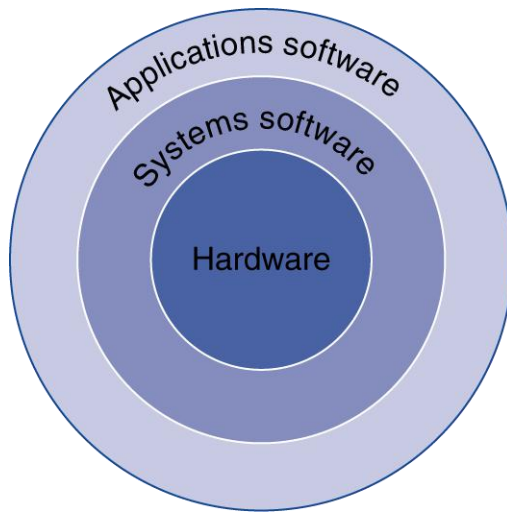


# Between Your Program and Hardware



- 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)

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

Compiler

Assembly  
language  
program  
(for MIPS)

```
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
```

Assembler

Binary machine  
language  
program  
(for MIPS)

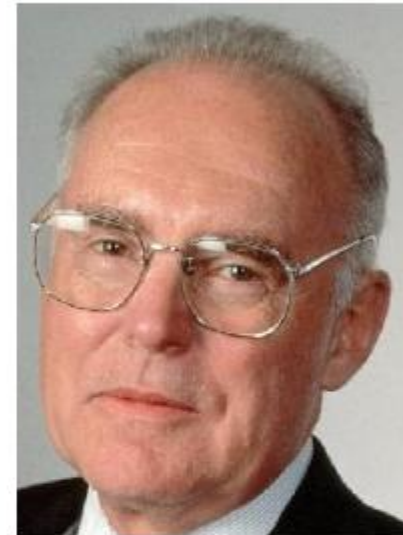
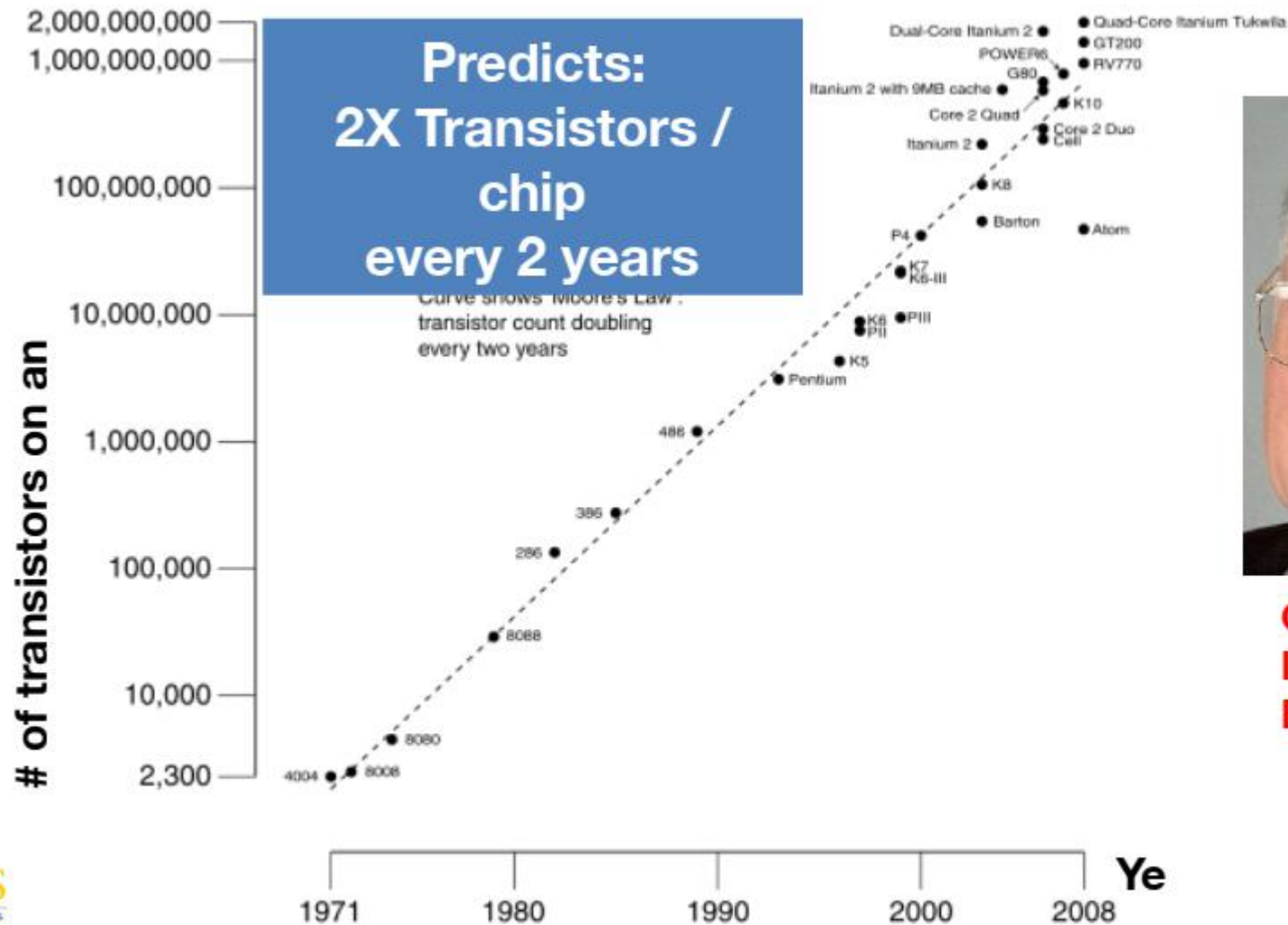
```
000000001010000100000000000011000
000000000000110000001100000100001
100011000110001000000000000000000
100011001111001000000000000000100
101011001111001000000000000000000
101011000110001000000000000000100
00000011111000000000000000001000
```

# Eight Great Ideas

- 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



# Moore's Law

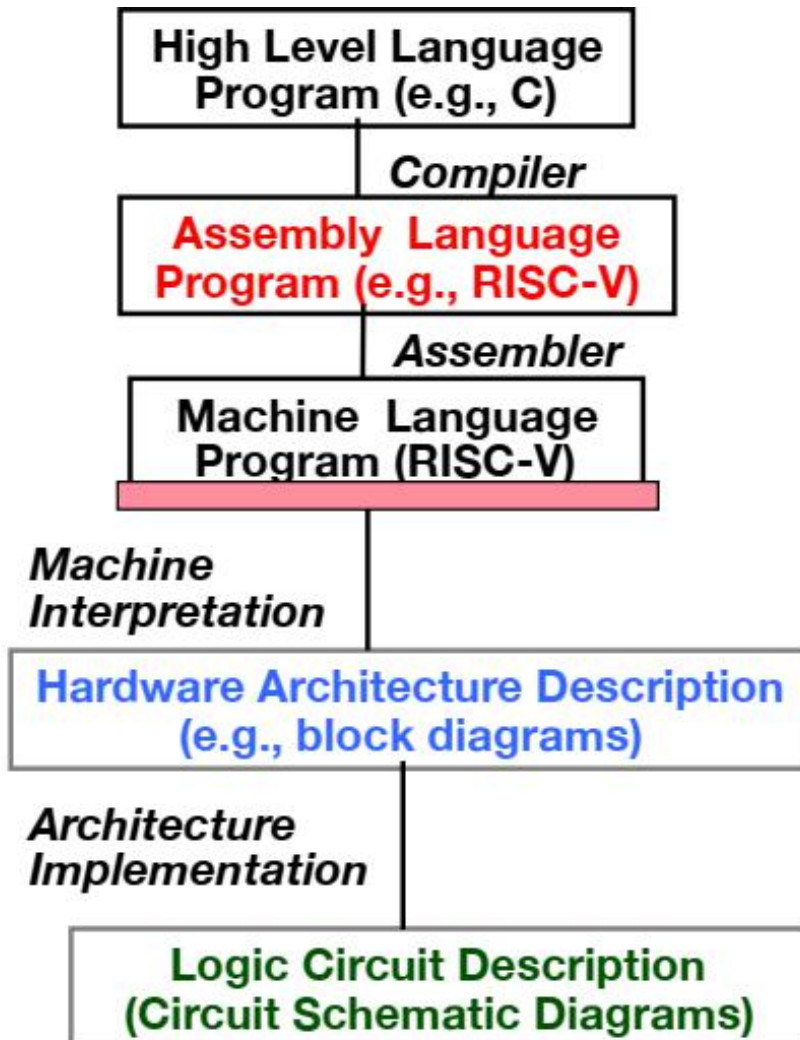


**Gordon Moore**  
**Intel Cofounder**  
**B.S. Cal 1950!**

# Post Moore's Law

- Moore's Law meant that the cost of transistors scaled down as technology scaled to smaller and smaller feature sizes.
- And the resulting transistors resulted in increased single-task performance
- But single-task performance improvements hit a brick wall years ago...
- And now the newest, smallest fabrication processes  $< 14\text{nm}$ , might have greater cost/transistor!!!! So, why shrink????

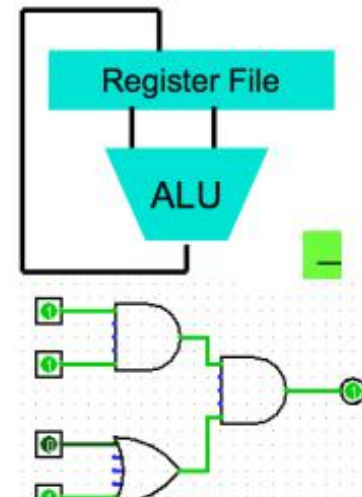
# Abstraction



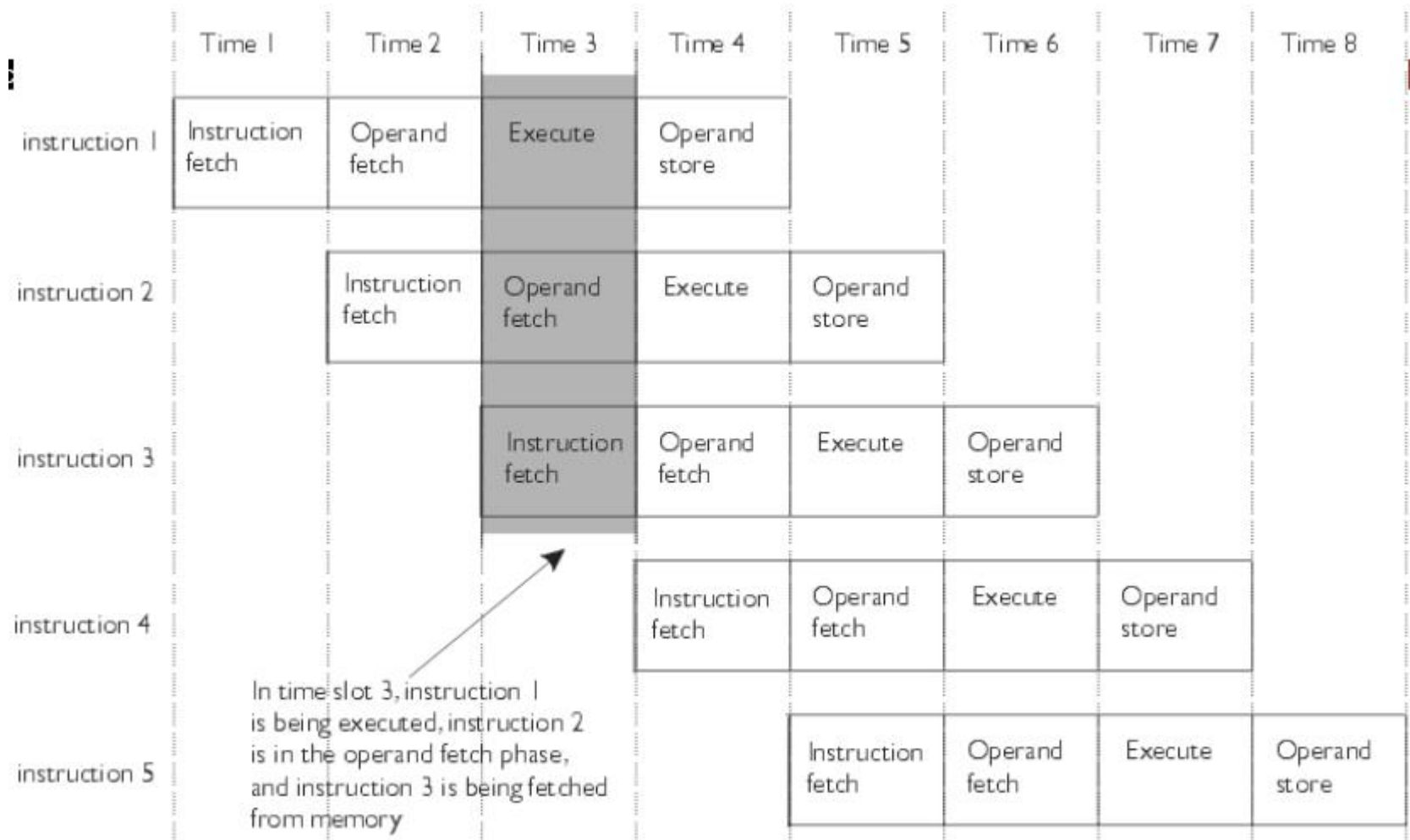
```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Anything can be represented  
as a *number*,  
i.e., data or instructions

```
0000 1001 1100 0110 1010
1010 1111 0101 1000 0000
1100 0110 1010 1111 0101
0101 1000 0000 1001 1100
```



# Pipeline





# Parallelism

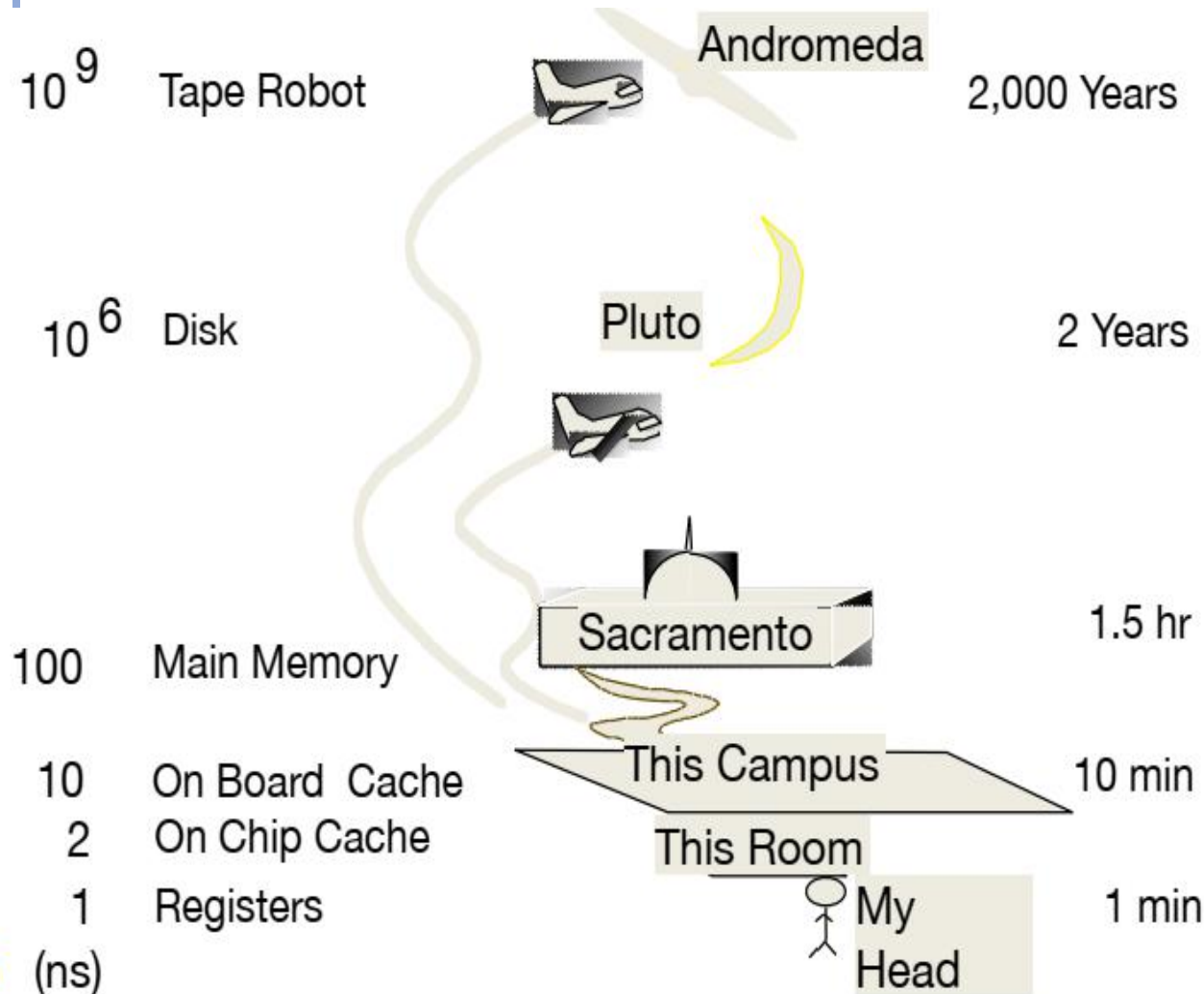
- Raspberry Pi 4 B+
  - Quad core processor at 1.5 GHz
    - Each core is 3-issue, out-of-order superscalar
  - Plus GPU, 1-4 GB RAM, 2x USB3, 2x USB2, Gigabit Ethernet, 2x HDMI...
  - \$35-55
    - Nick is working on a board to turn one of these to power a fully autonomous, vision-guided drone
- Compare with a Cray-1 from 1975:
  - 8 MB RAM, 80 MHz processor, 300MB storage, \$5M+
- Or modern high end servers:
  - You can get a 2u server which supports 4 processors
    - And each processor can have 20+ cores, so 80 processor cores!





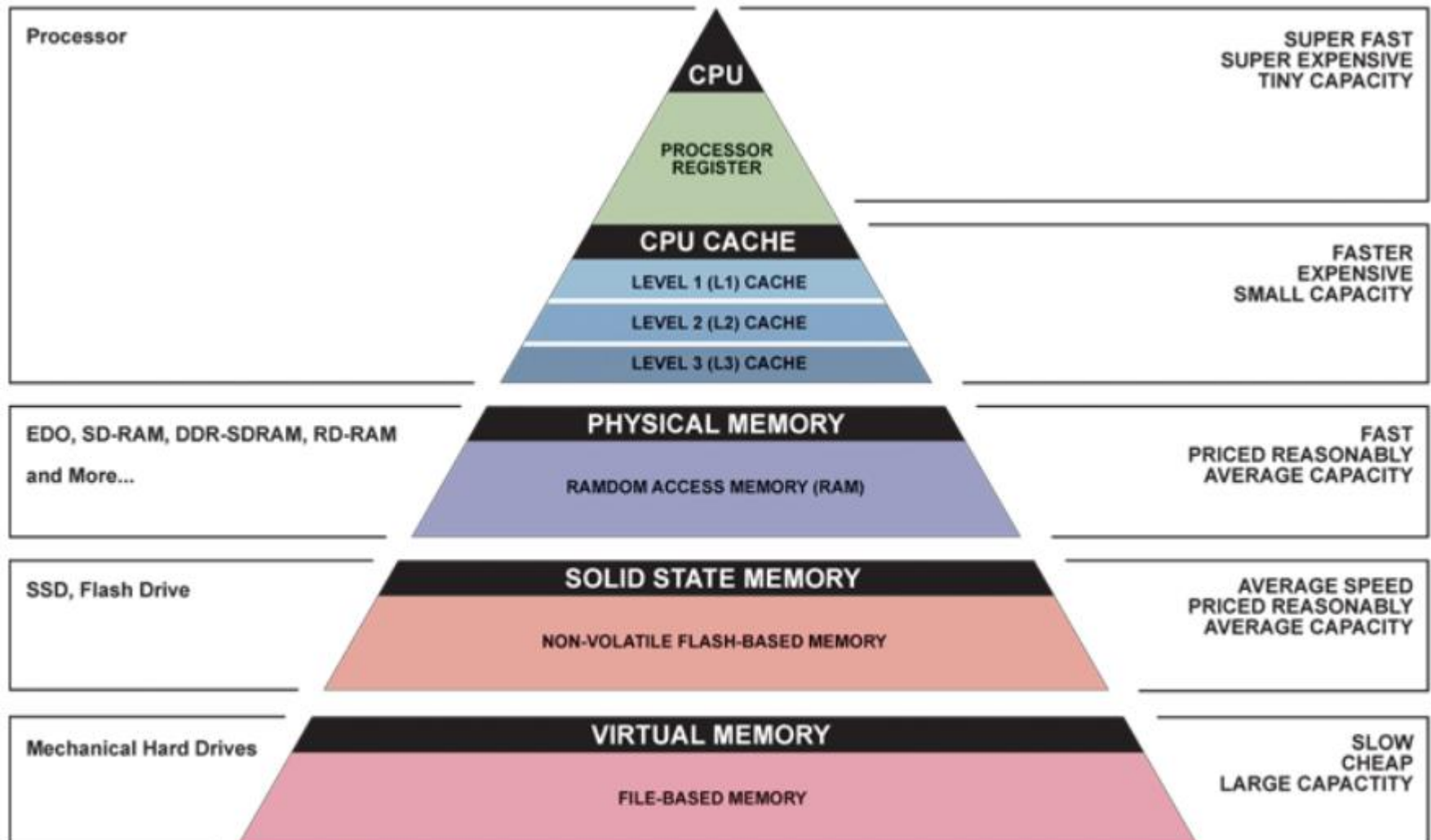
# Jim Gray's Storage Latency Analogy

- How far away is the Data?



**Jim Gray**  
**Turing Award**  
**B.S. Cal 1966**  
**Ph.D. Cal 1969!**

# Memory Hierarchy



# Fails Happen, so?

- 4 disks/server, 50,000 servers
- Failure rate of disks: 2% to 10% / year
  - Assume 4% annual failure rate
- On average, how often does a disk fail?
  - a) 1 / month
  - b) 1 / week
  - c) 1 / day
  - d) 1 / hour

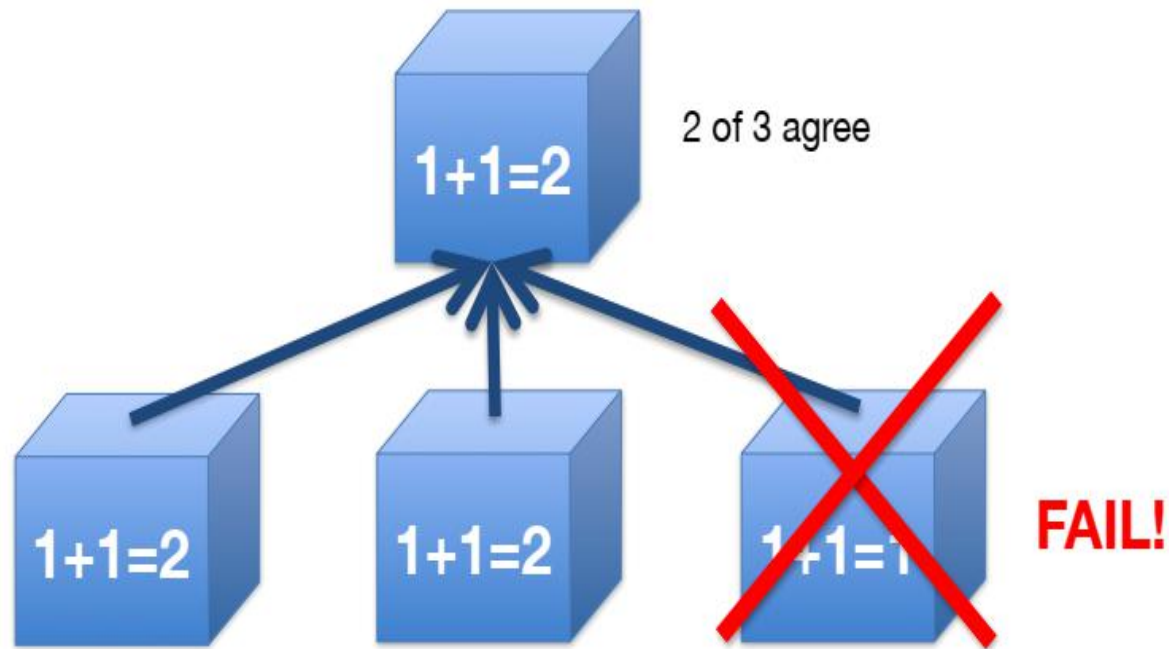
$50,000 \times 4 = 200,000$  disks

$200,000 \times 4\% = 8000$  disks fail

$365 \text{ days} \times 24 \text{ hours} = 8760$  hours

# Reliability via Redundancy

Redundancy so that a failing piece doesn't make the whole system fail



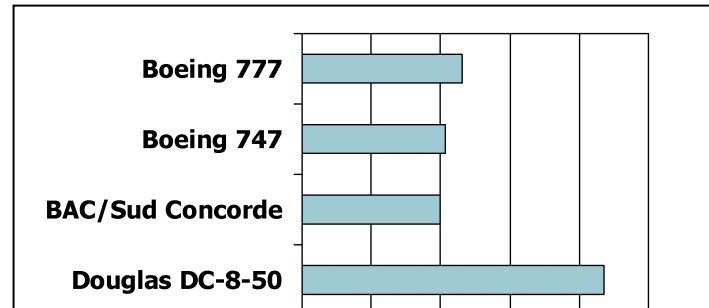
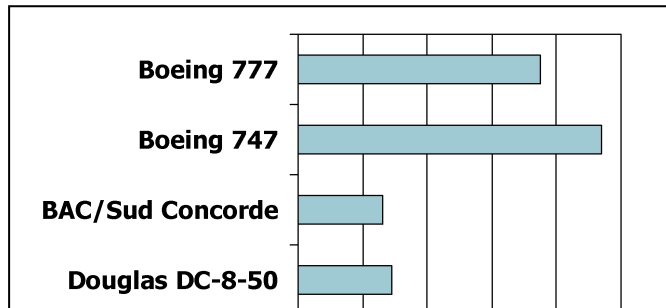
Increasing transistor density reduces the cost of redundancy

## Lecture 2

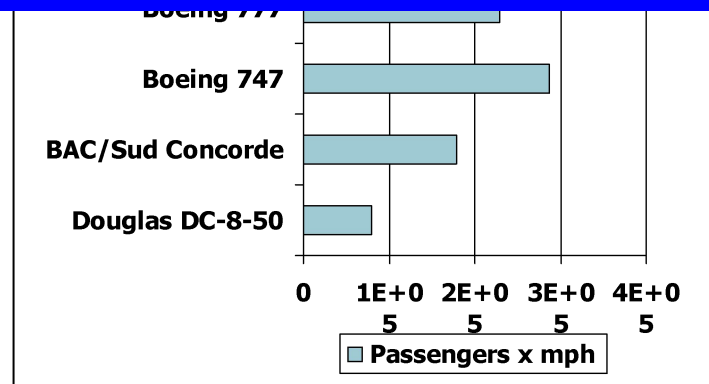
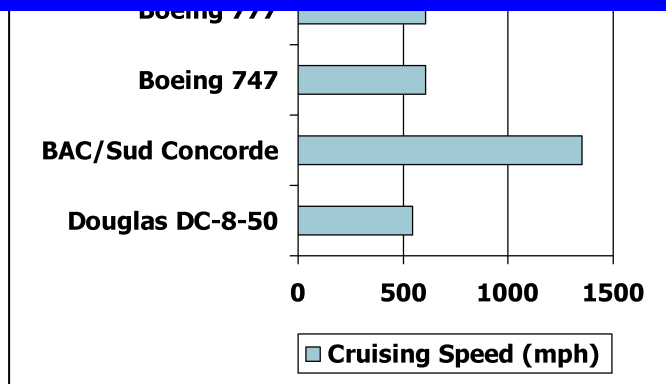
### **Performance**

# Defining Performance

- Which airplane has the best performance?



To evaluate the performance, we must define the **metric** first!





# Response Time and Throughput

- Response time
  - ◆ How long it takes to do a task
- Throughput
  - ◆ Total work done per unit time
    - e.g., tasks/transactions/... per hour
- How are response time and throughput affected by
  - ◆ Replacing the processor with a faster version?
  - ◆ Adding more processors?
- We'll focus on response time for now...

# Relative Performance

- Define Performance =  $1/\text{Execution Time}$

- “X is  $n$  time as fast as 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 of B / Execution Time of A  
= 15s / 10s = 1.5

- ◆ So A is 1.5 times as fast as B

- Increase performance = decrease execution time

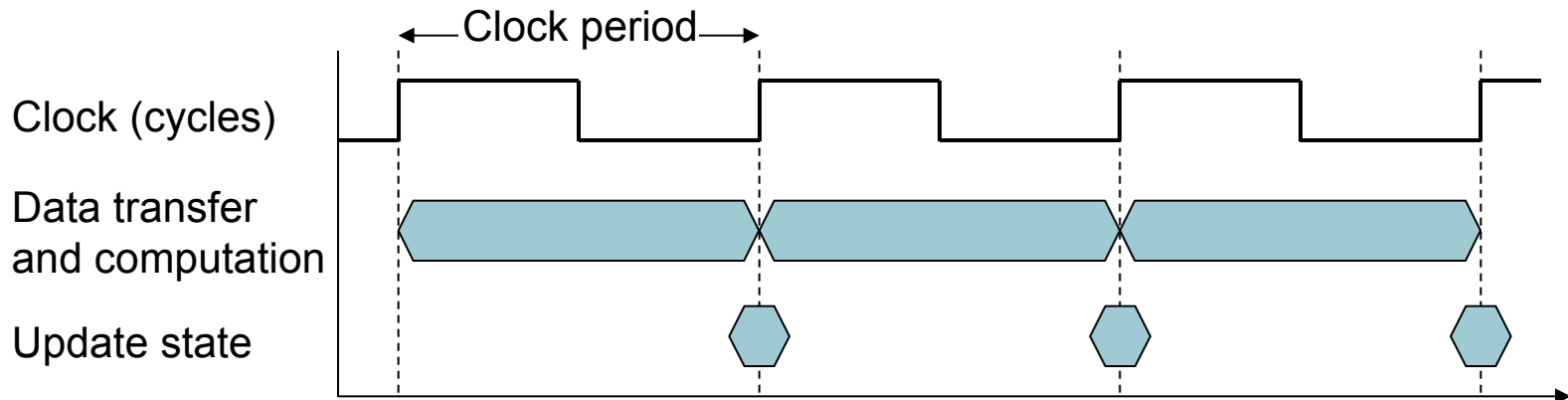
→ “**improve**” performance/execution time.

# Measuring Execution Time

- Elapsed time
  - ◆ Total response 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
  - ◆ Comprises user CPU time and system CPU time
  - ◆ Different programs are affected differently by CPU and system performance

# CPU Clocking

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



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

# CPU Time

CPU Time = No. of Clock Cycles × Clock Period

$$= \frac{\text{No. of Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - ◆ Reducing number of clock cycles (cycle count)
  - ◆ Increasing clock rate
  - ◆ Hardware designer must often trade off clock rate against cycle count

# CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - ◆ Aim for 6s CPU time
  - ◆ Can do faster clock, but causes  $1.2 \times$  number (#) of clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{No. of Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{No. of Clock Cycles}_A}{6s}$$

$$\begin{aligned}\text{No. of Clock Cycles}_A &= \text{CPU Time}_A \times \text{No. of Clock Rate}_A \\ &= 10s \times 2\text{GHz} = 20 \times 10^9\end{aligned}$$

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



# Instruction Count and CPI

*No. of Clock Cycles* = Instruction Count  $\times$  Cycles per Instruction (CPI)

CPU Time = Instruction Count  $\times$  CPI  $\times$  Clock Period

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

- 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

# CPI Example

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

$$\begin{aligned}\text{CPU Time}_A &= \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A \\ &= 1 \times 2.0 \times 250\text{ps} = 1 \times 500\text{ps} \end{aligned}$$

A is faster...

$$\begin{aligned}\text{CPU Time}_B &= \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B \\ &= 1 \times 1.2 \times 500\text{ps} = 1 \times 600\text{ps} \end{aligned}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{1 \times 600\text{ps}}{1 \times 500\text{ps}} = 1.2$$

...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

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

Relative frequency

# CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C. IC is short for “instruction count”.

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  
 $= 2 \times 1 + 1 \times 2 + 2 \times 3$   
 $= 10$
- ◆ Avg. CPI =  $10/5 = 2.0$

- Sequence 2: IC = 6

- ◆ Clock Cycles  
 $= 4 \times 1 + 1 \times 2 + 1 \times 3$   
 $= 9$
- ◆ Avg. CPI =  $9/6 = 1.5$

# Performance Summary

## The BIG Picture

$$\begin{aligned}\text{CPU Time} &= \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}} \\ &= \text{IC} \times \text{CPI} \times T_c\end{aligned}$$

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

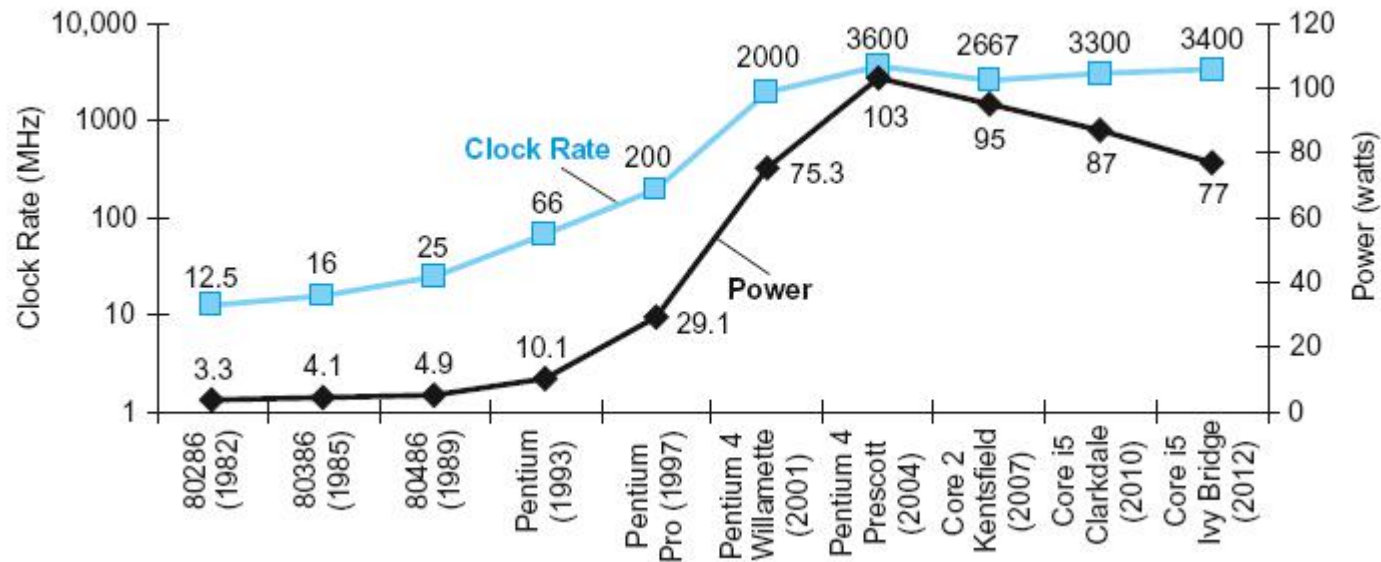
# Energy Consumption of a chip

- Energy consumption = dynamic energy + static energy
  - ◆ Dynamic energy (energy spent when transistors switch from  $0 \rightarrow 1$   $1 \rightarrow 0$ ) is primary
  - ◆ Static energy is the energy cost when no transistor switches
- Energy for  $0 \rightarrow 1 \rightarrow 0$ :  $\text{Energy} \propto \text{Capacitive load} \times \text{Voltage}^2$
- Energy for  $0 \rightarrow 1$  or  $1 \rightarrow 0$ :  $\text{Energy} \propto 1/2 \times \text{Capacitive load} \times \text{Voltage}^2$
- Energy per second (power):

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



# Power Trends



- In CMOS IC technology

$$\text{Power} \propto \frac{1}{2} \text{Capacitive load} \times \text{Voltage}^2 \times \text{Frequency}$$

×23

5V → 1.5V

×270

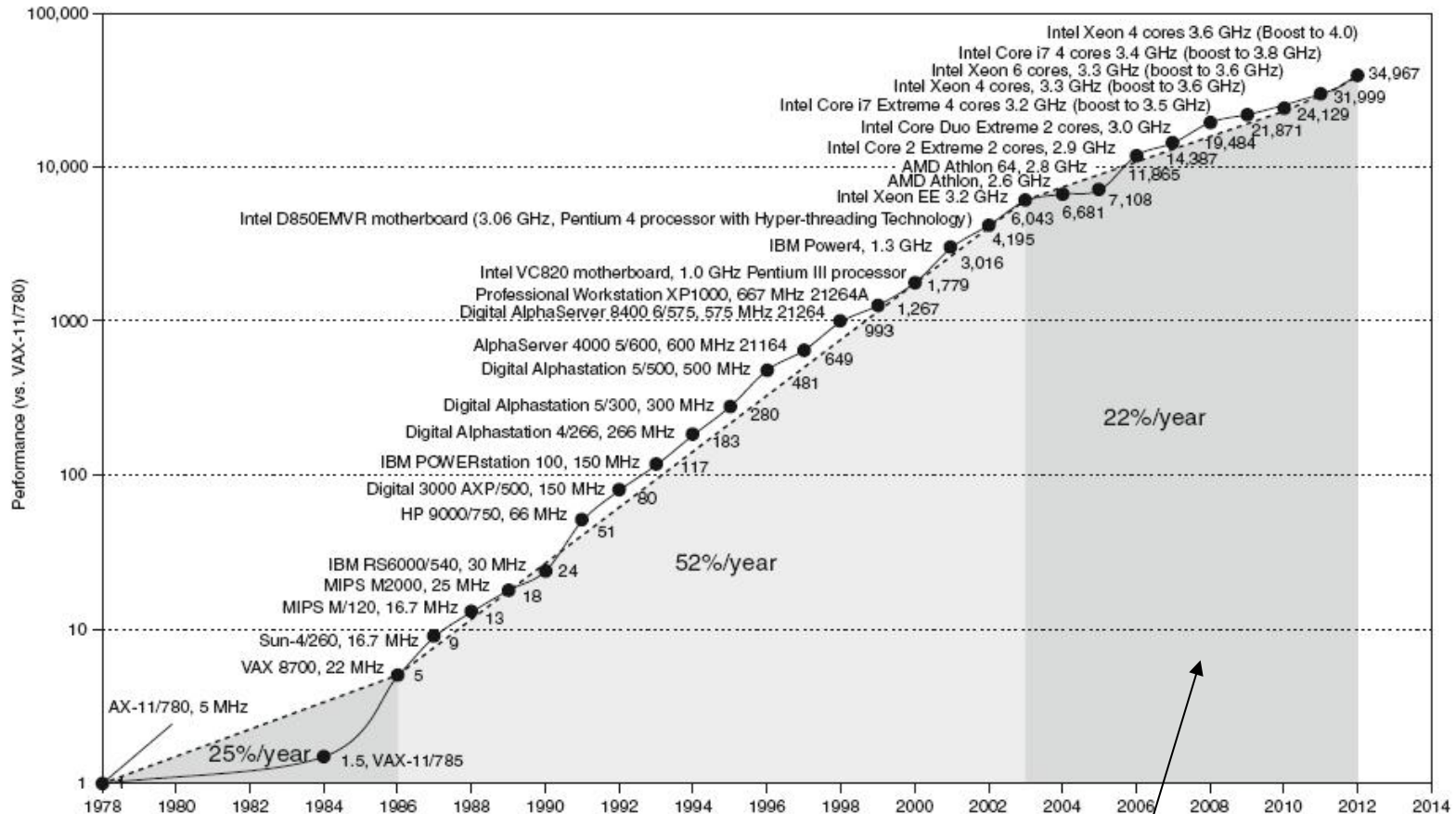
# Reducing Power

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

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

- 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

# Benchmark Suites

- Each vendor announces a SPEC (Standard Performance Evaluation Cooperative) rating for their system
  - ◆ a measure of execution time for a fixed collection of programs
  - ◆ is a function of a specific CPU, memory system, IO system, operating system, compiler
  - ◆ enables easy comparison of different systems
  
- The key is coming up with a collection of relevant programs

# SPEC CPU Benchmark

- Programs used to measure performance
  - ◆ Supposedly typical of actual workload
- Standard Performance Evaluation Cooperative (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
  - ◆ Normalized 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 <sup>9</sup>	CPI	Clock cycle time (seconds x 10 <sup>-9</sup> )	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

# SPEC Power Benchmark

- Power consumption of the server at different workload levels
  - ◆ Performance: ssj\_ops (server side Java operations per second)
  - ◆ Power: Watts (Joules/sec)

$$\text{Overall ssj_ops per Watt} = \left( \sum_{i=0}^{10} \text{ssj\_ops}_i \right) / \left( \sum_{i=0}^{10} \text{power}_i \right)$$

# SPECpower\_ssj2008 for Xeon X5650

Target Load %	Performance (ssj_ops)	Average Power (Watts)
100%	865,618	258
90%	786,688	242
80%	698,051	224
70%	607,826	204
60%	521,391	185
50%	436,757	170
40%	345,919	157
30%	262,071	146
20%	176,061	135
10%	86,784	121
0%	0	80
Overall Sum	4,787,166	1,922
$\Sigma \text{ssj\_ops} / \Sigma \text{power} =$		2,490

# Amdahl's Law

- Architecture design is very **bottleneck-driven** – make the common case fast, do not waste resources on a component that has little impact on overall performance/power
- Amdahl's Law: performance improvements through an enhancement is limited by the **fraction of time** the enhancement comes into play
- Example: multiply accounts for 80s/100s
  - ◆ How much improvement in multiply performance to get  $5\times$  overall?

$$20 = \frac{80}{n} + 20$$

■ Can't be done!

- Corollary: make the common case fast

# Fallacy: Low Power at Idle

- Look back at i7 power benchmark
  - ◆ At 100% load: 258W
  - ◆ At 50% load: 170W (66%)
  - ◆ At 10% load: 121W (47%)
- Google data center
  - ◆ Mostly operates at 10% – 50% load
  - ◆ At 100% load less than 1% of the time
- Consider designing processors to make power proportional to load

# Concluding Remarks

- Knowledge of hardware improves software quality:
  - ◆ compilers, OS, threaded programs, memory management
- Important trends:
  - ◆ growing transistors
  - ◆ move to multi-core
  - ◆ slowing rate of performance improvement
  - ◆ power/thermal constraints
- Reasoning about performance: clock speeds, CPI, benchmark suites, performance equations
- Next: assembly instructions