



NATIONAL RESEARCH  
UNIVERSITY



# Computer Architecture and Operating Systems

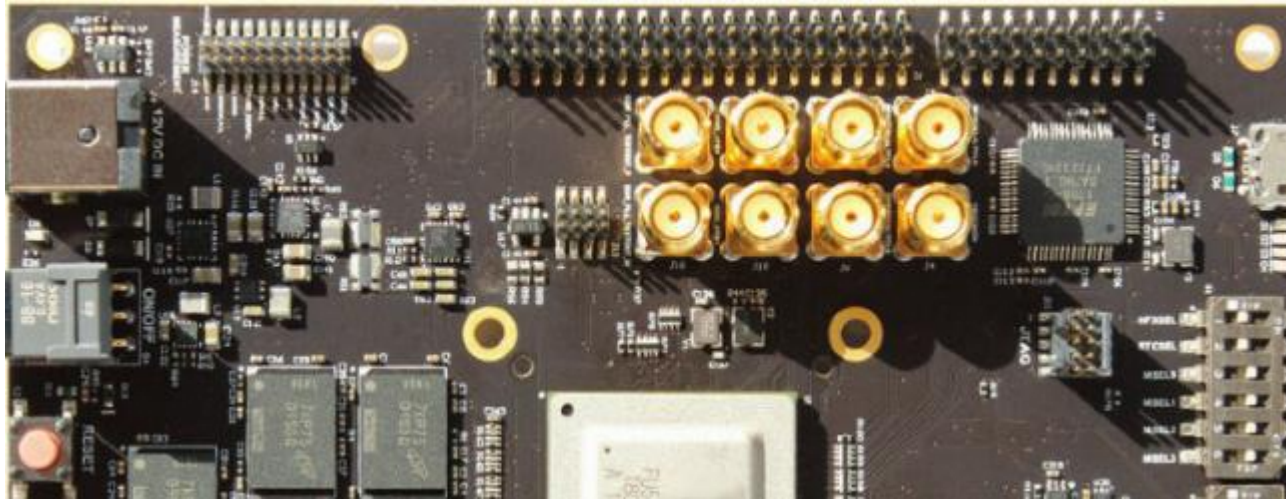
## Lecture 1: Introduction

**Andrei Tatarnikov**

[atatarnikov@hse.ru](mailto:atatarnikov@hse.ru)

[@andrewt0301](#)

# Course Resources



- **Wiki**

[http://wiki.cs.hse.ru/ACOS\\_DSBA\\_2020/2021](http://wiki.cs.hse.ru/ACOS_DSBA_2020/2021)

- **Web site**

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

- **Telegram channel**

<https://t.me/joinchat/AAAAAFDXhCd-WvYYZwBPGQ>

# Course Team

## Instructors



**Andrei Tatarnikov**

## Assistants

**TODO**

# 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 work under the hood
- Better understand performance
- Be familiar with system programming
- Be familiar with system tools

# Performance: Matrix Multiplication (part 1)

## Python

Floating-point operations:

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

Running time:

Performance:

```
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))
```

# Performance: Matrix Multiplication (part 2)

## Java

Floating-point operations:

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

Running time:

Performance:

```
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);
    }
}
```

# Performance: Matrix Multiplication (part 3)

## C Language

Floating-point operations:

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

Running time:

Performance:

```
#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;
}
```



# Performance: Matrix Multiplication (part 4)

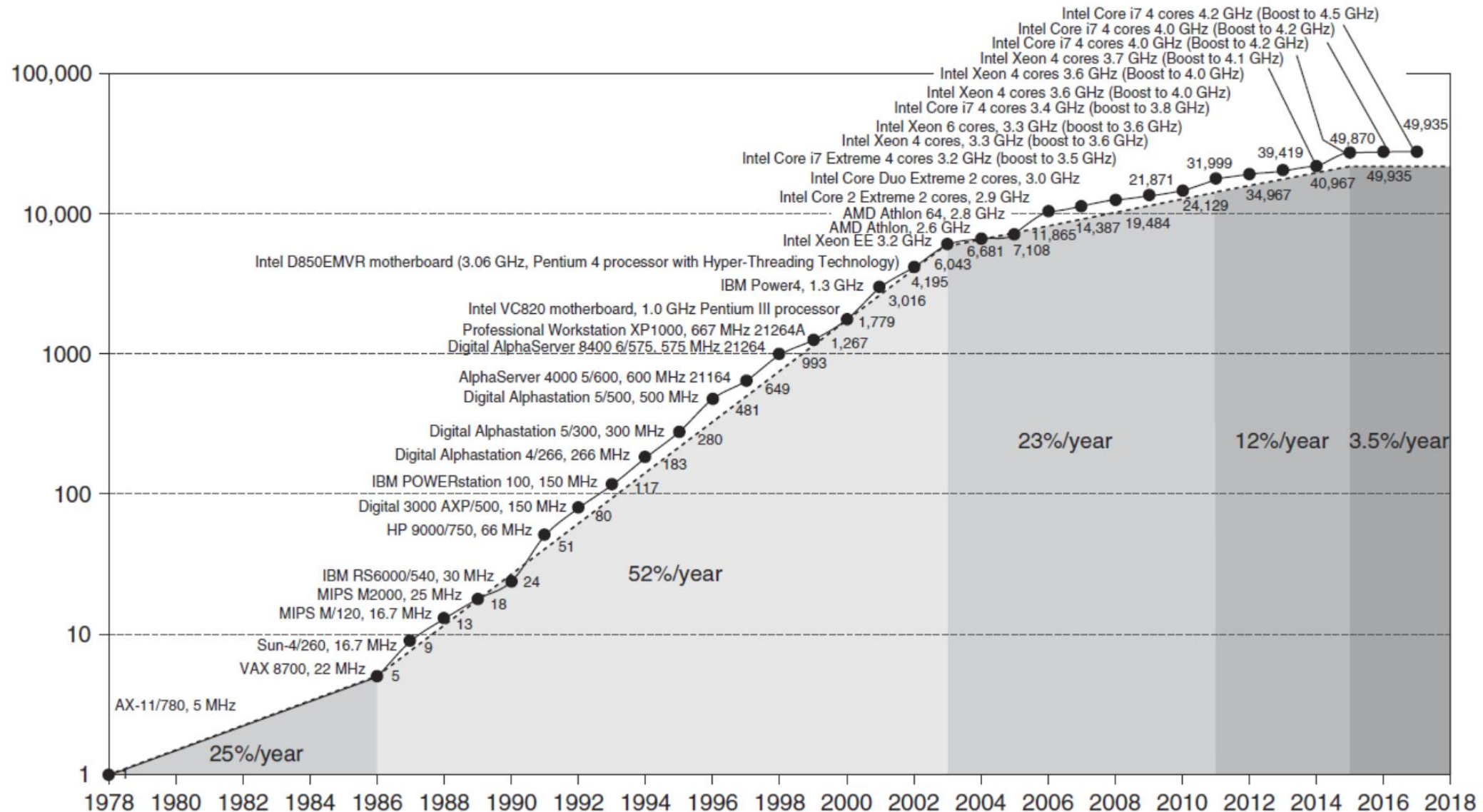
## C Language: Optimizations

### 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];  
        }  
    }  
}
```

# Brief History

# Performance Growth Trend



# Eight Great Ideas

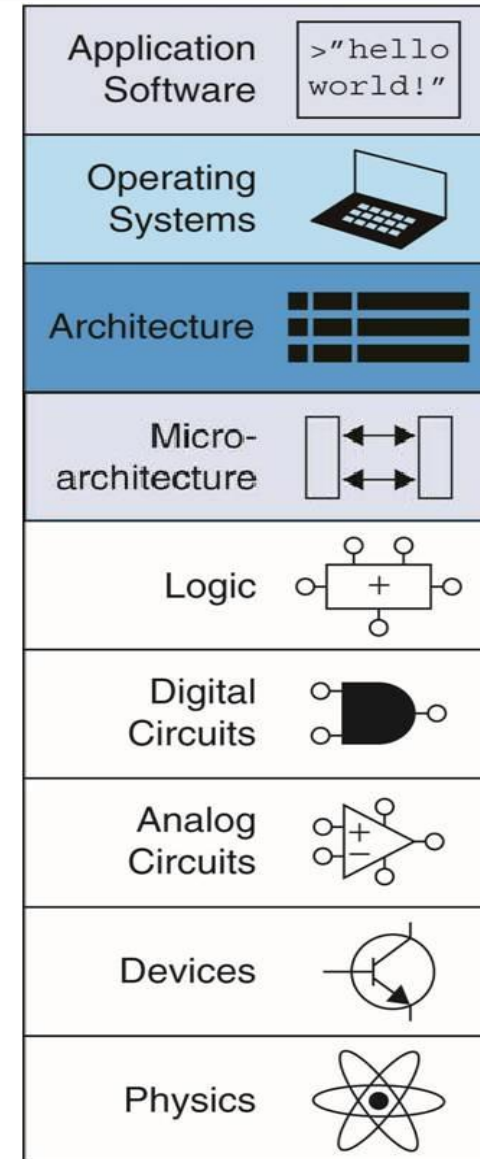
- Design for *Moore's Law*
- Use *abstraction* to simplify design
- Make the *common case fast*
- Performance *via parallelism*
- Performance *via pipelining*
- Performance *via prediction*
- *Hierarchy* of memories
- *Dependability* *via* redundancy



# Abstraction

- Hiding details when they are not important

Focus  
of this  
course



# 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
```