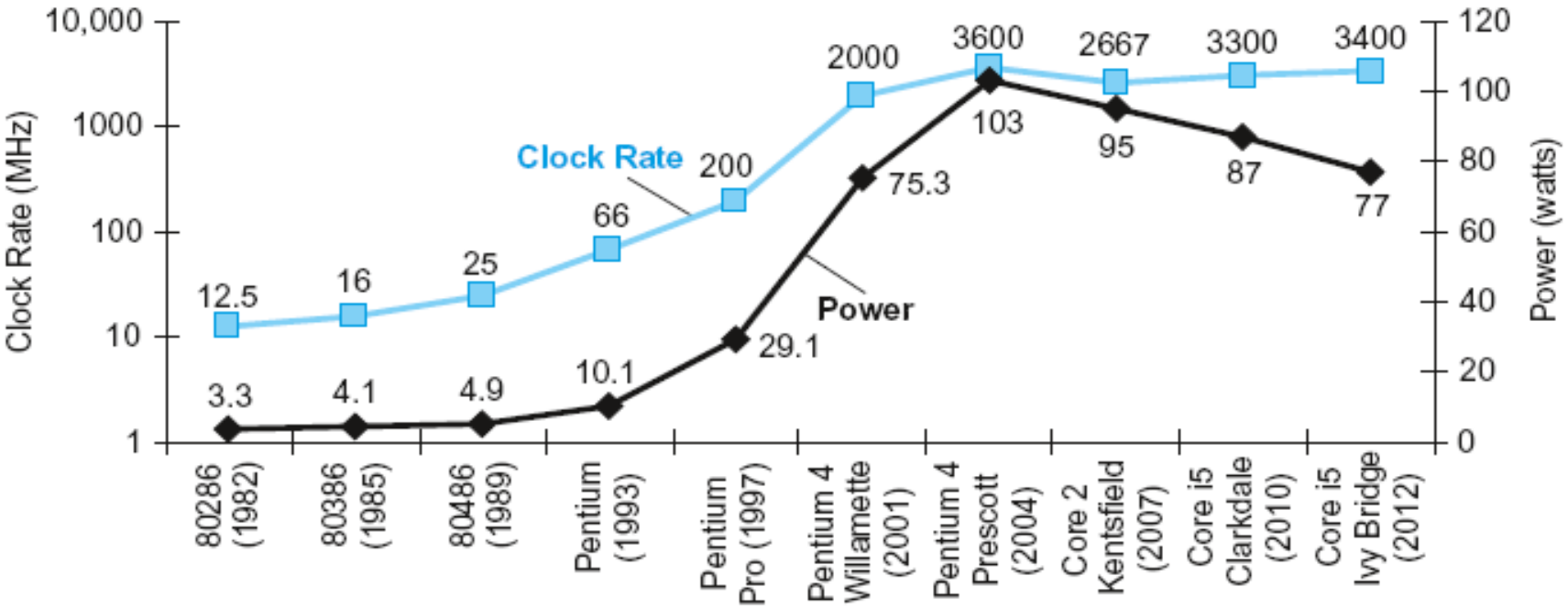
  CPU time new = 1.95 x IC x CC  so  2.2/1.95  means 12.8% faster

# Performance Summary

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

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, $T_c$

# Power Trends




- In complementary metal–oxide–semiconductor (CMOS) integrated circuit technology

$$\text{Power} = \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×30 | 5V → 1V | ×1000

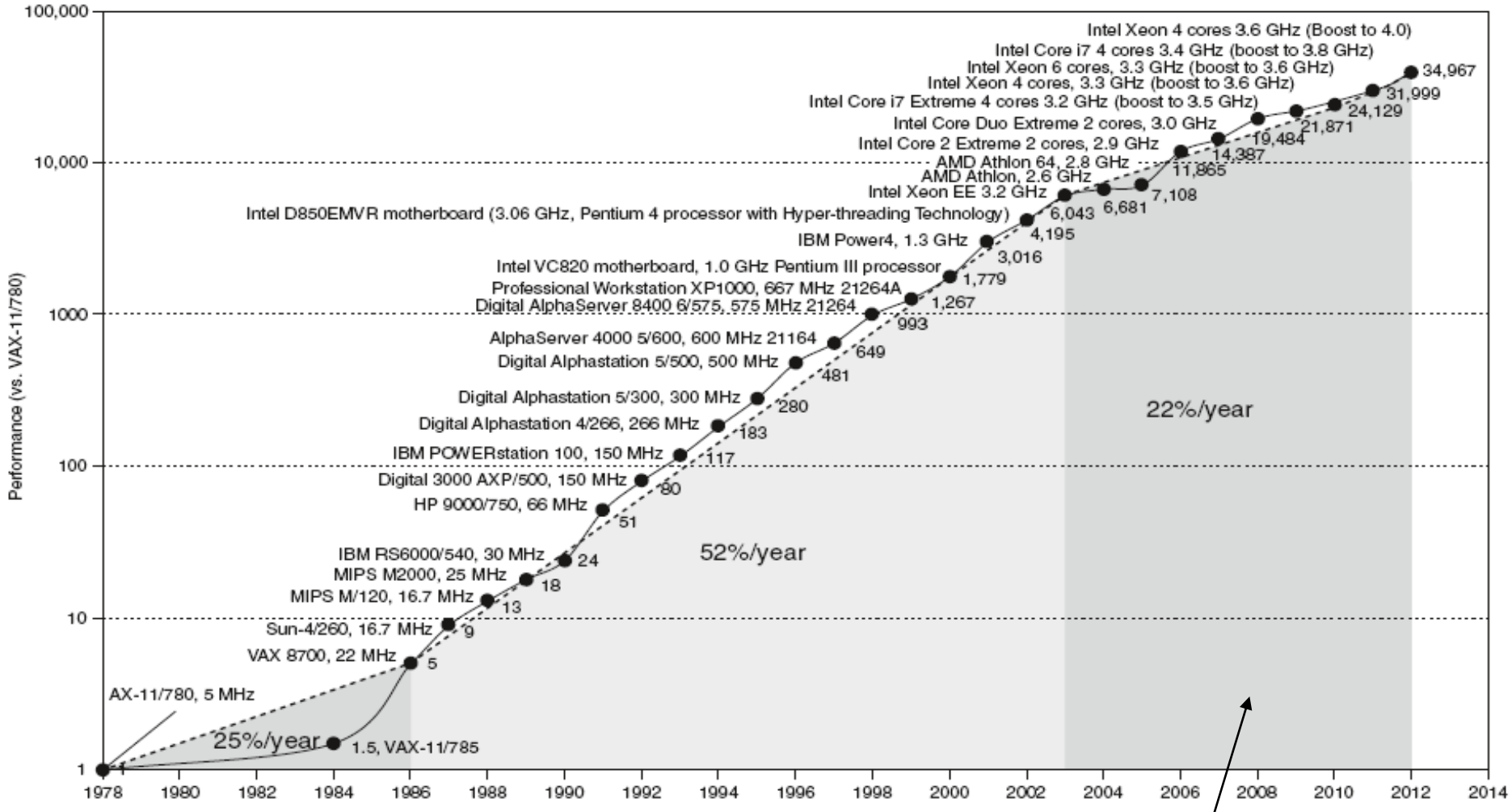
# Reducing Power

- 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
  - We can't reduce voltage further
  - We can't remove more heat
- How else can we improve performance?

# Uniprocessor Performance




Constrained by power, instruction-level parallelism, memory latency

# Multiprocessors

- Multicore microprocessors
  - More than one processor per chip
- Requires explicitly parallel programming
  - Compare with instruction level parallelism
    - Hardware executes multiple instructions at once
    - Hidden from the programmer
  - Hard to do
    - Programming for performance
    - Load balancing
    - Optimizing communication and synchronization

# 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 |

# Profiling Tools

- Many profiling tools
  - gprof (static instrumentation)
  - cachegrind, Dtrace (dynamic instrumentation)
  - **perf** (performance counters)

- perf in **linux-tools**, based on event sampling
  - Keep a list of where “interesting events” (cycle, cache miss, etc) happen
  - **CPU Feature: Counters for hundreds of events**
    - Performance: Cache misses, branch misses, instructions per cycle, …
  - Intel® 64 and IA-32 Architectures Software Developer's Manual: Appendix A lists all counters http://www.intel.com/products/processor/manuals/index.htm
  - perf user guide: http://code.google.com/p/kernel/wiki/PerfUserGuid

# Exercise 1

```
void copymatrix1(int n, int (*src)[n],
                        int (*dst)[n]) {
  int i,j;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      dst[i][j] = src[i][j]; }
```

```
void copymatrix2(int n, int (*src)[n],
                        int (*dst)[n]) {
  int i,j;
  for (j = 0; j < n; j++)
    for (i = 0; i < n; i++)
      dst[i][j] = src[i][j]; }
```

- `copymatrix1 vs copymatrix2`
  - What do they do?
  - What is the difference?
  - Which one performs better? Why?

- `perf stat -e cycles -e cache-misses ./copymatrix1`
  `perf stat -e cycles -e cache-misses ./copymatrix2`
  - What's the output like?
  - How to interpret it?
  - Which program performs better?

# Exercise 2

```
void lower1 (char* s) {
   int i;
   for (i = 0; i < strlen(s); i++)
      if (s[i] >= 'A' && s[i] <= 'Z')
         s[i] -= 'A'-'a';
}
```

```
void lower2 (char* s) {
   int i;
   int n = strlen(s);
   for (i = 0; i < n; i++)
      if (s[i] >= 'A' && s[i] <= 'Z')
         s[i] -= 'A'-'a'; }
```

- `lower1 vs lower2`
  - What do they do?
  - What is the difference?
  - Which one performs better? Why?

- `perf stat -e cycles -e cache-misses ./lower1`
  `perf stat -e cycles -e cache-misses ./lower2`
  - What does the output look like?
  - How to interpret it?
  - Which program performs better?