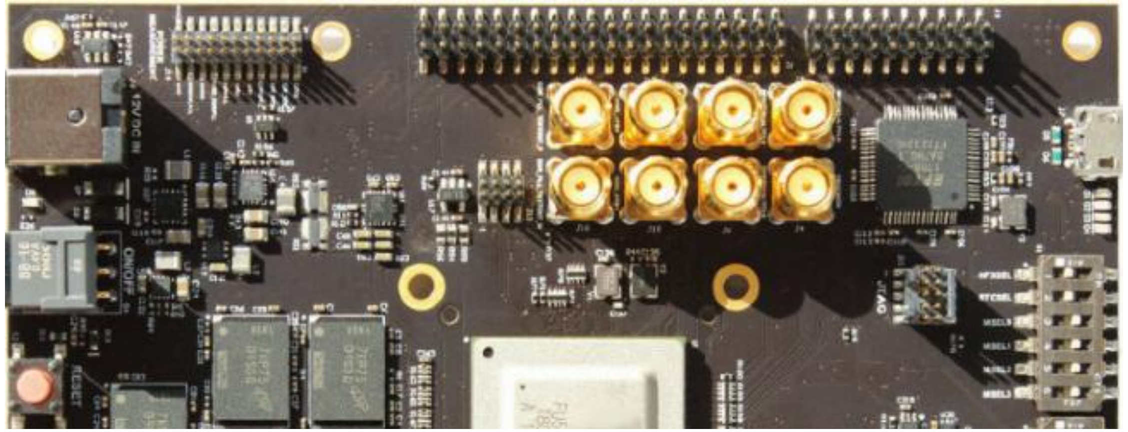
Computer Architecture and Operating Systems
Lecture 1: Introduction

Andrei Tatarnikov

andrewt0301@gmail.com

@andrewt0301

Course Resources



- **Website**

https://andrewt0301.github.io/hse-acos-course/

- **Wiki**

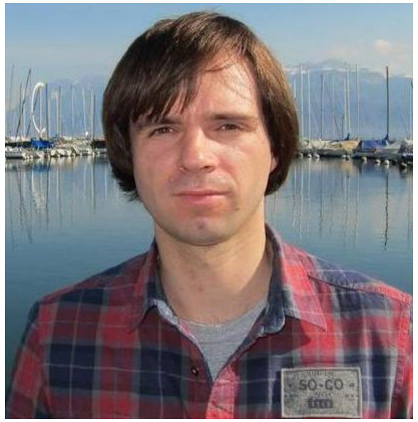
http://wiki.cs.hse.ru/ACOS_DSBA_2023/24
http://wiki.cs.hse.ru/ACOS_COMPDS_2023/2024

- **Telegram**

https://t.me/+LbeE_5yrTQEzYjQy (DSBA)
https://t.me/+OAQIO_59VjZhY2U6 (COMPDS/EAD)

DSBA Course Team

Instructors



Andrei Tatarnikov

Alexey Kanakhin

Roman Stolyarov

Artem Viktorov

Alexander Makhov

Assistants

Fedor Pakhurov

Oleg Malchenko

Vladislav Kirichok

Pavel Nedbay

Artem Borisov

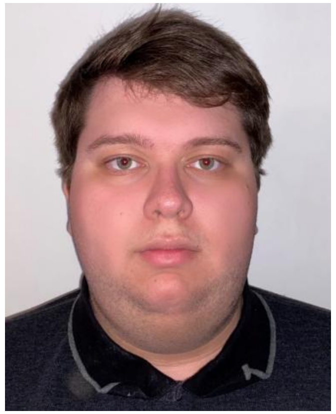
COMPDS/EAD/CSS Course Team

Instructors

Andrei Tatarnikov

Alexandra Borisova

Boris Galitsky



Nikita Golovanov

Assistants

Fedor Pakhurov

Pavel Ivanov

Pavel Nedbay

German Perov

Igor Demushkin

Namig Damirov

Arseny Bolotnikov

Course Outline

Syllabus (see the web site for details)

- Module 3: Computer Architecture
  - Computer architecture
  - Assembly language programming (RISC-V)
  - Home works, quizzes, and test
- Module 4: Operating Systems
  - Operating System Architecture (Linux)
  - System programming in C
  - Home works, quizzes, and test
- Final Exam

Course Motivation

- Increase your computer literacy
- Have an idea how computers under the hood
- Better understand performance
- Be familiar with system programming
- Be familiar with system tools

Example: Matrix Multiplication (part 1)

Python

Floating-point operations:

$2 * n^3 = 2 * (2^{10})^3 = 2^{31}$

Running time:

503.130450 sec.

Performance:

~ 4,27 MFLOPS

```python
import random
from time import time

n = 1024

A = [[random.random()
      for row in range(n)]
      for col in range(n)]
B = [[random.random()
      for row in range(n)]
      for col in range(n)]
C = [[0
      for row in range(n)]
      for col in range(n)]

start = time()
for i in range(n):
    for j in range(n):
        for k in range(n):
            C[i][j] += A[i][k] * B[k][j]
end = time()

print('%0.6f' % (end - start))
```

Example: Matrix Multiplication (part 2)

Java

Floating-point operations:

$2 * n^3 = 2 * (2^{10})^3 = 2^{31}$

Running time:

12.946224 sec.

Performance:

~ 165 MFLOPS

```java
public class Matrix {
    static int n = 1024;
    static double[][] A = new double[n][n];
    static double[][] B = new double[n][n];
    static double[][] C = new double[n][n];

    public static void main(String[] args) {
        java.util.Random r = new java.util.Random();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                A[i][j] = r.nextDouble();
                B[i][j] = r.nextDouble();
                C[i][j] = 0;
            }
        }
        long start = System.nanoTime();
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                for (int k = 0; k < n; k++) {
                    C[i][j] += A[i][k] * B[k][j];
                }
            }
        }
        long stop = System.nanoTime();
        System.out.println((stop - start) * 1e-9);
    }
}
```

Example: Matrix Multiplication (part 3)

C Language

Floating-point operations:

$2 * n^3 = 2 * (2^{10})^3 = 2^{31}$

Running time:

13.714264 sec.

Performance:

~ 153 MFLOPS

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define n 1024
double A[n][n];
double B[n][n];
double C[n][n];

float tdiff(struct timeval *start, struct timeval *end) {
    return (end->tv_sec - start->tv_sec) +
           1e-6*(end->tv_usec - start->tv_usec);
}
int main(int argc, const char *argv[]) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            A[i][j] = (double)rand() / (double)RAND_MAX;
            B[i][j] = (double)rand() / (double)RAND_MAX;
            C[i][j] = 0;
        }
    }
    struct timeval start, end;
    gettimeofday(&start, NULL);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            for (int k = 0; k < n; k++) {
                C[i][j] += A[i][k] * B[k][j];
            }
        }
    }
    gettimeofday(&end, NULL);
    printf("%0.6f\n", tdiff(&start, &end));
    return 0;
}
```

Example: Matrix Multiplication (part 4)

C Language: Optimizations

Loop order: i, j, k

```
for (int i= 0; i < n; i++) {
  for (int j= 0; j < n; j++) {
    for (int k= 0; k < n; k++) {
      C[i][j]+= A[i][k]*B[k][j];
    }
  }
}
```

Running time:
13.714264 sec.
Performance:
~ 153 MFLOPS

Loop order: i, k, j

```
for (int i= 0; i < n; i++) {
  for (int k= 0; k < n; k++) {
    for (int j= 0; j < n; j++) {
      C[i][j]+= A[i][k]*B[k][j];
    }
  }
}
```

Running time:
2.739385 sec.
Performance:
~ 795 MFLOPS

Loop order: j, k, i

```
for (int j= 0; j < n; j++) {
  for (int k= 0; k < n; k++) {
    for (int i= 0; i < n; i++) {
      C[i][j]+= A[i][k]*B[k][j];
    }
  }
}
```

Running time:
19.074106 sec.
Performance:
~ 113 MFLOPS

Example: Matrix Multiplication (part 5)

Feature	Specifiction
Model	MacBook Pro 9,1
Processor Name	Quad-Core Intel Core i7
Processor Speed	2,3 GHz
Number of Processors	1
Total Number of Cores	4
Floating-Point Operations per Cycle	4
L2 Cache (per Core)	256 KB
L3 Cache:	6 MB
Hyper-Threading Technology	Enabled
Memory	8 GB

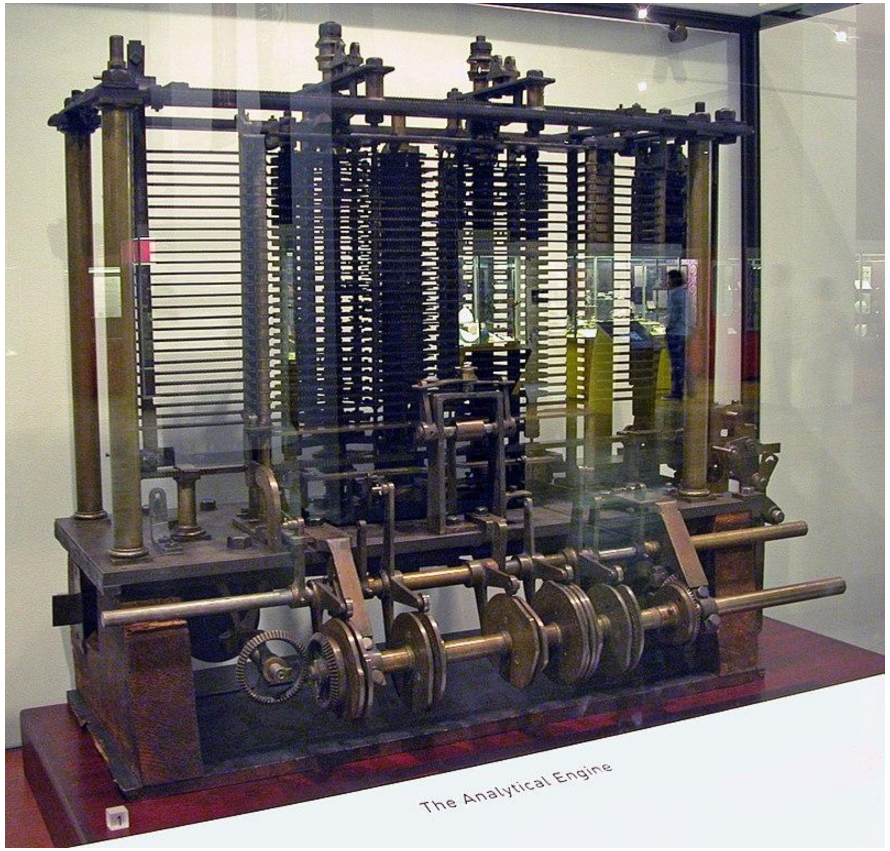
Peak = $(2.3 * 10^9)$ * 1 * 4 * 4 = 36 800 MFLOPS

What affects performance?

Hardware/Software Component	How It Affects Performance
Algorithm	Determines both the number of source-level statements and the number of I/O operations executed
Programming Language, Compiler, and Architecture	Determines the number of computer instructions for each source-level statement
Processor and Memory System	Determines how fast instructions can be executed
I/O System (Hardware and Operating System)	Determines how fast I/O operations may be executed

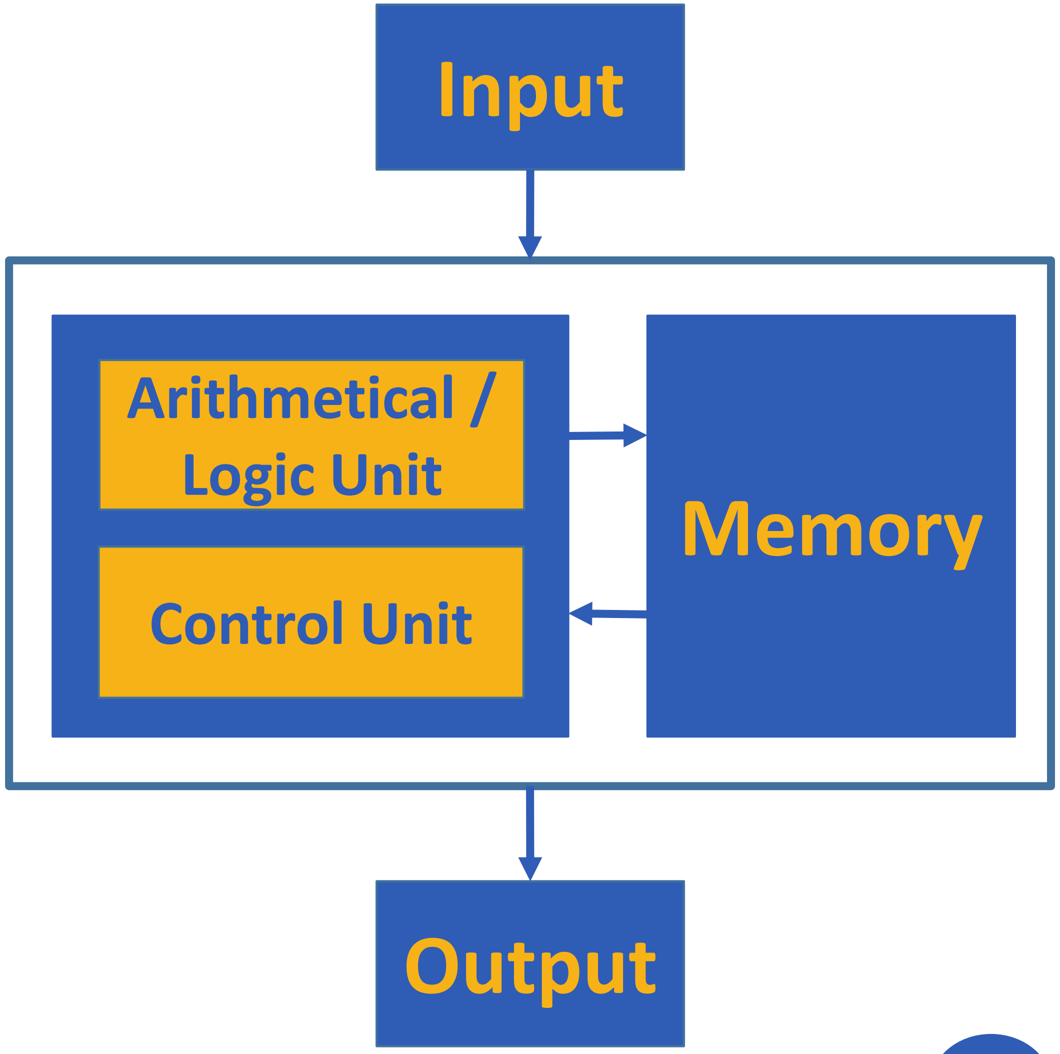
History: 0th Generation – Mechanical

- 1834–71: Analytical Engine designed by Charles Babbage
- Mechanical gears, where each gear represented a discrete value (0-9)
- Programs provided as punched cards
- Never finished due to technological restrictions



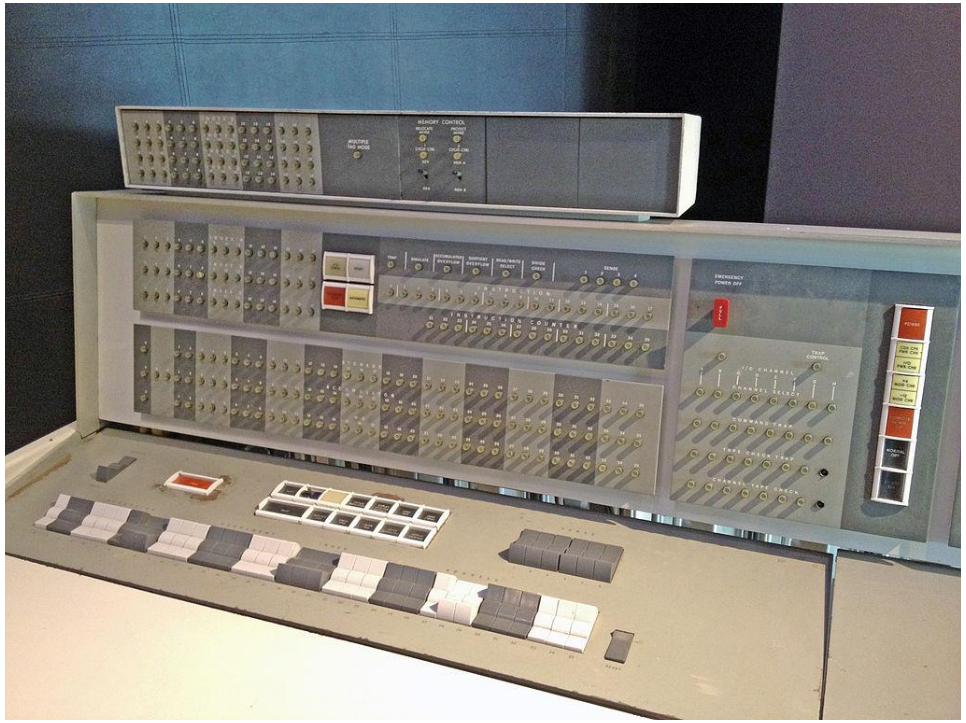
History: 1st Generation - Vacuum Tubes

- 1945–55: first machines were created (Atanasoff–Berry, Z3, Colossus, ENIAC)
- All programming in pure machine language
- Connecting boards and wires, punched cards (later)
- Stored program concept



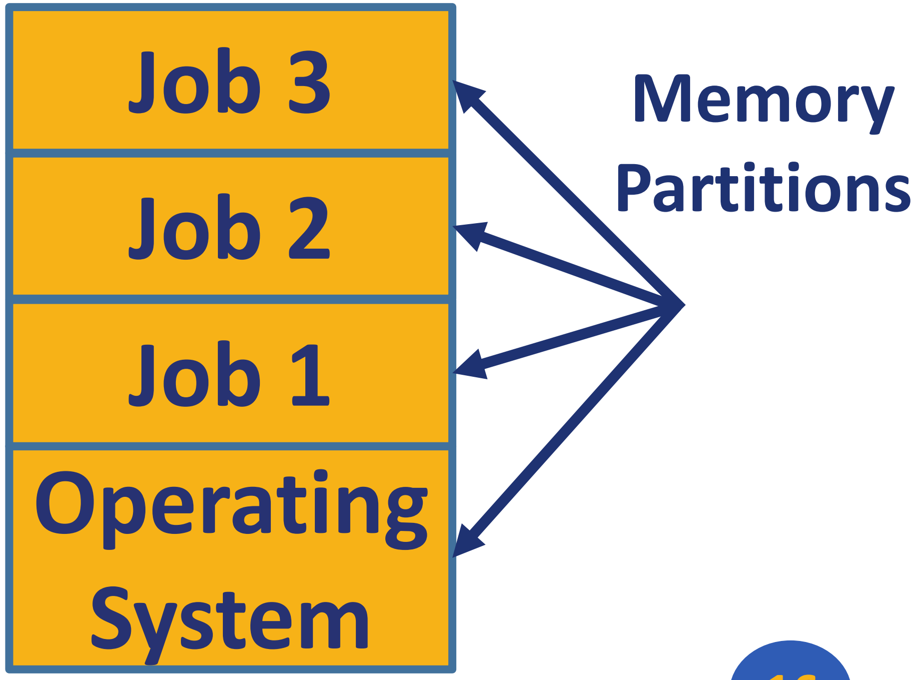
History: 2nd Generation - Transistors

- 1955–65: era of mainframes (e.g. IBM 7094) used in large companies
- Programming in assembly language and FORTRAN
- Batch systems (IO was separated from calculations)
- Punched cards and magnetic tape
- Loaders (OS ancestors)

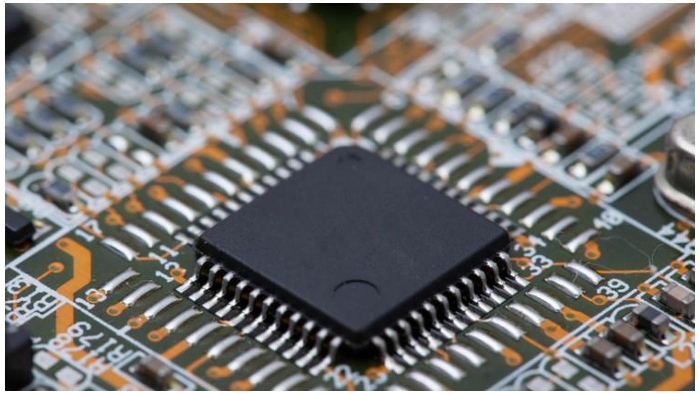


History: 3rd Generation – Integrated Circuits

- 1965–1980: computer lines using the same instruction set architecture (e.g. IBM 360)
- First operating systems (e.g. OS/360, MULTICS)
- Multiprogramming and timesharing
- Computer as utility
- Programming languages and compilers (LISP, BASIC, C)



History: 4^{th} Generation – VLSI and PC

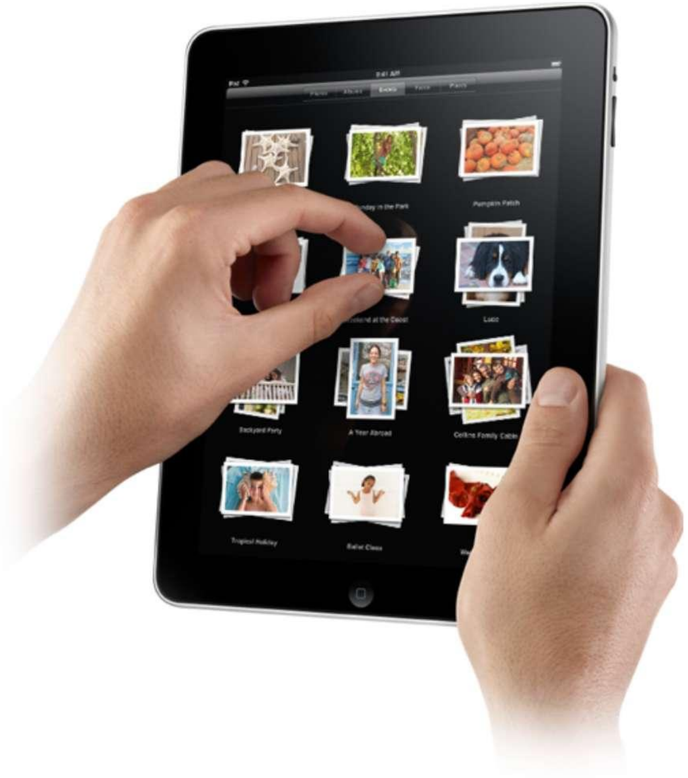


- 1980–Present: personal computers, laptops, servers (Apple, IBM, etc.)
- Architectures: x86-64, Itanium, ARM, MIPS, PowerPC, SPARC, RISC-V, etc.

- Operating systems: UNIX (System V and BSD), MINIX, Linux, MacOS, DOS, Windows (NT)
- ISA (CISC, RISC, VLIW), caches, pipelines, SIMD, vectors, hyperthreading, multicore

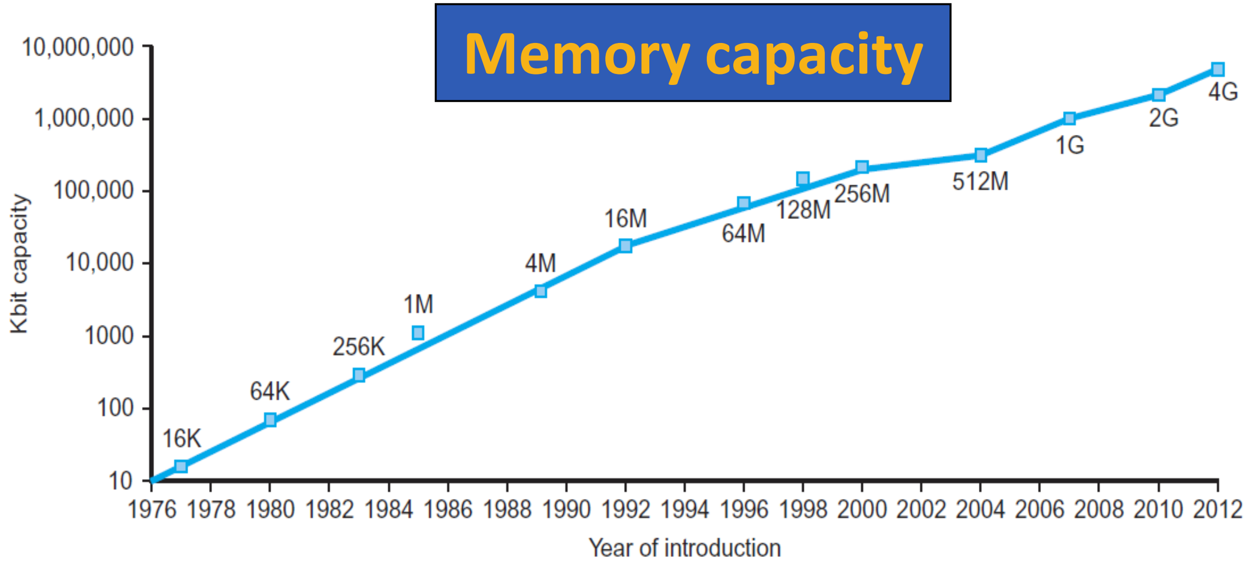
History: 5th Generation – Mobile devices

- 1990–Present: mobile devices, embedded systems, IoT devices
- Custom processors and FPGAs
- Mobile operating systems: Symbian, iOS, Android, Windows Mobile
- Real-time operating systems



Technology Trends

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

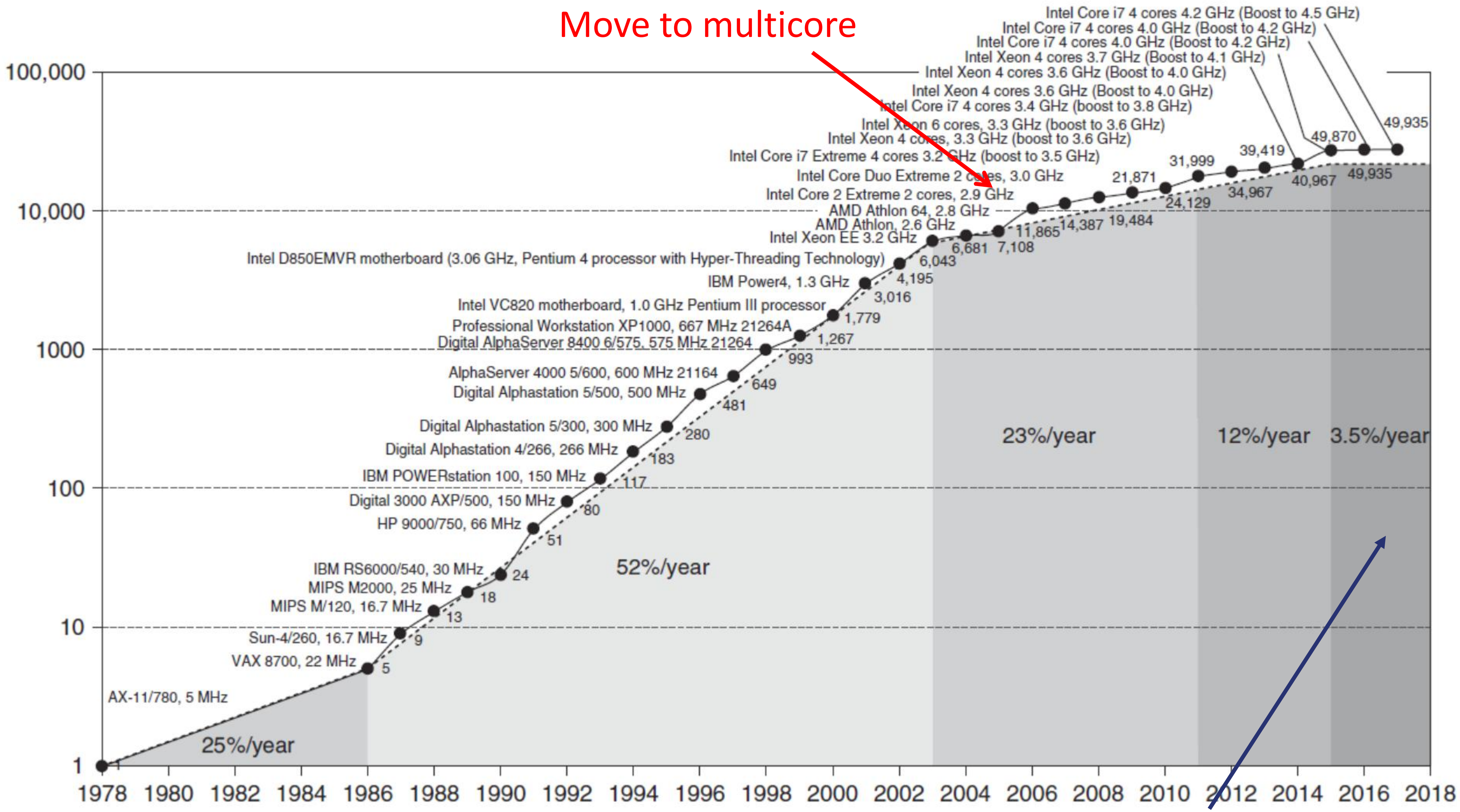


Year	Technology	Relative performance/cost
1951	Vacuum tube	1
1965	Transistor	35
1975	Integrated circuit (IC)	900
1995	Very large scale IC (VLSI)	2,400,000
2013	Ultra large scale IC	250,000,000,000

Moore's Law

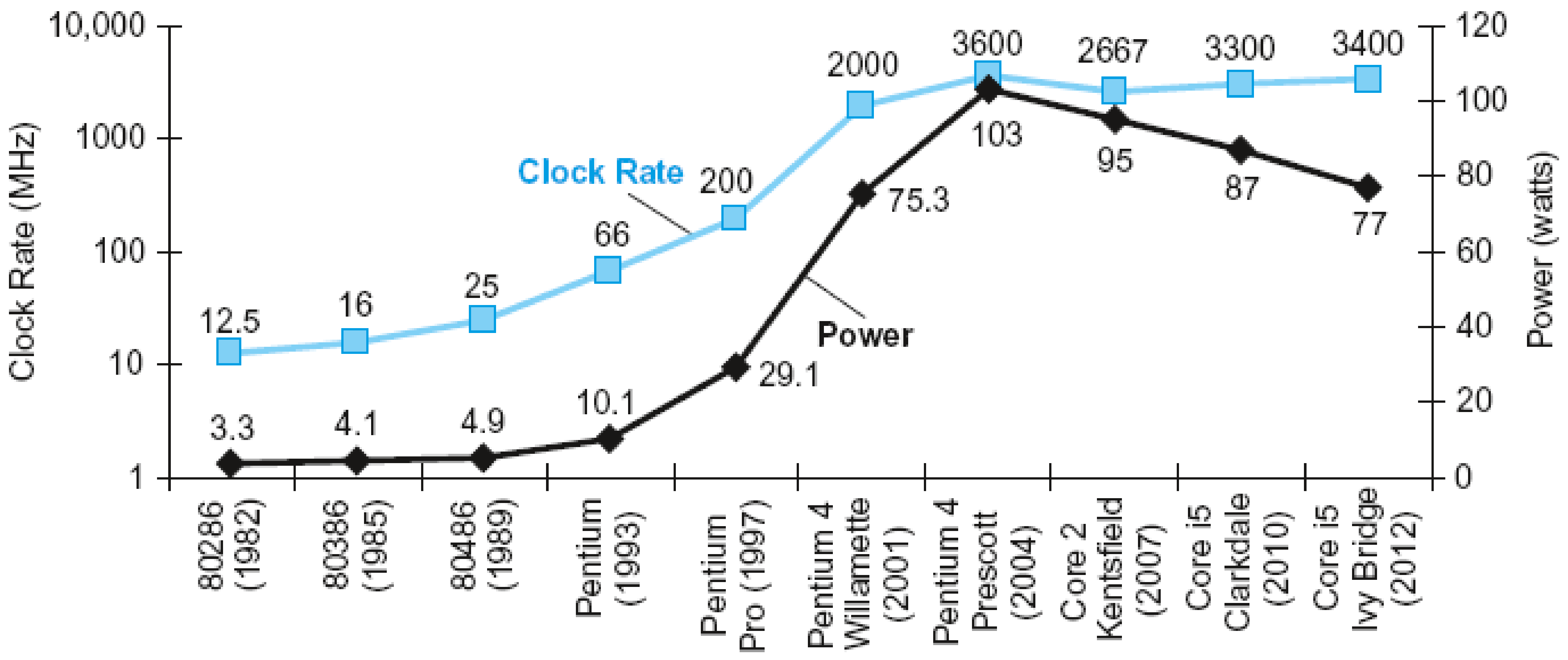
- Gordon Moore (1929-...) cofounded Intel in 1968 with Robert Noyce
- **Moore's Law:** number of transistors on a computer chip doubles every year (observed in 1965)
- Limited by power consumption
- Slowed down since 2010

Single Core Performance

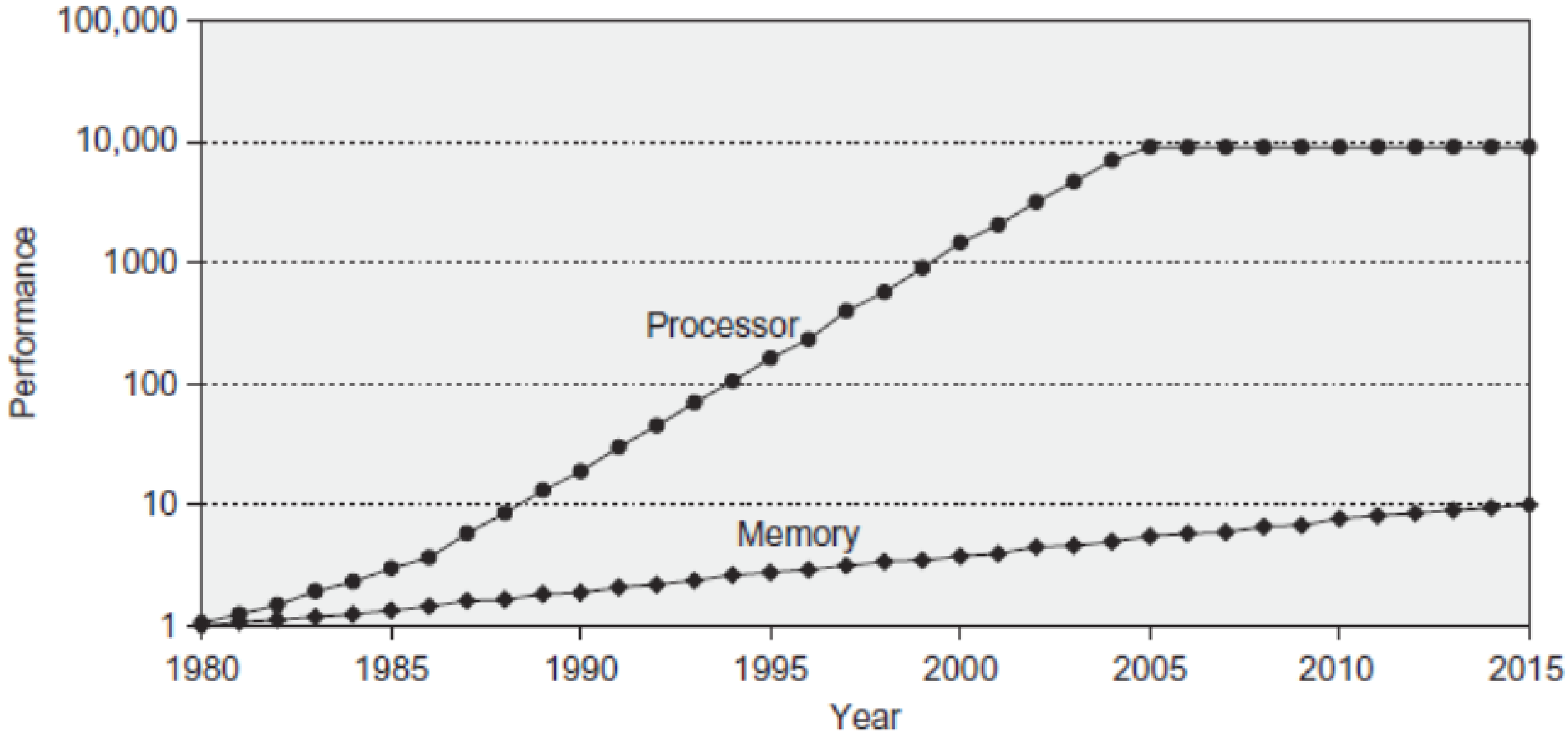


Constrained by power, instruction-level parallelism, memory latency

Power Trends



Memory Performance Gap

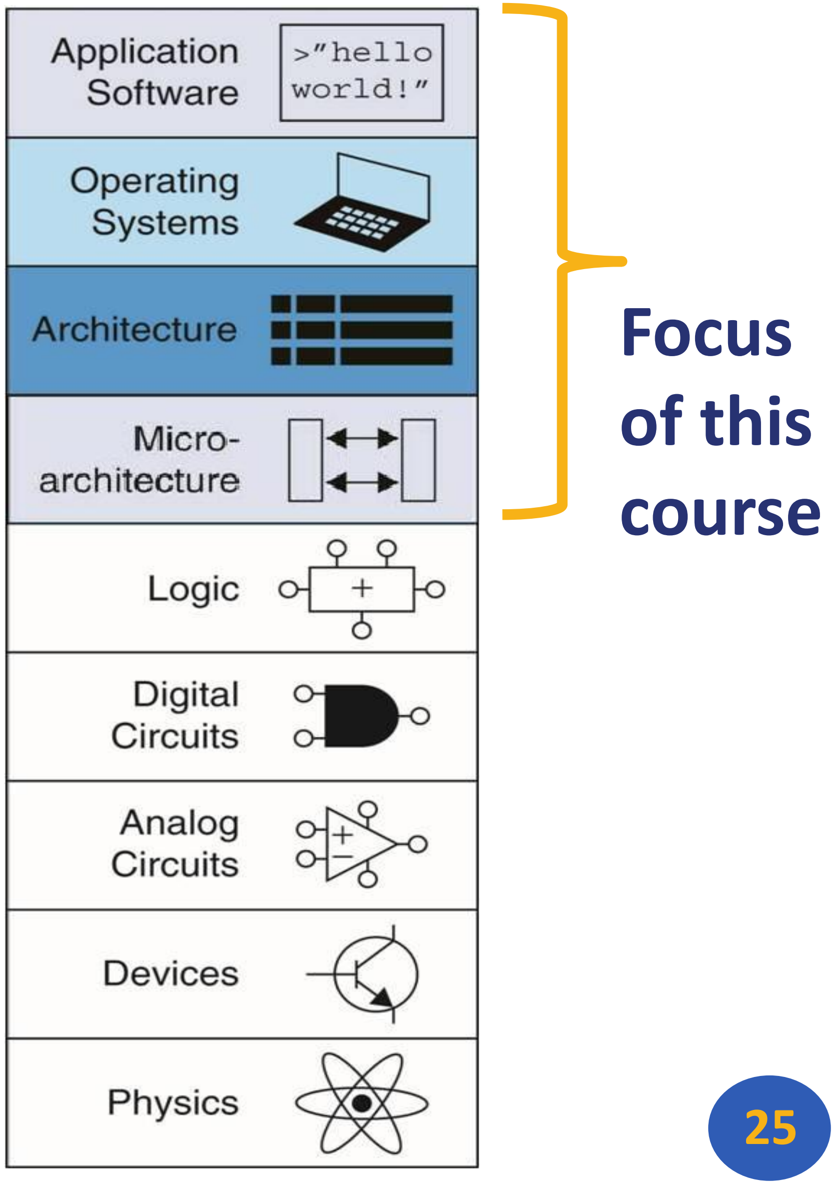


Current Challenges

- Single core performance improvement has ended
  - More powerful microprocessor might not help
- Memory-efficient programming
  - Temporal locality
  - Spatial locality
- Parallelism to improve performance
  - Data-level parallelism
  - Thread-level parallelism
  - Request-level parallelism
- Performance tuning require changes in the application

Concluding Remarks

- To create software that efficiently deals with big data, we need to understand how hardware is organized and managed by operating system
  - Computer architecture
  - Assembly language
  - Compiler basics
  - Operating systems



Any Questions?

```
            .text
__start:    addi t1, zero, 0x18
            addi t2, zero, 0x21
cycle:      beq t1, t2, done
            slt t0, t1, t2
            bne t0, zero, if_less
            nop
            sub t1, t1, t2
            j cycle
            nop
if_less:    sub t2, t2, t1
            j cycle
done:       add t3, t1, zero
```