Chapter 1: Computer Abstractions and Technology

1.6 – 1.7: Performance and power

ITSC 3181 Introduction to Computer Architecture
https://passlab.github.io/ITSC3181/

Department of Computer Science
Yonghong Yan
yyan7@uncc.edu
https://passlab.github.io/yanyh/

Lectures for Chapter 1 and C Basics
Computer Abstractions and Technology

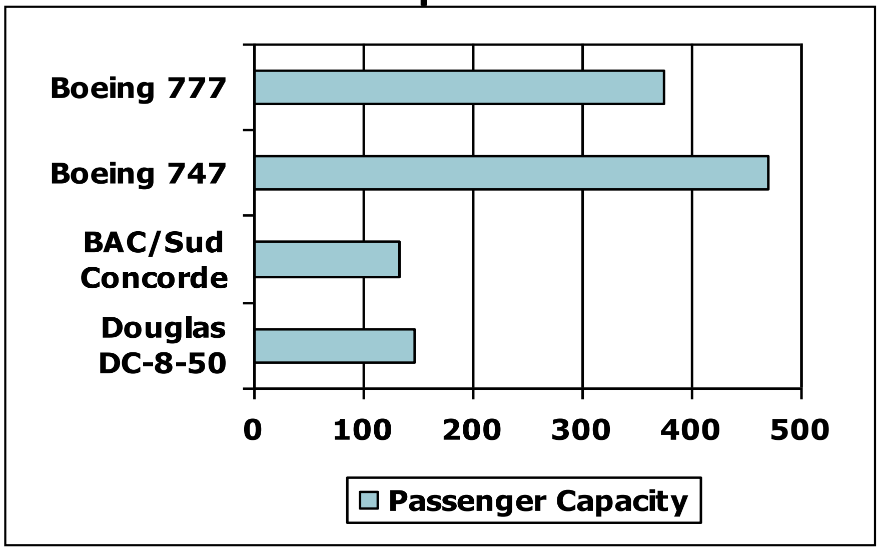
- **Lecture 01: Chapter 1**
  - **1.1 – 1.4: Introduction, great ideas, Moore's law, abstraction, computer components, and program execution**

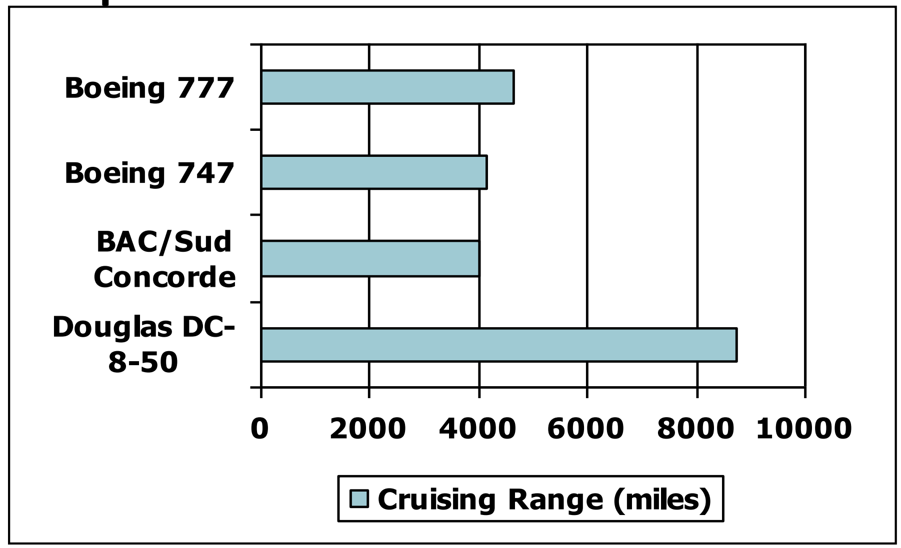
☛ **Lecture 02: Chapter 1 and Memory/Binary System**
  - **1.6 – 1.7: Performance, power and technology trends**
  - **Memory and Binary Systems**

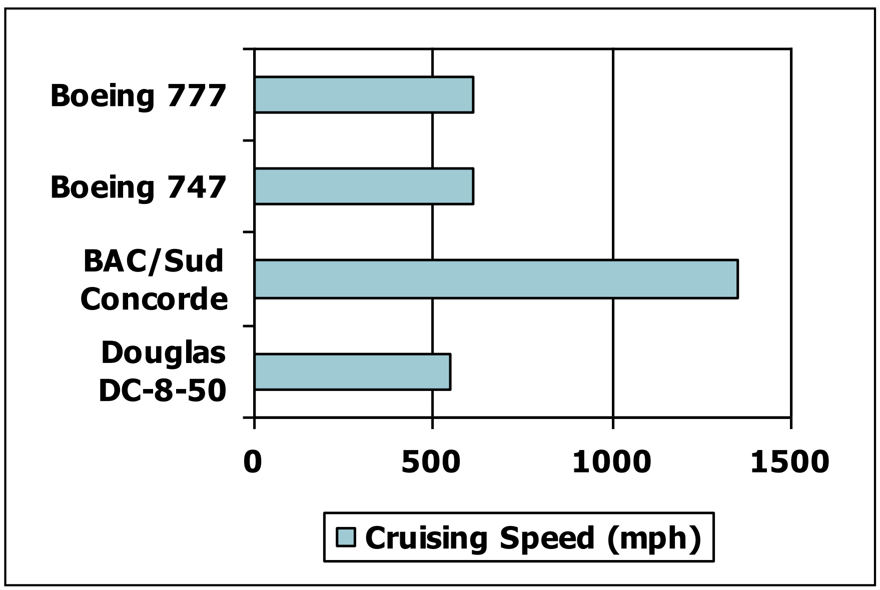
- **Lecture 03: C Basics**
- **Lecture 03/4: Number System, Compilation, Assembly, Linking and Program Execution**
- **Lecture 05:**
  - **1.8 - 1.9: Multiprocessing and benchmarking**

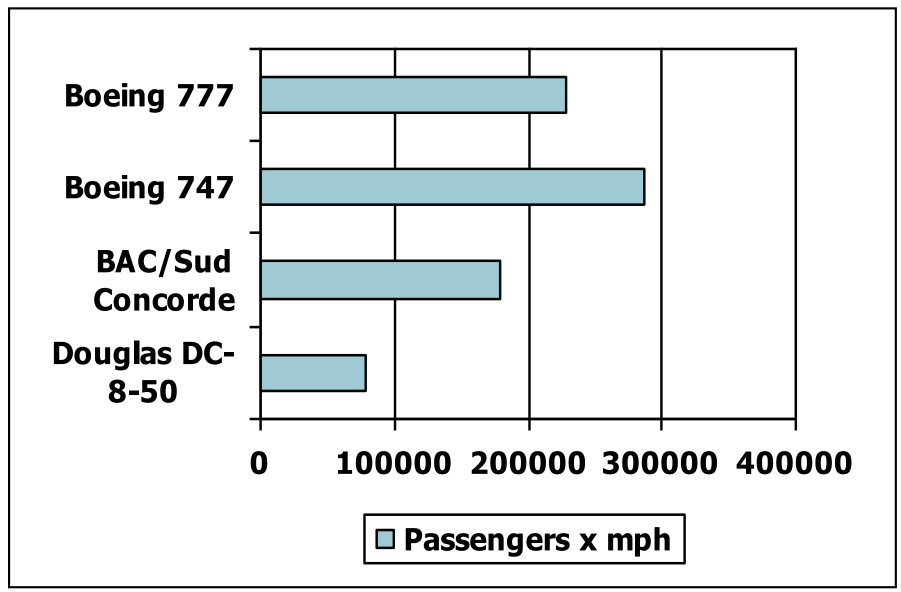
Defining Performance

- Which airplane has the best performance?









Response Time and Throughput

- Response time ←→ Latency
  - How long it takes to do a task
- Throughput ←→ Bandwidth
  - 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/Execution Time
- "X is n time faster than Y", i.e. speedup

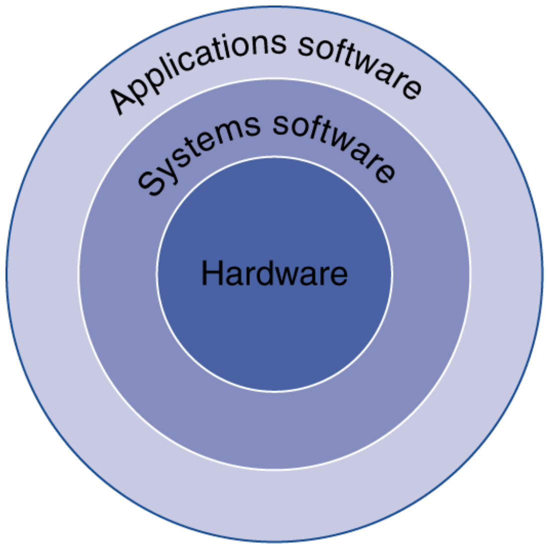
$$\text{Performance}_X / \text{Performance}_Y = \text{Execution time}_Y / \text{Execution time}_X = n$$

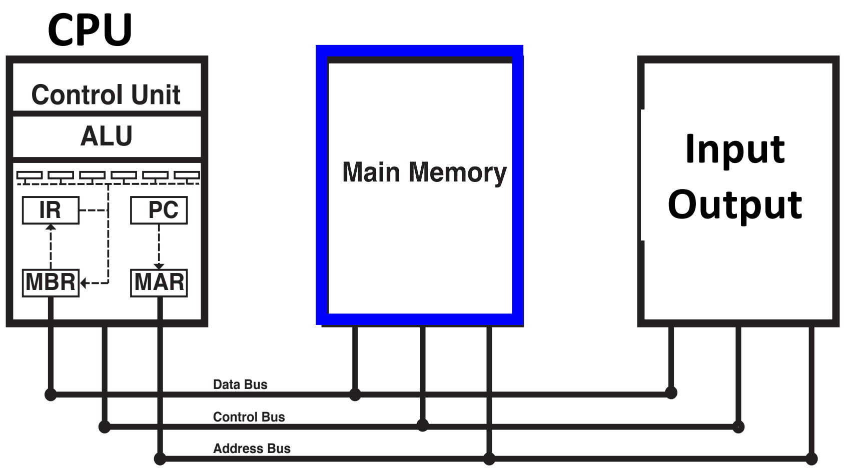
- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time_B / Execution Time_A = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

Below Your Program

Program Performance is impacted by many things

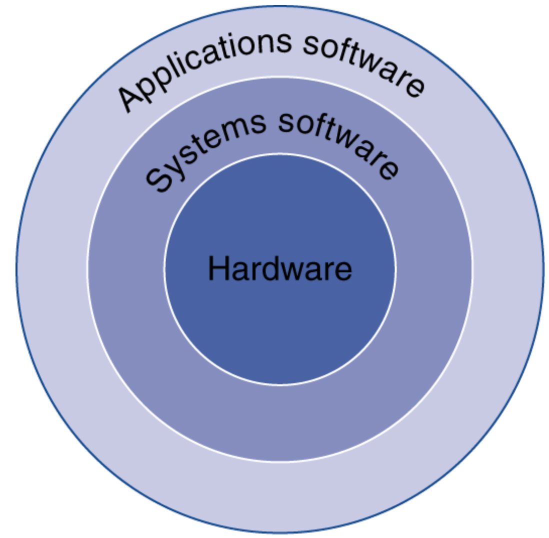
- Program, i.e. Application software
  - Written in high-level language
- System software
  - Compiler: translates HLL code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - Processor, memory, I/O controllers





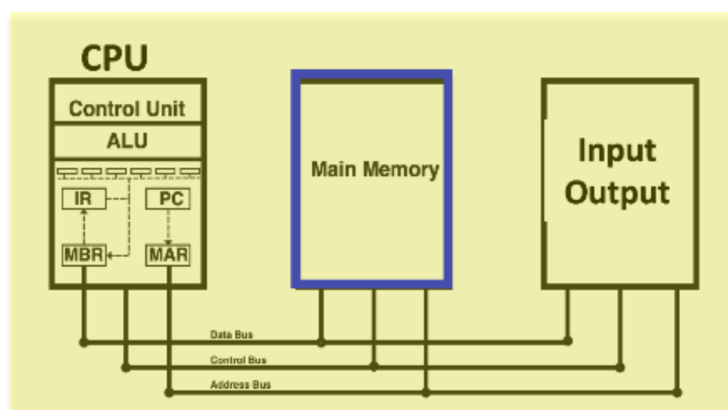
Measuring Execution Time 1/3

- Wall clock time, response time, real time
  - Total response time, including all aspects
    - **CPU Time + I/O + OS overhead + idle time**
    - **printf consume OS/system and I/O time**
- Execution time (time cmd from terminal)

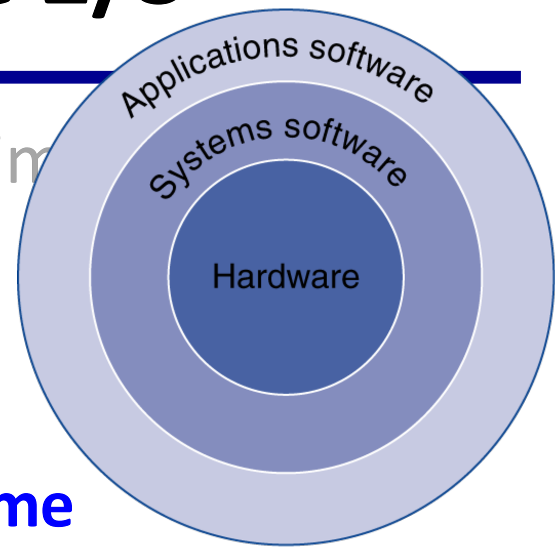



```
[yanyh@fornax ~]$ time ./sum 10000000
=====================================================================
        Sum 10000000 numbers
---------------------------------------------------------------------
Performance:                Runtime (ms)     MFLOPS
---------------------------------------------------------------------
Sum:                        24.999857       800.004578

real        0m0.200s
user        0m0.179s
sys         0m0.020s
```

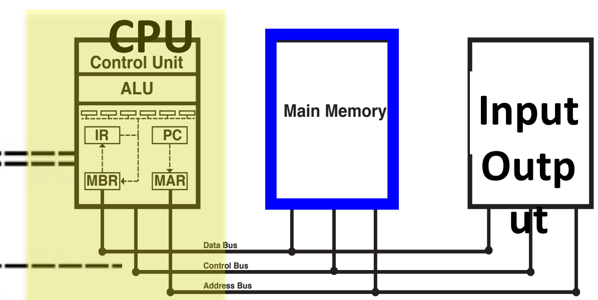
https://passlab.github.io/ITSC3181/exercises/sum/sum_full.c



Measuring Execution Time 2/3



- Wall clock time, response time, real time (tim
- CPU time
  - Time spent processing a given job
    - Not including I/O time, other jobs' shares
  - Comprises **user CPU time** and **system CPU time**
  - Different programs are affected differently by CPU and system
  - "time" command in Linux




```
[yanyh@fornax ~]$ time ./sum 10000000
======================================================================
        Sum 10000000 numbers
----------------------------------------------------------------------
Performance:                Runtime (ms)      MFLOPS
----------------------------------------------------------------------
Sum:                        24.999857        800.004578

real    0m0.200s
user    0m0.179s
sys     0m0.020s
```

Understanding time command output

- **Real** is wall clock time - time from start to finish of the call. This is all elapsed time including time slices used by other processes and time the process spends blocked (for example if it is waiting for I/O to complete).
- **User** is the amount of CPU time spent in user-mode code (outside the kernel) *within* the process. This is only actual CPU time used in executing the process. Other processes and time the process spends blocked do not count towards this figure.
- **Sys** is the amount of CPU time spent in the kernel within the process. This means executing CPU time spent in system calls *within the kernel,* as opposed to library code, which is still running in user-space. Like 'user', this is only CPU time used by the process. See below for a brief description of kernel mode (also known as 'supervisor' mode) and the system call mechanism.

```
[yanyh@fornax ~]$ time ./sum 10000000
====================================================
        Sum 10000000 numbers
----------------------------------------------------
Performance:                Runtime (ms)     MFLOPS
----------------------------------------------------
Sum:                        24.999857       800.004578

real    0m0.200s
user    0m0.179s
sys     0m0.020s
```

Measuring Execution Time of Specific Operations 3/3

- Elapsed time of the sum function: use timer

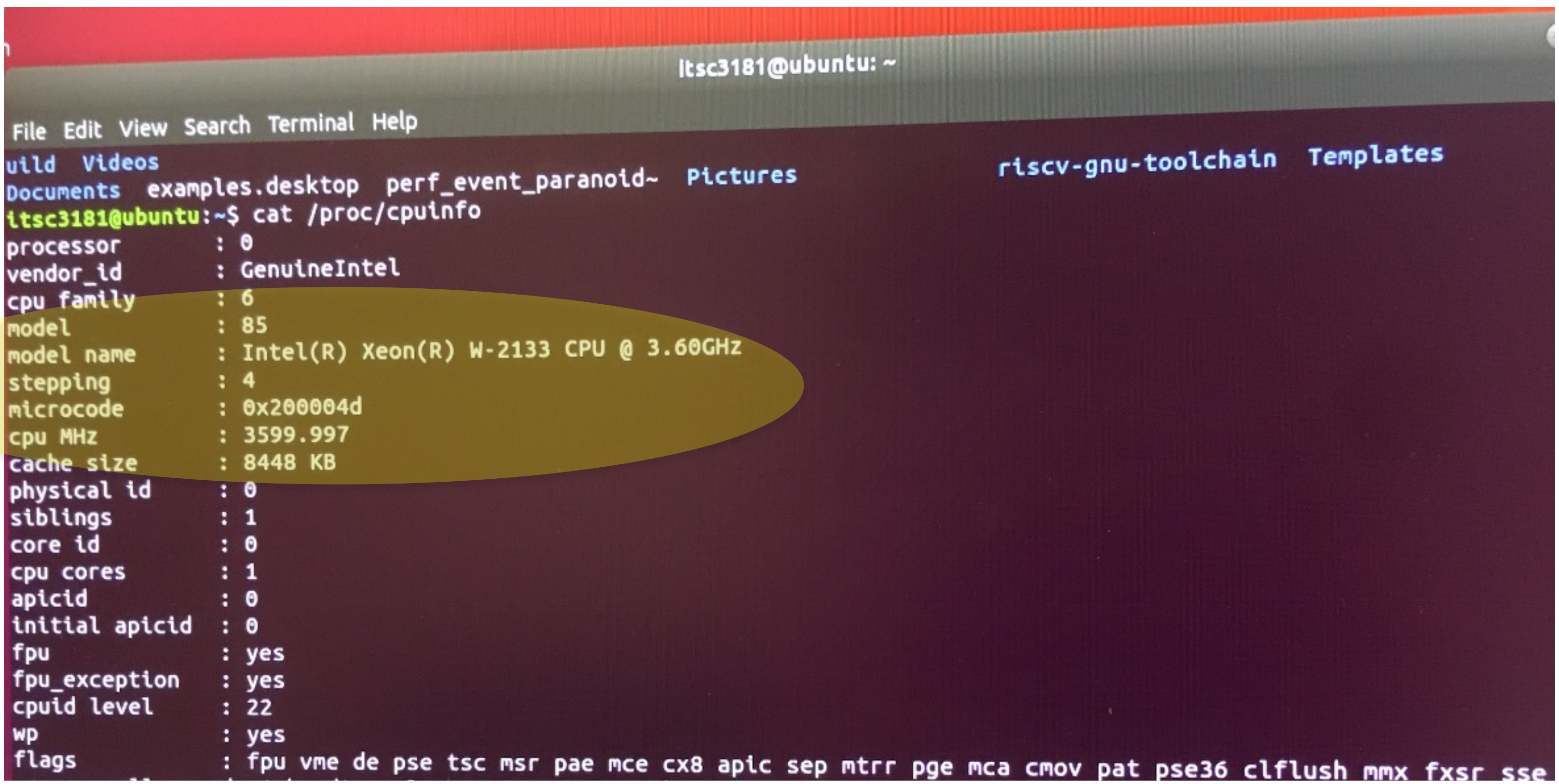
```
elapsed = read_timer();
REAL result = sum(N, X, a);
elapsed = (read_timer() - elapsed);
```

https://passlab.github.io/ITSC3181/exercises/sum/sum_full.c

```
[yanyh@fornax ~]$ time ./sum 10000000
==============================================================
        Sum 10000000 numbers
Performance:              Runtime (ms)      MFLOPS
--------------------------------------------------------------
Sum:                      24.999857         800.004578

real    0m0.200s
user    0m0.179s
sys     0m0.020s
```

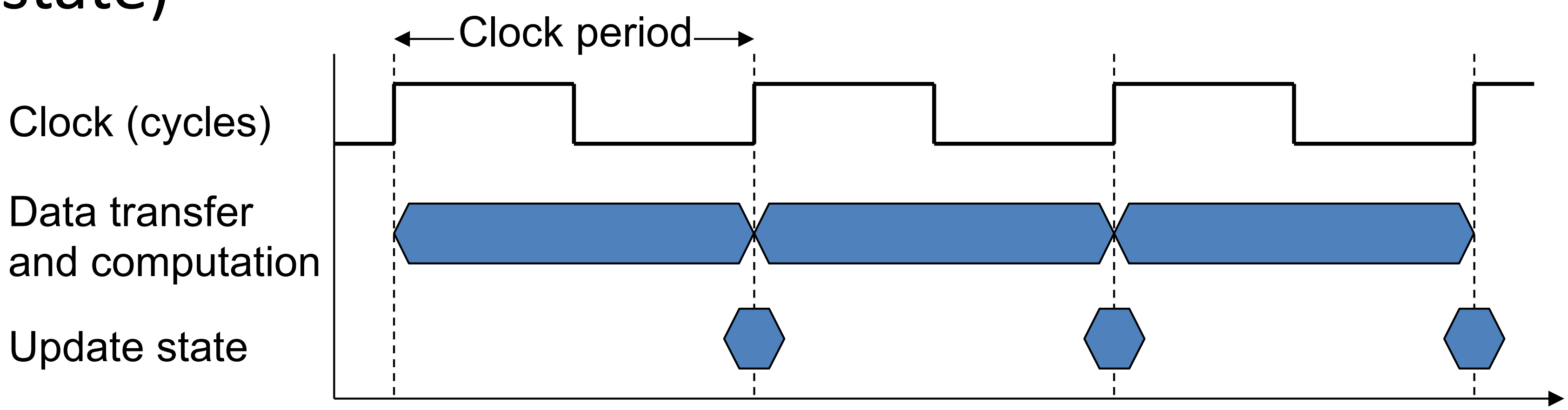
CPU Frequency and Clocking

- CPU Frequency can be obtained by checking /proc/cpuinfo
  - Intel Xeon ® W-2133 CPU @ 3.60 GHz
  - From Intel official website: https://ark.intel.com/content/www/us/en/ark/products/125040/intel-xeon-w-2133-processor-8-25m-cache-3-60-ghz.html



CPU Clocking

- Operation of digital hardware governed by a constant-rate clock, alternating high-low voltage (0 and 1 binary state)



- **Clock period: duration of a clock cycle**
  - e.g., 250ps = 0.25ns = 250×10^{-12}s
- **Clock frequency (rate): cycles per second**
  - e.g., 4.0GHz = 4000MHz = 4.0×10^{9}Hz
- **Clock period, or cycle time is 1/Frequency**

About the Unit

10^{-3} s	**ms**	**millisecond**
10^{-6} s	***μs***	**microsecond**
10^{-9} s	**ns**	**nanosecond**
10^{-12} s	ps	picosecond

10^{3} Hz	**kHz**	**kilohertz**
10^{6} Hz	**MHz**	**megahertz**
10^{9} Hz	**GHz**	**gigahertz**

Prefixes for multiples of bits (bit) or bytes (B)

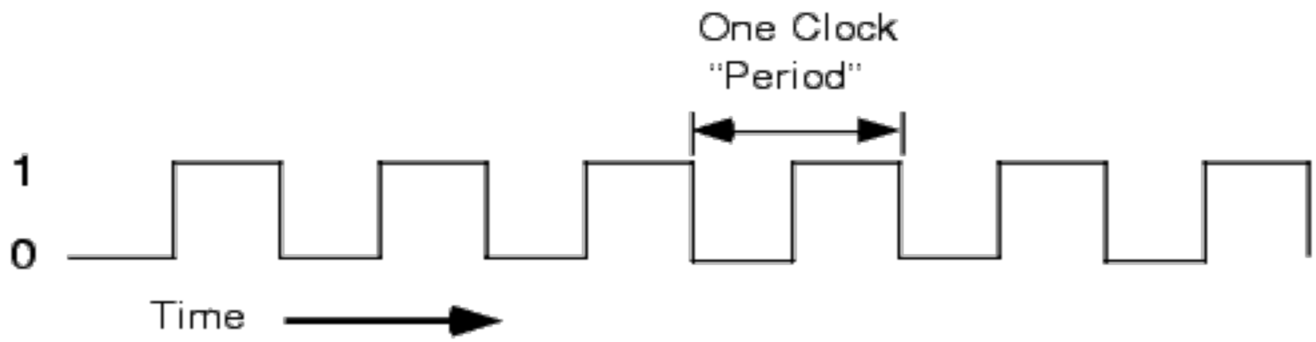
Decimal				**Binary**			
Value		**SI**		**Value**		**IEC**	
1000	10^{3}	k	kilo	1024	2^{10}	Ki	kibi
1000^{2}	10^{6}	M	mega	1024^{2}	2^{20}	Mi	mebi
1000^{3}	10^{9}	G	giga	1024^{3}	2^{30}	Gi	gibi
1000^{4}	10^{12}	T	tera	1024^{4}	2^{40}	Ti	tebi
1000^{5}	10^{15}	P	peta	1024^{5}	2^{50}	Pi	pebi
1000^{6}	10^{18}	E	exa	1024^{6}	2^{60}	Ei	exbi
1000^{7}	10^{21}	Z	zetta	1024^{7}	2^{70}	Zi	zebi
1000^{8}	10^{24}	Y	yotta	1024^{8}	2^{80}	Yi	yobi

CPU Time

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

$$\text{CPU Time}(s) = \#\ \text{CPU Clock Cycles} \times \text{Clock Cycle Time}\ (s)$$

$$= \frac{\#\ \text{CPU Clock Cycles}}{\text{Clock Rate}\ (Hz)}$$



CPU Time Example

- Computer A: 2GHz clock, 10s CPU time to execute a program
- Designing Computer B
  - Aim for 6s CPU time to execute the same program
  - Can do faster clock, but causes to have 1.2 X of clock cycles of A
- How fast must Computer B clock be?

$$\textit{Equation}:\ \text{CPU Time}(s) = \#\,\text{CPU Clock Cycles} \times \text{Clock Cycle Time}\ (s)$$

$$= \frac{\#\,\text{CPU Clock Cycles}}{\text{Clock Rate}\ (Hz)}$$

1. $\text{Clock Rate}_\text{B}(Hz) = \frac{\#\,\text{Clock Cycles}_\text{B}}{\text{CPU Time}_\text{B}} = \frac{1.2 \times \#\,\text{Clock Cycles}_\text{A}}{6\text{s}}$
2. $\#\,\text{Clock Cycles}_\text{A} = \text{CPU Time}_\text{A} \times \text{Clock Rate}_\text{A}$
   $= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$
3. $\text{Clock Rate}_\text{B}(Hz) = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$

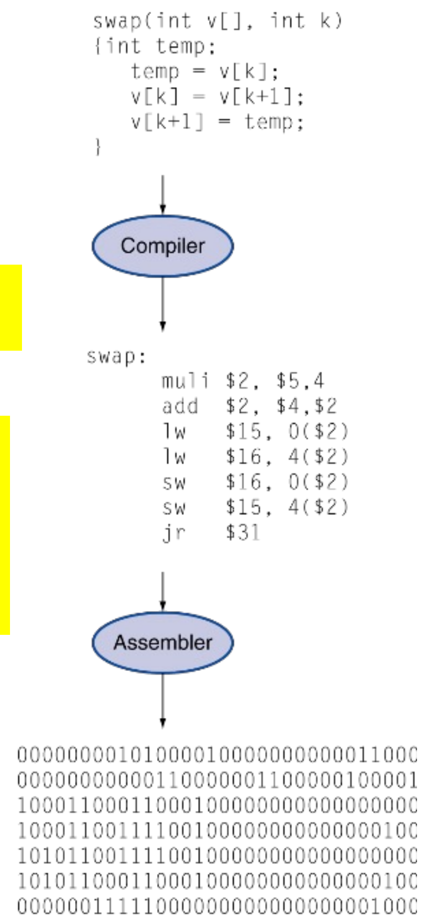
Instruction Count and CPI

- Hardware/CPU executes a program instruction by instructions

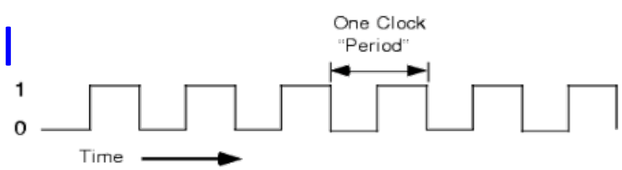
$$\text{CPU Time}(s) = \#\,\boldsymbol{CPU\ Clock\ Cycles} \times \text{Clock Cycle Time }(s)$$
$$= \frac{\#\,\boldsymbol{CPU\ Clock\ Cycles}}{\text{Clock Rate }(Hz)}$$

$$\#\,\textbf{CPU}\ \boldsymbol{Clock\ Cycles} = \#\,\text{Instruction Count} \times \#\,\text{Cycles per Instruction }(CPI)$$

$$\text{CPU Time}(s) = \#\,\boldsymbol{Instruction\ Count} \times \#\,\boldsymbol{CPI} \times \text{Clock Cycle Time }(s)$$
$$= \frac{\#\,\boldsymbol{Instruction\ Count} \times \#\,\boldsymbol{CPI}}{\text{Clock Rate }(Hz)}$$



- 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 program and same set of instructions (ISA)
- Which is faster, and by how much?

$$Equation: \text{CPU Time}(s) = \text{\# CPU Clock Cycles} \times \text{Clock Cycle Time } (s)$$

$$= \frac{\text{\# CPU Clock Cycles}}{\text{Clock Rate } (Hz)}$$

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250ps = I \times 500ps$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500ps = I \times 600ps$$

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

…by this much

CPI in More Detail

$$\# \textbf{ CPU } \boldsymbol{Clock\ Cycles} = \# \text{ Instruction Count} \times \# \text{ Cycles per Instruction } (CPI)$$

- 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

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

CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

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

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
    = 2×1 + 1×2 + 2×3
    = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - Avg. CPI = 9/6 = 1.5

Performance Summary

The BIG Picture

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

- 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

Questions in HW and Tests

- CPU Time = # Clock Cycles * Cycle Time (s) = # Clock Cycles/ClockRate (Hz)

- # Clock Cycles = # Instruction Count * Cycles Per Instruction (CPI)

- Most questions give you two cases (two computers, e.g.) and some known parameters, and you solve the unknown based on the questions

Download, Compile and Execute sum_full.c for Lab 03

- wget https://passlab.github.io/ITSC3181/exercises/sum/sum_full.c

- gcc sum_full.c
- gcc -save-temps sum_full.c -o sum
- ./sum 100000
- time ./sum 10000000

- Checkout sum_full.c and sum_full.s
- Generate assembly code by yourself
  - X86 assembly code is generated using gcc
    - gcc -c -save-temps sum.c
  - https://godbolt.org/

Using Perf for Lab 03

http://www.brendangregg.com/perf.html

```
yanyh@cocsce-l1d39-15:~$ perf stat ./sum 10000000
================================================================================
        Sum 10000000 numbers
--------------------------------------------------------------------------------
Performance:            Runtime (ms)     MFLOPS
--------------------------------------------------------------------------------
Sum:                    23.000002       869.565145

 Performance counter stats for './sum 10000000':

        159.899354      task-clock:u (msec)       #    0.994 CPUs utilized
                 0      context-switches:u        #    0.000 K/sec
                 0      cpu-migrations:u          #    0.000 K/sec
             1,184      page-faults:u             #    0.007 M/sec
       596,350,910      cycles:u                  #    3.730 GHz
     1,620,168,612      instructions:u            #    2.72  insn per cycle
       210,036,915      branches:u                # 1313.557 M/sec
             3,690      branch-misses:u           #    0.00% of all branches

       0.160907353 seconds time elapsed
```

Use perf to collect cycle information for lab 03

- perf stat ./sum 0
  - To collect non-sum instructions profiles as baseline
- perf stat ./sum 1000000
  - To collect instruction profiles that include sum and non-sum ins
  - Instructions and cycles each can be subtracted, but not CPI
  - E.g. for sum 10000000
    - Cycles = 596,350,910 – 308,987
    - Instructions = 1,620,168,612 - 162,870
  - If N=10000000 is huge, baseline can be ignored
  - **Notice the differences of CPU frequency (1.058 GHz vs 3.730 GHz)**

```
yanyh@cocsce-l1d39-15:~$ perf stat ./sum 0
==============================================================================
        Sum 0 numbers
------------------------------------------------------------------------------
Performance:             Runtime (ms)      MFLOPS
------------------------------------------------------------------------------
Sum:                     0.000000         -nan

 Performance counter stats for './sum 0':

          0.292052       task-clock:u (msec)       #    0.301 CPUs utilized
                 0       context-switches:u        #    0.000 K/sec
                 0       cpu-migrations:u          #    0.000 K/sec
                47       page-faults:u             #    0.161 M/sec
           308,987       cycles:u                  #    1.058 GHz
           162,870       instructions:u            #    0.53  insn per cycle
            34,761       branches:u                #  119.023 M/sec
             3,536       branch-misses:u           #   10.17% of all branches

       0.000970656 seconds time elapsed
```

```
yanyh@cocsce-l1d39-15:~$ perf stat ./sum 10000000
==============================================================================
        Sum 10000000 numbers
------------------------------------------------------------------------------
Performance:             Runtime (ms)      MFLOPS
------------------------------------------------------------------------------
Sum:                     23.000002        869.565145

 Performance counter stats for './sum 10000000':

        159.899354       task-clock:u (msec)       #    0.994 CPUs utilized
                 0       context-switches:u        #    0.000 K/sec
                 0       cpu-migrations:u          #    0.000 K/sec
             1,184       page-faults:u             #    0.007 M/sec
       596,350,910       cycles:u                  #    3.730 GHz
     1,620,168,612       instructions:u            #    2.72  insn per cycle
       210,036,915       branches:u                # 1313.557 M/sec
             3,690       branch-misses:u           #    0.00% of all branches

       0.160907353 seconds time elapsed
```

Using PAPI

- perf only profiles the whole program execution
- PAPI can read hardware counter of a specific part of a program
  - Hardware counter records # cycles, # instructions, etc during program execution

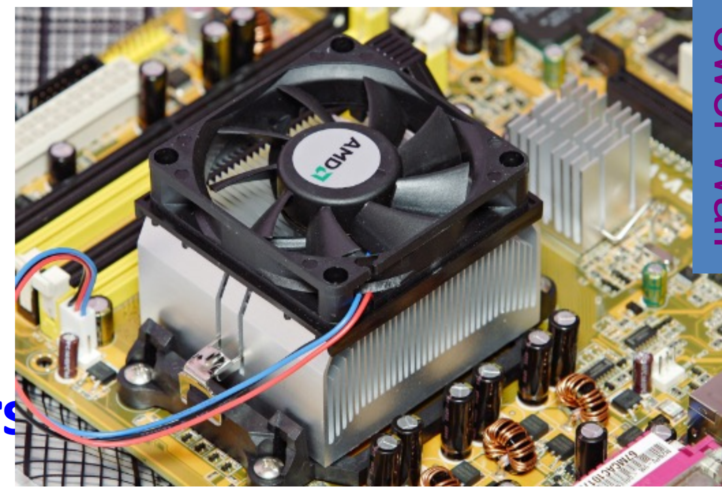
```c
//compile and run:
//gcc papi_example.c  -lpapi -o papi_example
//./papi_example

#include <stdio.h>
#include <papi.h>

#define NUM_PONTOS 400000000
#define NUM_EVENTS 3

int main(int argc, char **argv){
        int EventSet = PAPI_NULL;
        long long values[NUM_EVENTS];
        PAPI_library_init(PAPI_VER_CURRENT);
        PAPI_create_eventset(&EventSet);
        PAPI_add_event( EventSet, PAPI_TOT_INS);
        PAPI_add_event( EventSet, PAPI_TOT_CYC);
        PAPI_add_event( EventSet, PAPI_L1_DCM);
        PAPI_start(EventSet);
        PAPI_reset(EventSet);
        long long int i;
        float pi=0.0;
        for(i=0;i<NUM_PONTOS;i++){
                pi += 4.0/(4.0*i+1.0);
                pi -= 4.0/(4.0*i+3.0);
        }
        PAPI_read(EventSet, values);
        printf("INS: %lld, CYC: %lld, L1 Misses:
alues[0]));
        PAPI_stop(EventSet, NULL);
        return 0;
}
```

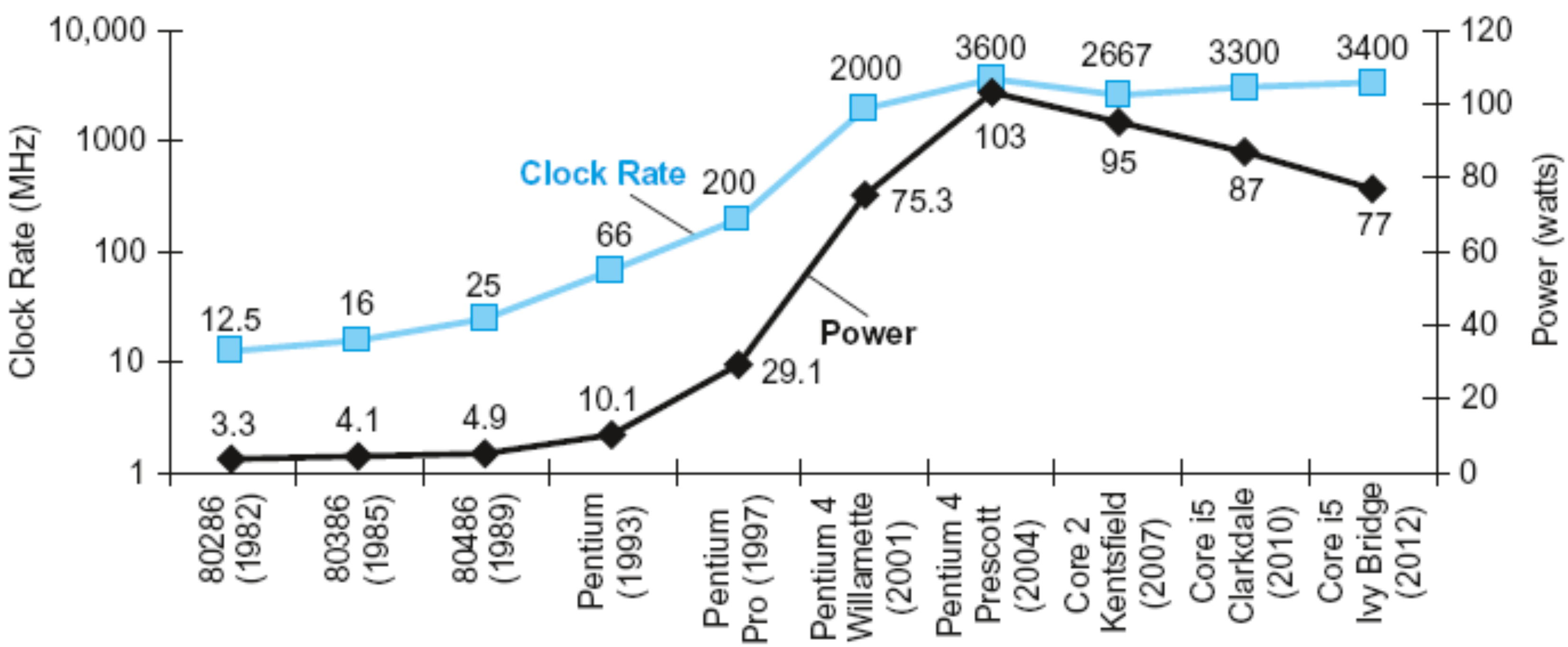
```
yanyh@cocsce-l1d39-10:~$ gcc papi_example.c -lpapi -o papi_example
yanyh@cocsce-l1d39-10:~$ ./papi_example
INS: 11600001549, CYC: 12149354772, L1 Misses: 1074, CPI: 1.047358
```

https://passlab.github.io/ITSC3181/resources/#papi
https://passlab.github.io/ITSC3181/resources/papi_example.c

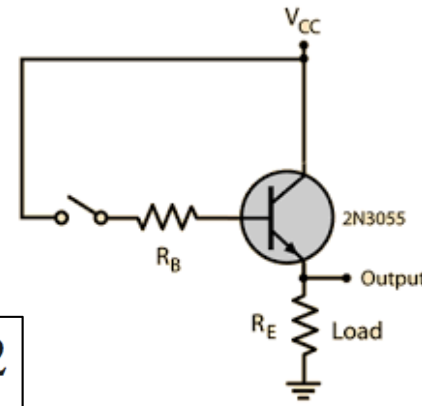
Power and Energy

- Problem:
  - Get power in and distribute around
  - get power out: dissipate heat
- **Revisit Moore's Law**
  - **Transistor density double every 2 years**
  - **Translate to frequency till ~2005**





Dynamic Energy and Power



- Dynamic energy
  - Transistor switch from 0 -> 1 or 1 -> 0

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

- Dynamic power

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

- Reducing clock rate reduces power, not energy
- The capacitive load:
  - a function of the number of transistors connected to an output and the technology, which determines the capacitance of the wires and the transistors.

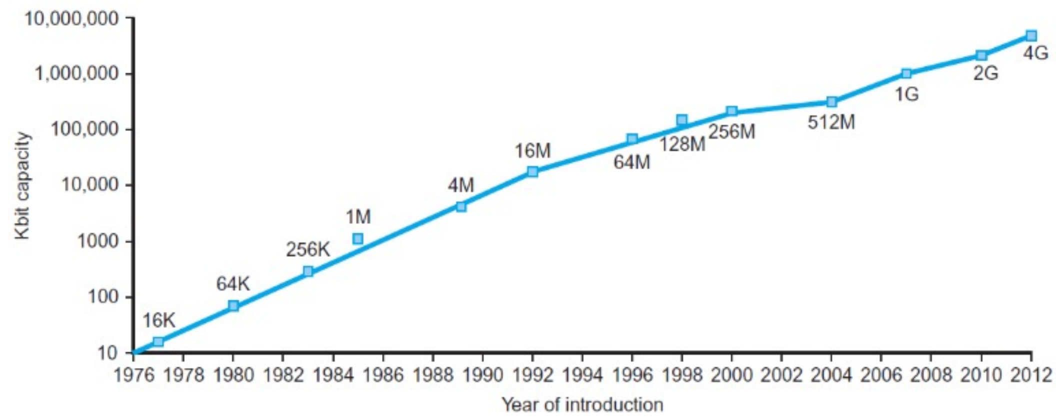
An Example from Textbook

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

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

Technology Trends

- Electronics technology continues to evolve
  - Increased capacity and performance
  - Reduced cost

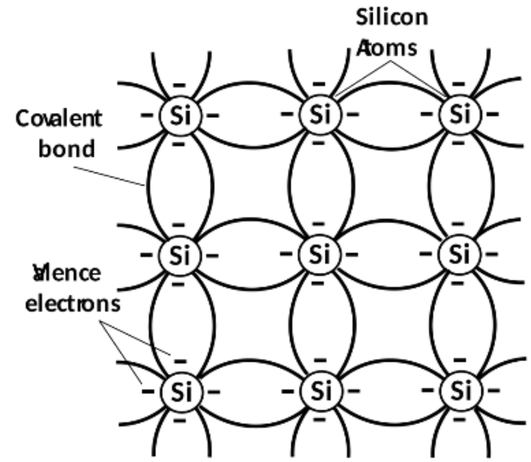


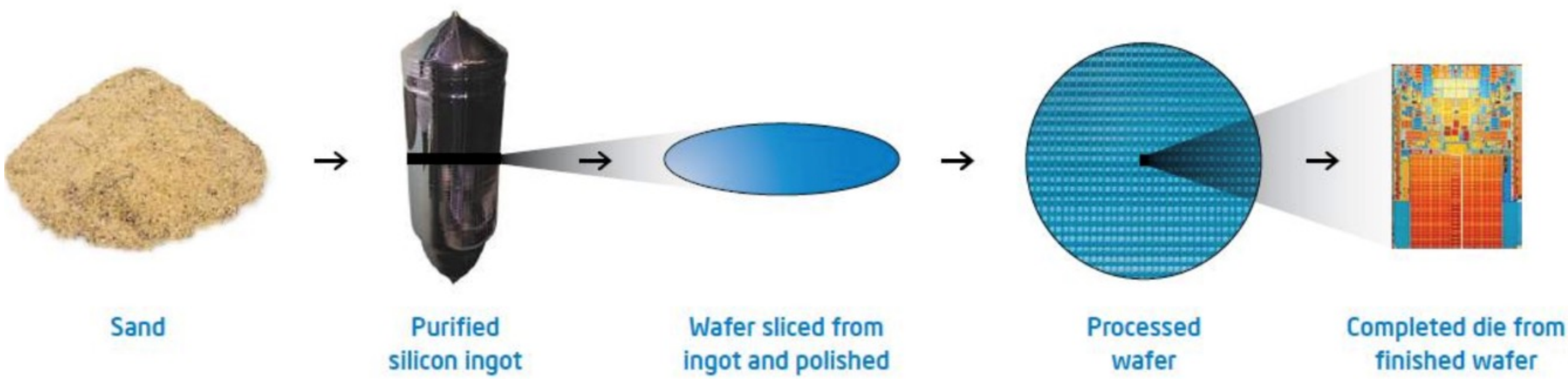
DRAM capacity

Year	Technology	Relative performance/cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit (IC)	900
1995	Very large scale IC (VLSI)	2,400,000
2013	Ultra large scale IC	250,000,000,000

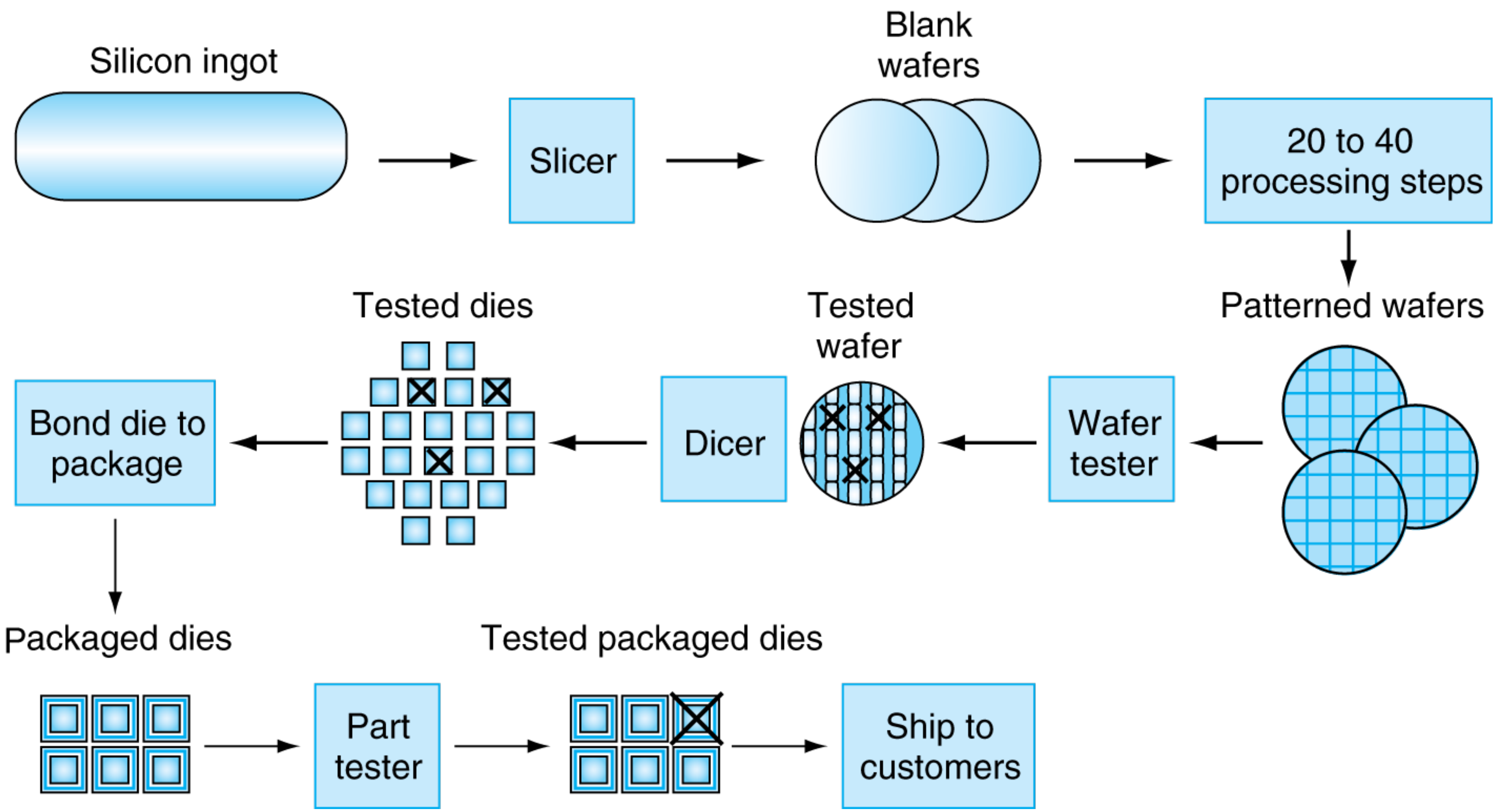
Semiconductor Technology

- Silicon: semiconductor
  - How to turn sand into gold
- Add materials to transform properties:
  - Conductors
  - Insulators
  - Switch



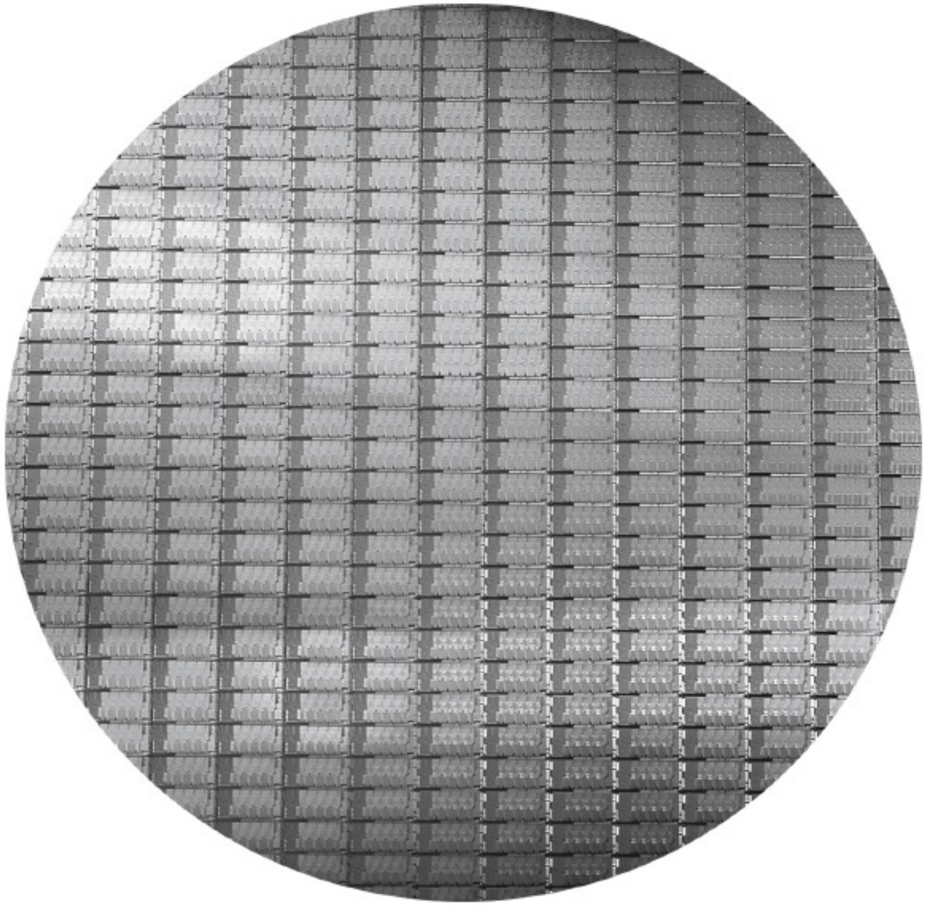


Manufacturing ICs



- Yield: proportion of working dies per wafer

Intel Core i7 Wafer



- 300mm wafer, 280 chips, 32nm technology
- Each chip is 20.7 x 10.5 mm

Silicon Valley