Chapter 1: Introduction

252-0061-00 V Systems Programming and Computer Architecture

This course covers in *depth*…

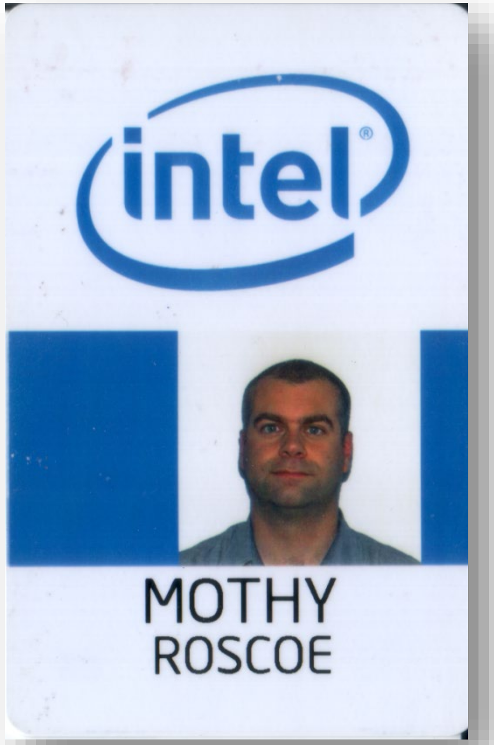
- How to write fast and correct code
- How to write good *systems* code
- What makes programs go fast (and slow)
- Programming in C
  - *Still* the systems programming language of choice
- Programming in Assembly Language
  - What the machine understands
- Programs as more than mathematical objects
  - E.g. how does Facebook work?
  - How programs interact with the hardware

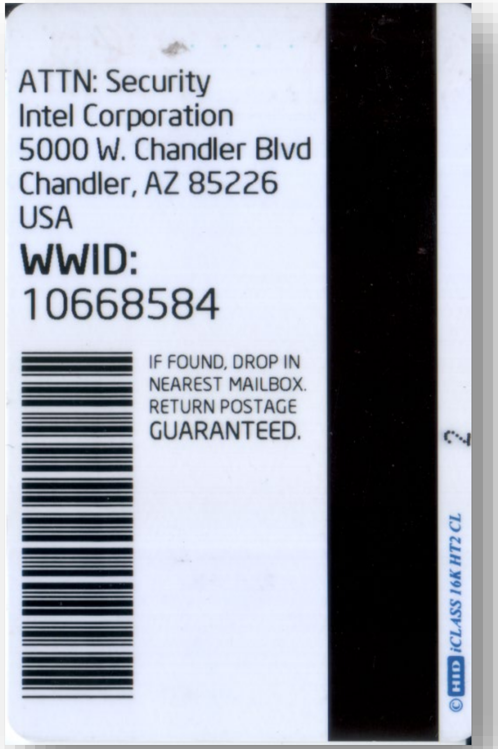
Who are we?

Prof. Timothy Roscoe

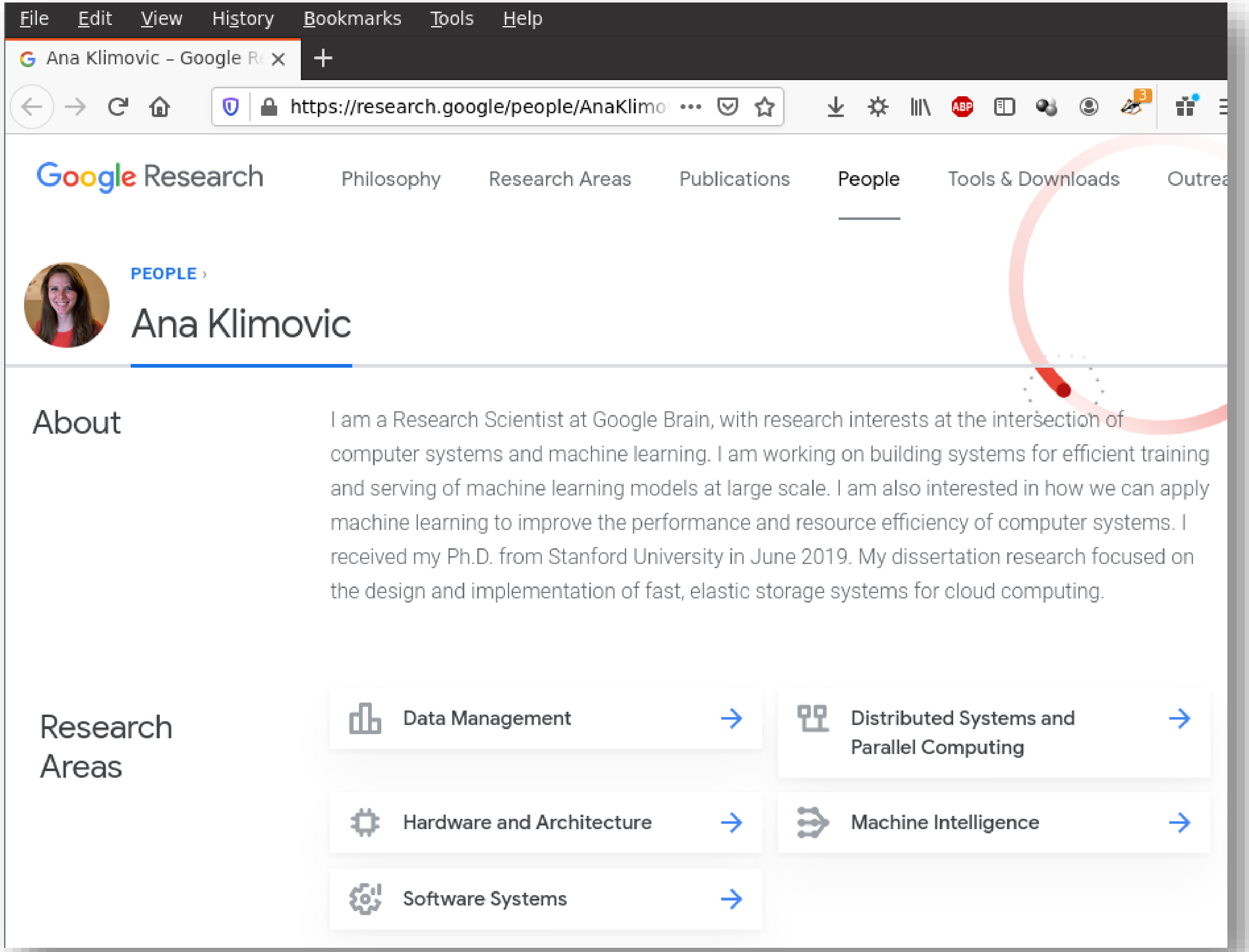
Prof. Ana Klimovic

Full Disclosure





Full Disclosure



Acknowledgements

- Lots of material from the famous **CS 15-213** at Carnegie Mellon University
  - Basis for the **book**
- Some C programming slides adapted from **CSE333** at University of Washington
  - Many thanks to (ex-)Prof. Steve Gribble
- New material:
  - Considerable evolution…
  - Multicore, devices, etc.
  - Mostly our fault ☺

1.1: Logistics

Systems Programming and Computer Architecture

Lectures

- Physical:
  - 10:00-12:00 Tuesdays and Wednesdays
  - HG E 7 (Tue) & NO C 60 (Wed)

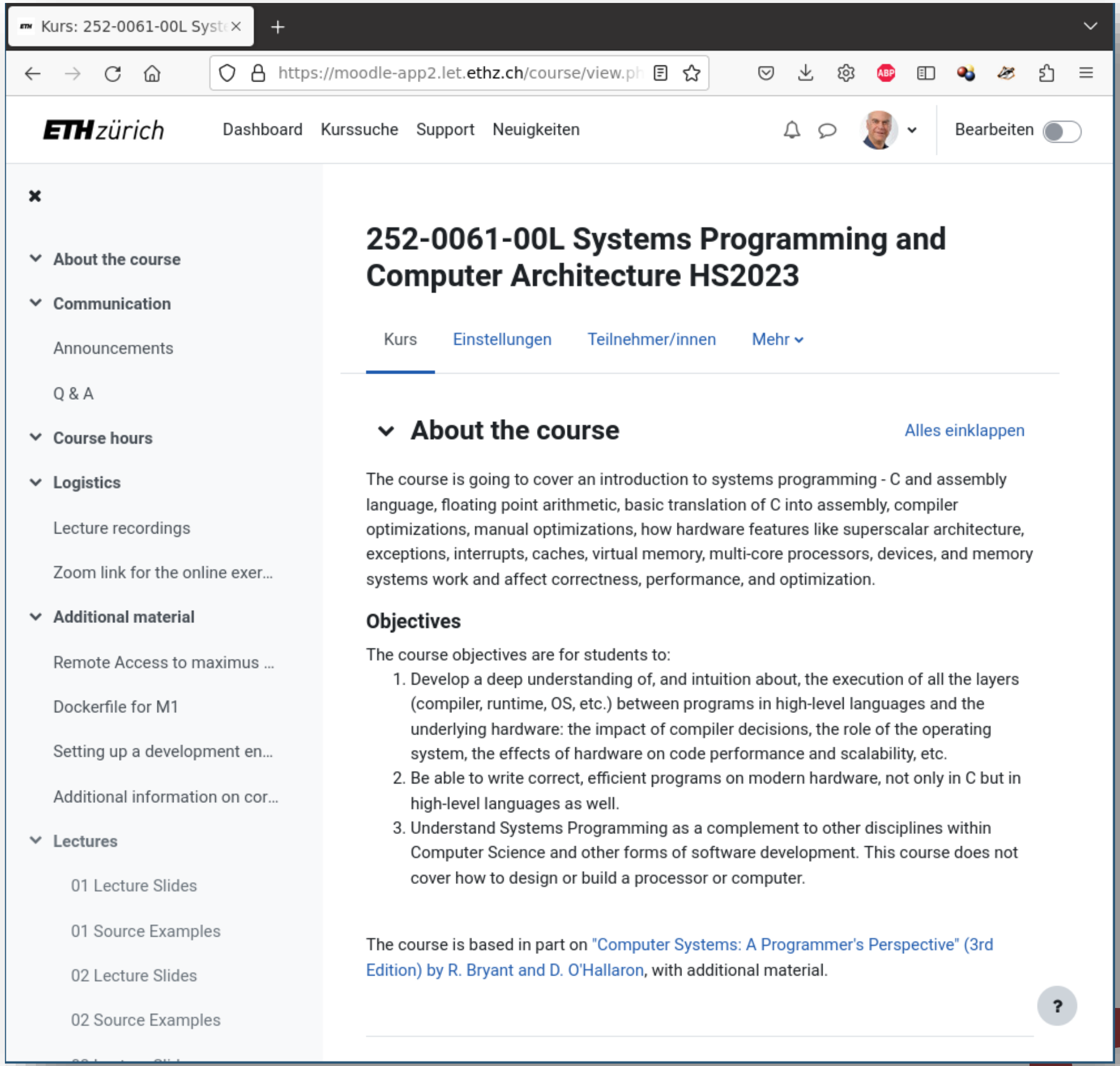
- Recordings will appear after a few days
  - https://video.ethz.ch/lectures/d-infk/2023/autumn.html

The slides are not intended to be understood without the lectures…

Moodle

https://moodle-app2.let.ethz.ch/course/view.php?id=18098

- The first place to look!
- Links posted here
- All lecture materials will be posted on Moodle.
- Ask questions in the forum
  - TAs and Profs will monitor the forum!
- We will not answer questions on Discord.



Tutorial sessions

- Very important!
- Logistics:
  - Wednesday, 12:00-14:00 or 14:00-16:00
  - See myStudies for rooms and **streams**
- Content:
  - Tools and skills for **lab exercises**
  - Knowledge needed for exams, but not in the lectures!
- There **will** be a session this Wednesday (tomorrow)

Language

- We’ll teach in English (and C…)
  - If we speak too fast, or say something unclear, raise your hand!
  - Please ask questions!

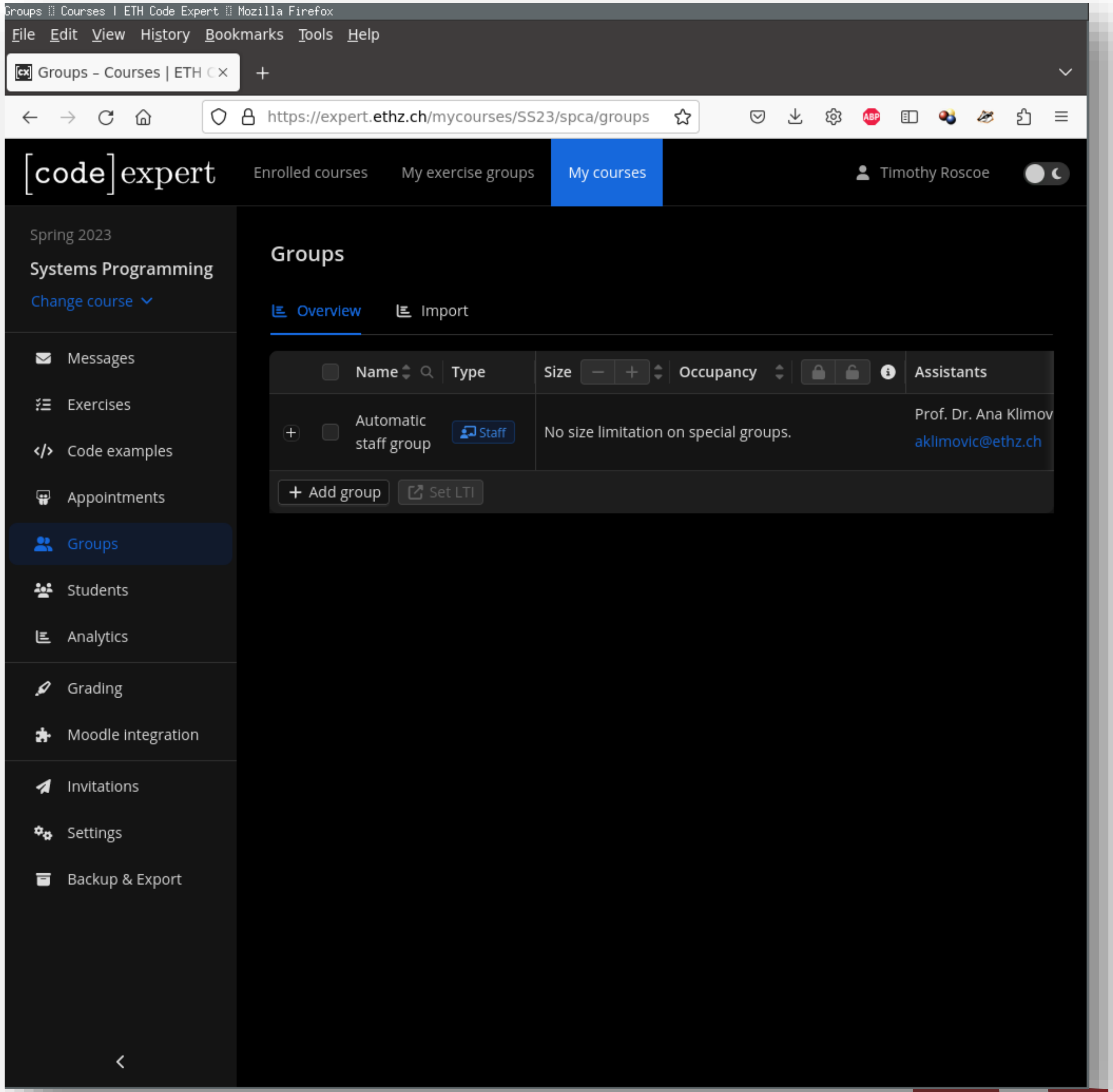
- Assistants speak German, English, Italian, French, …

Asking questions

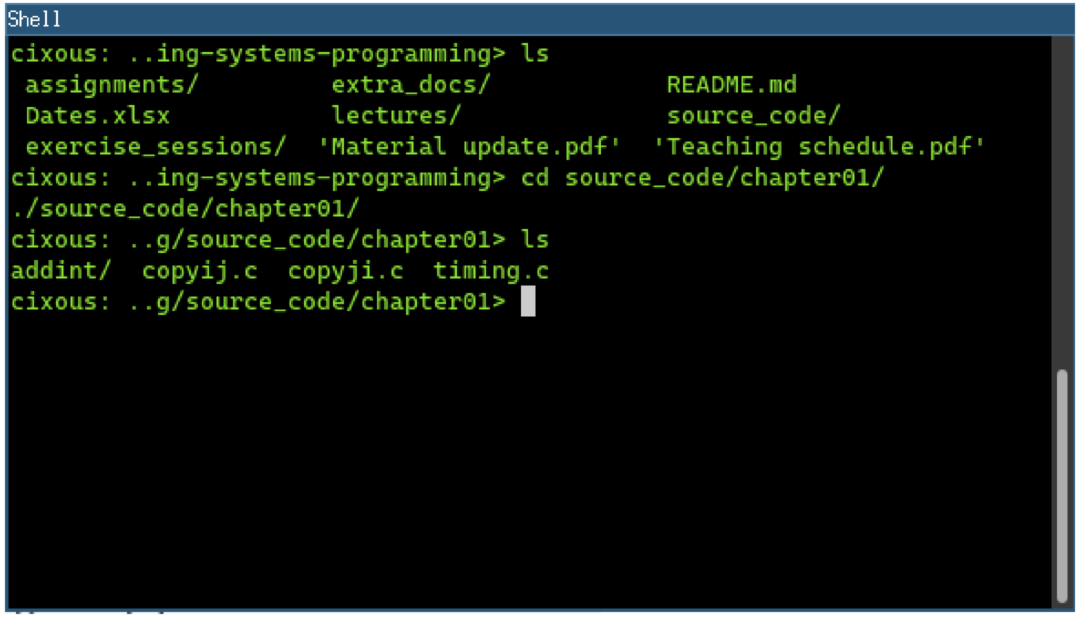
1. Ask during the lectures
2. Ask on the Moodle forum outside the lectures
3. Ask your friends
4. Check the web
5. Ask your teaching assistant
6. Ask *another* teaching assistant
7. Email us (`troscoe@inf.ethz.ch` or `aklimovic@ethz.ch`)

What’s new?

- This year we will use **CodeExpert** for the exam.
  - There will be a midterm exam
  - It will not contribute to your grade
  - Some exercises, labs and demos will move to CodeExpert
- Course topics will not change significantly
  - Mode of answering questions will
  - More C experience will be needed



What’s new?



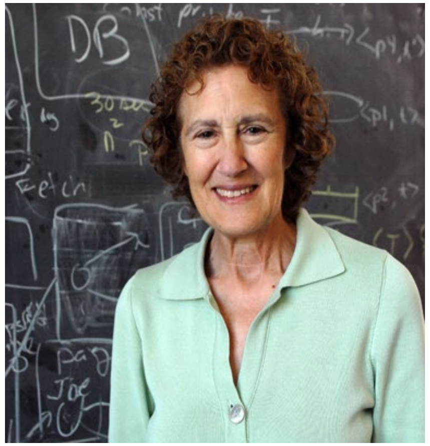
- You’ll be seeing a lot more code
  - We’ll make all source code available to play with
- You’ll need to be comfortable with the command line
  - We’ll spend a lot of time at the prompt.
  - Unix shell commands
  - `ls, more, make, gcc, gdb, grep, rm, mv, …`
- Why?
  - Closer to reality: the topic of this course
  - Makes in clearer what is really going on

Programming environment

- This course targets:
  - CodeExpert (of course)
  - Linux Ubuntu 22.04 LTS on 64-bit x86 PC hardware
- Various options for this exist:
  - Native install, or log into lab machines or Optimus
  - Windows Subsystem for Linux (from the Store)
  - Virtual machine installation (e.g. VirtualBox)
  - *Use another environment and wing it…*

- *We'll help, but there's a limit to how weird we're prepared to get.*

Questions?



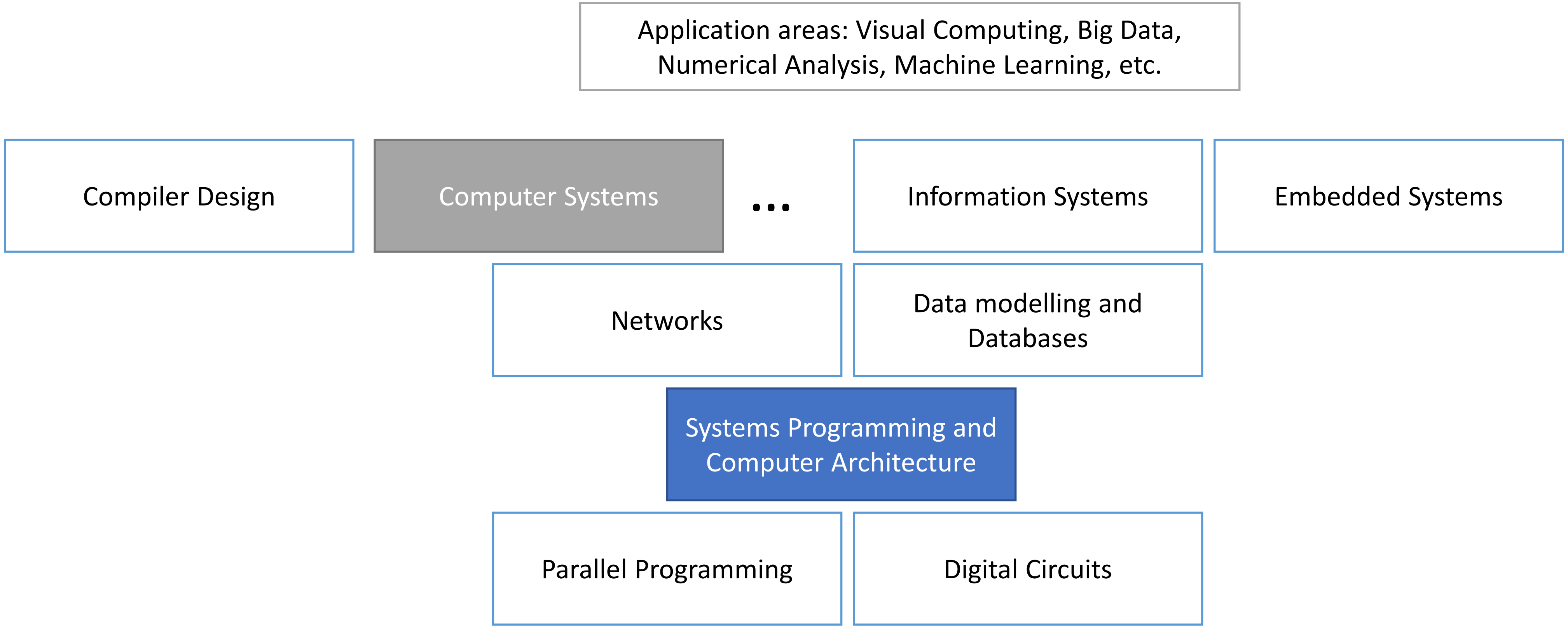
1.2: What is Systems Programming?

Systems Programming and Computer Architecture

“Systems” as a field

- Encompasses:
  - Operating systems
  - Database systems
  - Networking protocols and routing
  - Compiler design and implementation
  - Distributed systems
  - Cloud computing & online services
  - Big Data and machine learning frameworks
- On and above the *hardware/software* boundary

You are here:


Application areas: Visual Computing, Big Data, Numerical Analysis, Machine Learning, etc.
Compiler Design
Computer Systems
…
Information Systems
Embedded Systems
Networks
Data modelling and Databases
Systems Programming and Computer Architecture
Parallel Programming
Digital Circuits

“Systems” as a field

„In designing an operating system one needs both theoretical insight and horse sense. Without the former, one designs an ad hoc mess; without the latter one designs an elephant in best Carrara marble (white, perfect, and immobile).“

Roger Needham and David Hartley,
ACM Symposium on Operating Systems Principles,
1968

Motivation

- Most CS courses emphasize abstraction
  - Abstract data types (objects, contracts, etc.)
  - Program as mathematical object with well-defined behavior
  - Asymptotic analysis (worst-case, complexity)
- These abstractions have limitations
  - Often don’t survive contact with reality
  - Especially in the presence of bugs
  - Need to understand details of underlying implementations

Summary: Course Goals

- Become more effective programmers
  - Find and eliminate bugs efficiently
  - Understand and tune for program performance

- Prepare for later systems classes at ETHZ
  - Compilers, Operating Systems, Networks, Computer Architecture, Embedded Systems

Questions?

1.3: Motivation – Some inconvenient truths about computers

Systems Programming and Computer Architecture

Inconvenient truth:

Computers don’t really deal with numbers.

Systems@ETH zürich

Source code: chapter 1 → addint

Computers don't deal with integers

- Maths:

$$S = \{i_o \ldots i_{k-1}, i \in \mathbb{Z}\}$$

$$T = \sum_{j=0}^{k-1} i_j$$

- Reality:

```
#include <stdio.h>
#include <stdlib.h>

#define BUFFER_LENGTH 80

int main(int argc, char *argv[])
{
    char buffer[BUFFER_LENGTH];
    int total = 0;

    while( fgets( buffer, BUFFER_LENGTH, stdin ) ) {
         total += atoi(buffer);
    }
    printf("Total is %d\n", total);
    return 0;
}
```

This is not a bug – it is correct behavior!

Source code: chapter 1 → `addfloat`

Computers don't deal with reals either

- Maths:

$$\forall\, x, y, z \in \mathbb{R},$$

$$(x + y) + z = x + (y + z)$$

- Reality:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    float x = 1e20;
    float y = -1e20;
    float z = 3.14;

    printf("( x + y ) + z = %f\n", (x + y) + z);
    printf("x + ( y + z ) = %f\n", x + (y + z));
    return 0;
}
```

This is not a bug – it is correct behavior!

Computer arithmetic

- Does not generate random values
  - Arithmetic operations have important mathematical properties
- Cannot assume all “usual” mathematical properties
  - Due to finiteness of representations
  - Integer operations satisfy “ring” properties
    - Commutativity, associativity, distributivity
  - Floating point operations satisfy “ordering” properties
    - Monotonicity, values of signs
- Observation
  - Need to understand which abstractions apply in which contexts
  - Important issues for compiler writers and serious application programmers

Inconvenient truth:

All the best programmers know assembly.

You’ve got to know assembly

- Chances are, you’ll never write a program in assembly
  - Compilers are much better & more patient than you are
- But: understanding assembly is **key** to machine-level execution model
  - Behavior of programs in presence of **bugs**
    - High-level language model breaks down
  - Tuning program **performance**
    - Understand optimizations done/not done by the compiler
    - Understanding sources of program inefficiency
  - Implementing **system software**
    - Compiler has machine code as target
    - Operating systems must manage process state
  - Creating / fighting **malware**
    - x86 assembly is the language of choice!

Source code: chapter 1 → `timing`

Assembly example: measuring cycles

```
uint64_t rdtsc()
{
    uint32_t lo, hi;

    asm volatile("rdtsc; movl %%edx,%0; movl %%eax,%1"
          : "=r" (hi), "=r" (lo)
          :
          : "%edx", "%eax");
    return (lo | (((uint64_t)hi) << 32));
}
```

- Time Stamp Counter
  - Special 64-bit register in Intel-compatible machines
  - Incremented every clock cycle
  - Read with `rdtsc` instruction
- Cannot be read from a programming language
  - Requires assembly code to access
  - C compiler’s asm facility inserts assembly code into generated machine code

Source code: chapter 1 → `timing`

Assembly example: measuring cycles

- Can be used to measure *how many instruction cycles* a computation really took.

- Need to be careful about measurement methodology…

```
int main(int argc, char *argv[])
{
    uint64_t start, overhead;
    unsigned long result;

    // Measure the overhead.
    // We should really repeat this many times.
    start = rdtsc();
    overhead = rdtsc()-start;
    printf("Counter overhead is %lu cycles\n", overhead);

    // Time the function
    start = rdtsc();
    result = calc();
    printf("Time = %lu cycles\n", rdtsc()-start-overhead );
    printf("Result = %lu\n", result );
    return 0;
}
```

Inconvenient truth:

Memory is not a nice array that stores your data

The details of memory

- Memory is **not unbounded**
  - It must be allocated and managed
  - Many applications are memory-dominated
- Memory performance is **not uniform**
  - Cache and virtual memory effects can greatly affect program performance
  - Adapting program to characteristics of memory system can lead to major speed improvements
- Memory is **typed**
  - Different kinds of memory behave differently

Source code: chapter 1 → memory_bug

Memory-related bugs are still a nightmare.

```
typedef struct {
  int a[2];
  double d;
} struct_t;

double fun(int i) {
  volatile struct_t s;
  s.d = 3.14;
  s.a[i] = 1073741824; /* Possibly out of bounds */
  return s.d;
}
```

```
fun(0)   ->       3.14
fun(1)   ->       3.14
fun(2)   ->       3.1399998664856
fun(3)   ->       2.00000061035156
fun(4)   ->       3.14
fun(6)   ->       Segmentation fault
```

Actual results are system-specific…

Systems@ETH Zürich

Source code: chapter 1 → memory_bug

Memory referencing bug

```c
typedef struct {
  int a[2];
  double d;
} struct_t;
```

```
fun(0)  ->      3.14
fun(1)  ->      3.14
fun(2)  ->      3.1399998664856
fun(3)  ->      2.00000061035156
fun(4)  ->      3.14
fun(6)  ->      Segmentation fault
```

Explanation:

	Critical State	6
	?	5
	?	4
struct_t	d7 ... d4	3
struct_t	d3 ... d0	2
struct_t	a[1]	1
struct_t	a[0]	0

Location accessed by `fun(i)`

Systems@ETH Zürich

Source code: chapter 1 → `copyij`

Memory system performance

```
void copyij(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (i = 0; i < 2048; i++)
    for (j = 0; j < 2048; j++)
      dst[i][j] = src[i][j];
}
```

```
void copyji(int src[2048][2048],
            int dst[2048][2048])
{
  int i,j;
  for (j = 0; j < 2048; j++)
    for (i = 0; i < 2048; i++)
      dst[i][j] = src[i][j];
}
```

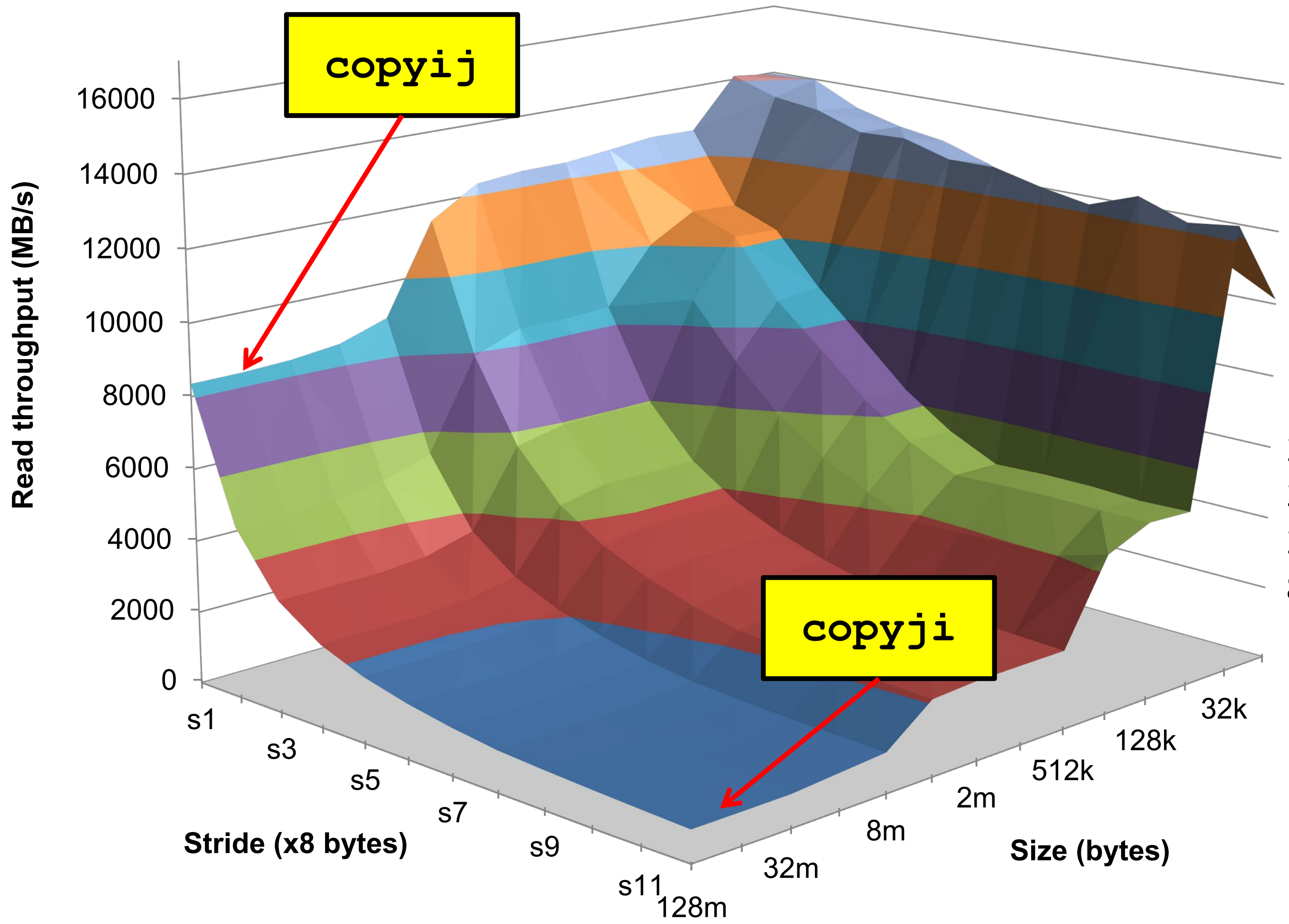
5.2 ms

162 ms !

Intel Core i7 2.7 GHz

- Hierarchical memory organization
- Performance depends on access patterns
  - Including how step through multi-dimensional array

The Memory Mountain



Intel Core i7
2.67 GHz
32 KB L1 d-cache
256 KB L2 cache
8 MB L3 cache

Inconvenient truth:

Performance is about much more than asymptotic complexity

There's much more to performance than asymptotic complexity

- Constant factors matter too – often more.
- Even exact op count does not predict performance
  - Easily see 10:1 performance range depending on how code written
  - Must optimize at multiple levels: algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs compiled and executed
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

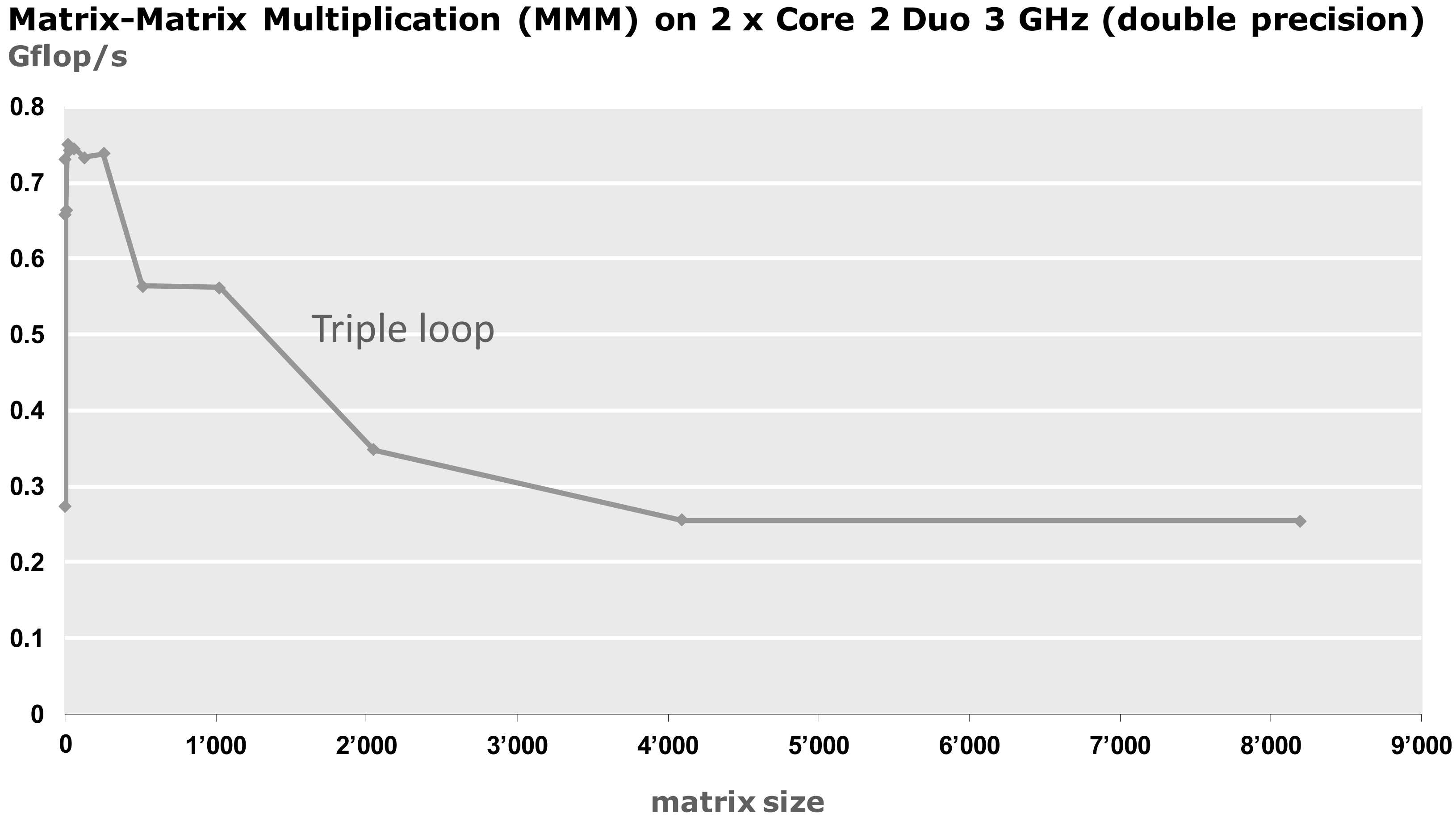
Example: matrix-matrix mutliplication

- Fundamental operation in ML, graphics, etc., etc.

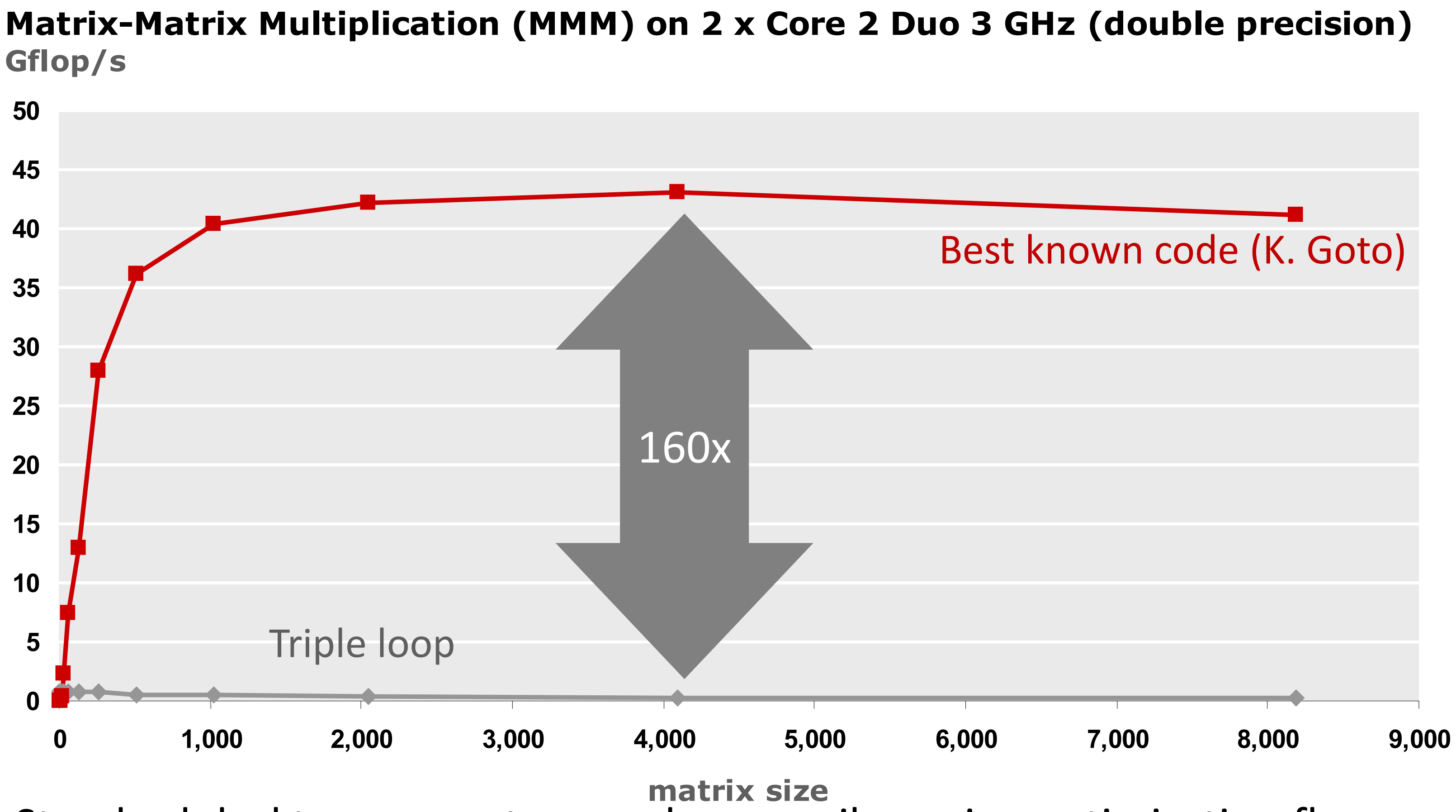
$$(\ldots) \leftarrow (\ldots) \times (\ldots)$$

- How complicated can this be?
- Basically requires n^3operations for $n \times n$ matrices
  - *Though some subcubic algorithms exist – ignore for now.*

Example: matrix-matrix multiplication

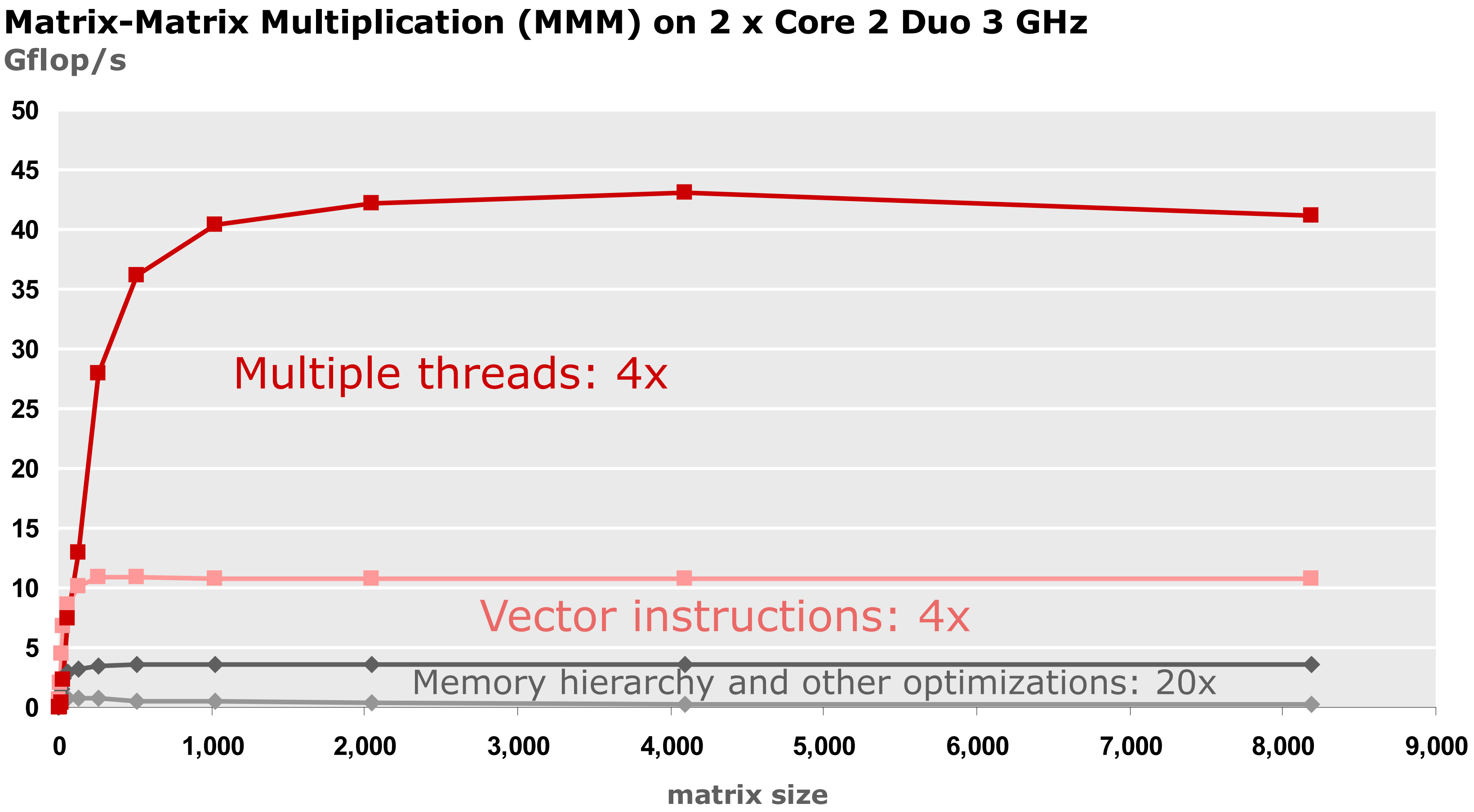


Example: matrix-matrix multiplication



- Standard desktop computer, vendor compiler, using optimization flags
- Both implementations have exactly the same operations count ($2n^3$)
- What is going on?

MMM plot: analysis



- Why? Blocking or tiling, loop unrolling, array scalarization, instruction scheduling, …
- ***Effect: less register spills, less L1/L2 cache misses, less TLB misses***

Inconvenient truth:

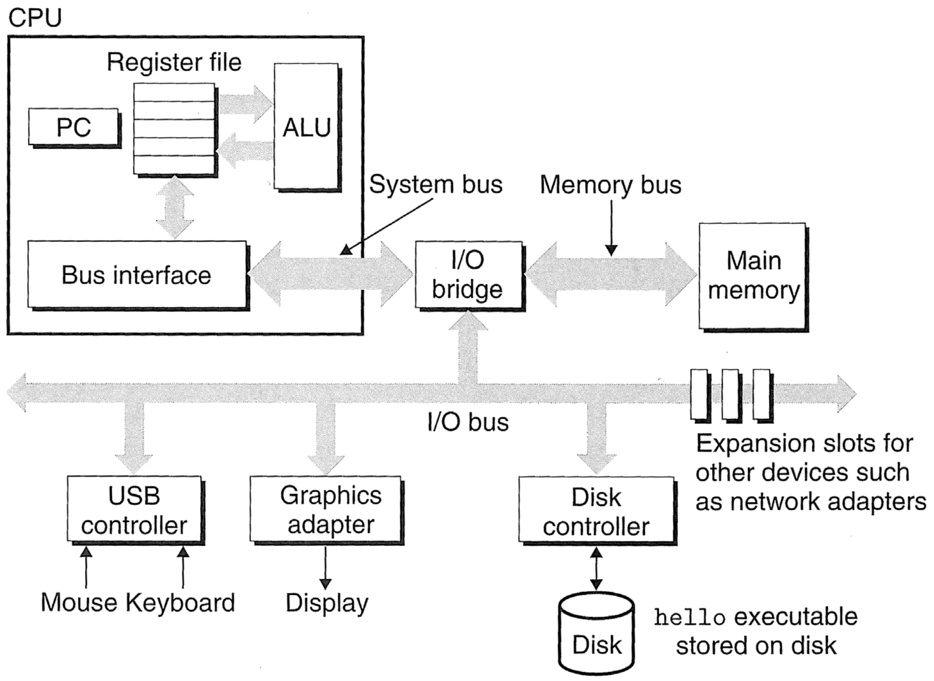
Computers don’t just execute programs
Programs don’t just calculate values

Computers don't just run programs

- They need to get data **in** and **out**
  - I/O critical to program reliability and performance
  - **Sense** the physical world
  - **Act** in the physical world
- They **communicate** over networks
  - Many system-level issues arise with a network
    - Concurrent operations by autonomous processes
    - Coping with unreliable media
    - Cross-platform interoperability
    - Complex performance issues

Lies our teachers tell us…

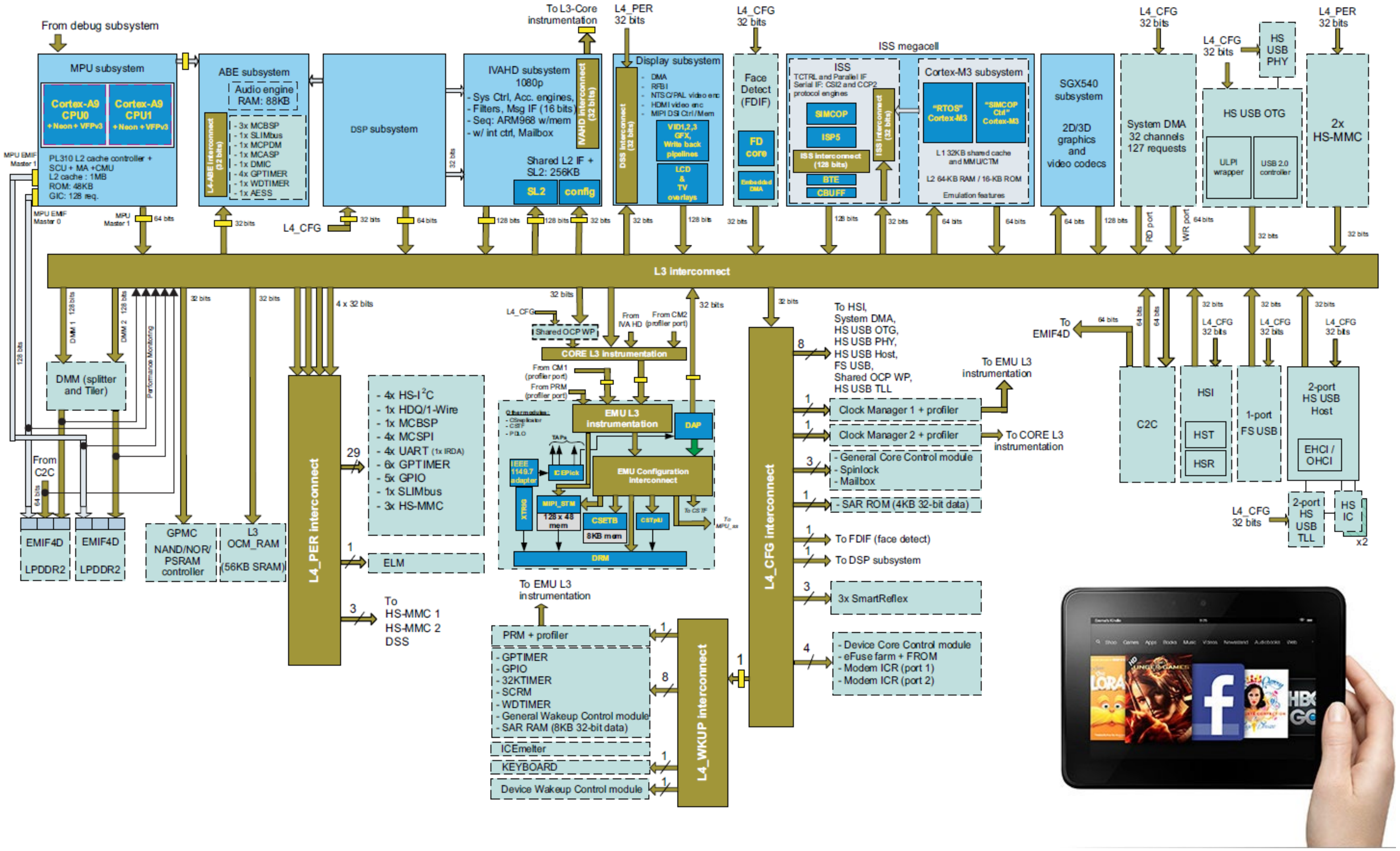
Figure 1.4
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.



systems, but all systems have a similar look and feel. Don't worry about the complexity of this figure just now. We will get to its various details in stages throughout the course of the book.

Computer Systems, A Programmer's Perspective, Bryant & O'Hallaron, 2011

A modern(ish) System-on-Chip



Texas Instruments OMAP 4460, c.2011

Inconvenient truth:

Programs are not semantic specifications

The role of “standards”

- Language standards aim to specify unambiguously what any program in the language does when compiled and executed.
  - Java: “write once, run anywhere”
  - Formal semantics
- The C standards should be viewed as rather different
  - Behavior frequently described as “*implementation dependent*”
  - What does this mean?

“Implementation defined”

“unspecified behavior where each implementation documents how the choice is made”

At least two options:

- Compiler is allowed to do anything, so optimizes out the code completely
- Compiler implements the most natural mapping to the target hardware and documents this.

WG14/N1256 Committee Draft — Septermber 7, 2007 **ISO/IEC 9899:TC3**

3. Terms, definitions, and symbols

For the purposes of this International Standard, the following definitions apply. Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to ISO/IEC 2382−1. Mathematical symbols not defined in this International Standard are to be interpreted according to ISO 31−11.

3.1
access
⟨execution-time action⟩ to read or modify the value of an object

NOTE 1 Where only one of these two actions is meant, "read" or "modify" is used.

NOTE 2 "Modify" includes the case where the new value being stored is the same as the previous value.

NOTE 3 Expressions that are not evaluated do not access objects.

3.2
alignment
requirement that objects of a particular type be located on storage boundaries with addresses that are particular multiples of a byte address

3.3
argument
actual argument
actual parameter (deprecated)
expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

3.4
behavior
external appearance or action

3.4.1
implementation-defined behavior
unspecified behavior where each implementation documents how the choice is made

EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

3.4.2
locale-specific behavior
behavior that depends on local conventions of nationality, culture, and language that each implementation documents

§3.4.2 General 3

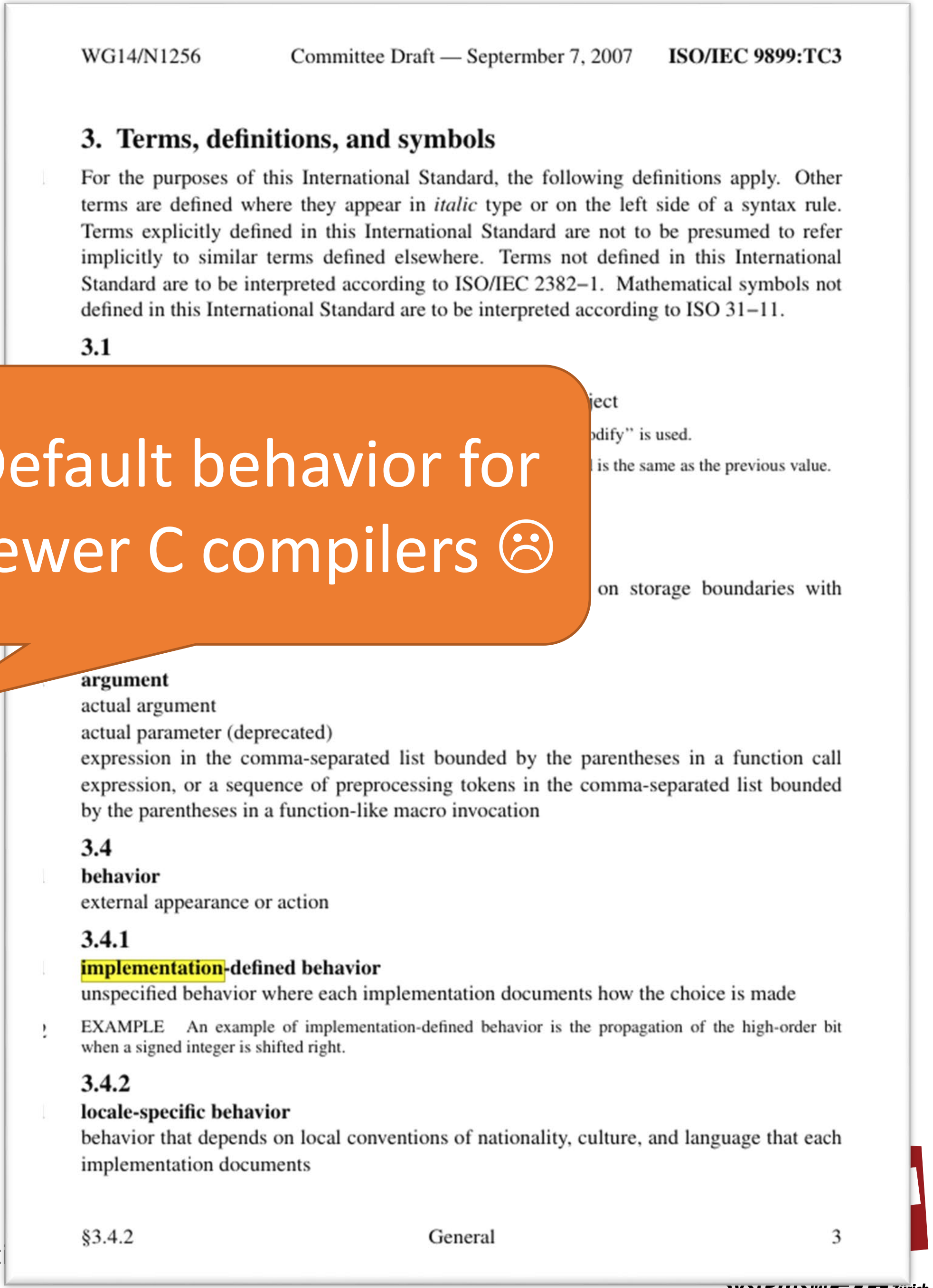
“Implementation defined”

“unspecified behavior where each implementation documents how the choice is made”

At least two interpretations:

- Compiler is allowed to do anything, so optimizes out the code completely
- Compiler implements the most natural mapping to the target hardware and documents this.

Default behavior for newer C compilers ☹


WG14/N1256 Committee Draft — Septermber 7, 2007 **ISO/IEC 9899:TC3**

3. Terms, definitions, and symbols

For the purposes of this International Standard, the following definitions apply. Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to ISO/IEC 2382−1. Mathematical symbols not defined in this International Standard are to be interpreted according to ISO 31−11.

3.1

ject

odify” is used.

is the same as the previous value.

on storage boundaries with

argument
actual argument
actual parameter (deprecated)
expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded by the parentheses in a function-like macro invocation

3.4
behavior
external appearance or action

3.4.1
implementation-defined behavior
unspecified behavior where each implementation documents how the choice is made

EXAMPLE An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

3.4.2
locale-specific behavior
behavior that depends on local conventions of nationality, culture, and language that each implementation documents

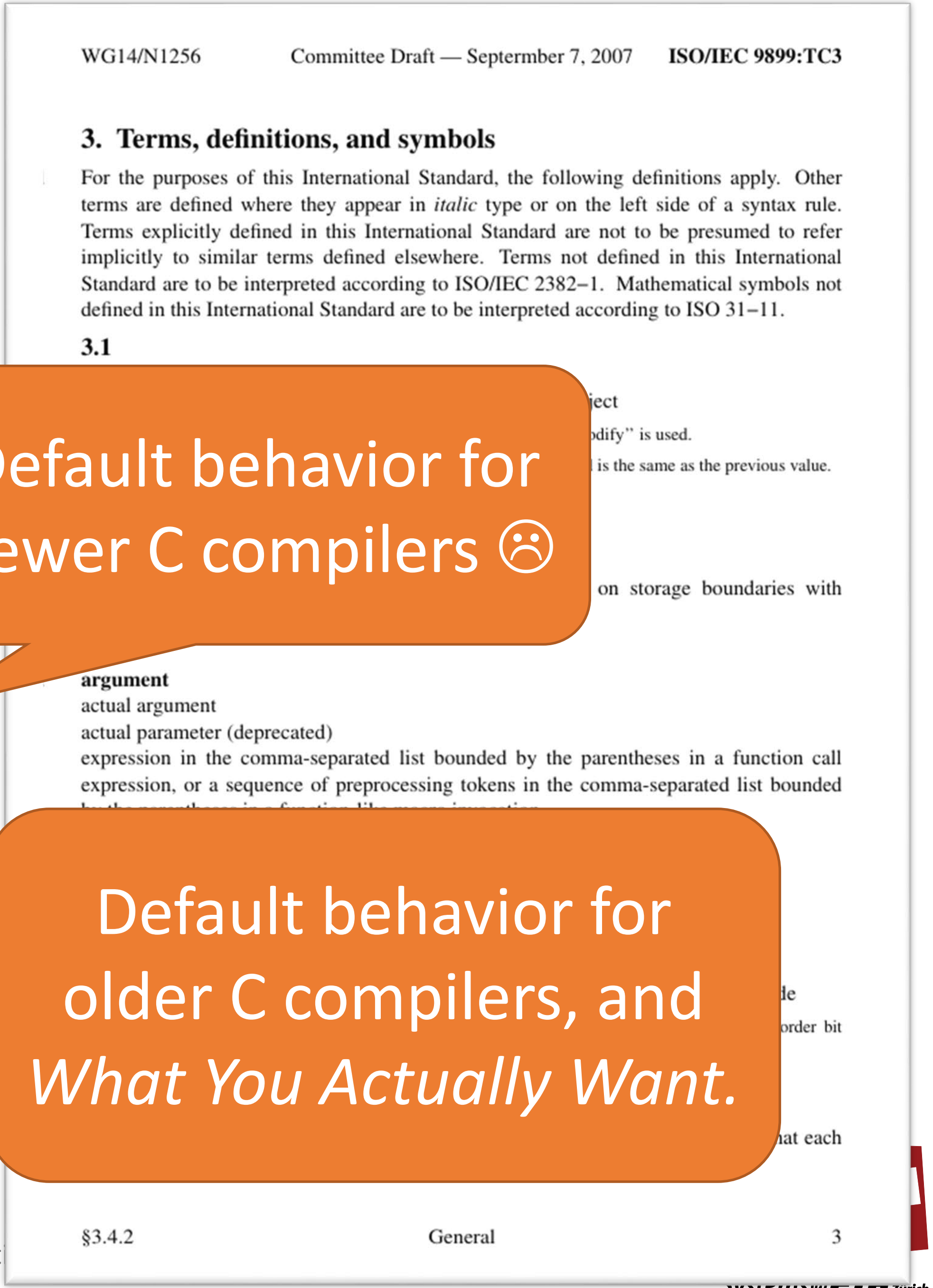
§3.4.2 General 3

“Implementation defined”

“unspecified behavior where each implementation documents how the choice is made”

At least two interpretations:

- Compiler is allowed to do anything, so optimizes out the code completely
- Compiler implements the most natural mapping to the target hardware and documents this.


WG14/N1256 Committee Draft — Septermber 7, 2007 ISO/IEC 9899:TC3

3. Terms, definitions, and symbols

For the purposes of this International Standard, the following definitions apply. Other terms are defined where they appear in *italic* type or on the left side of a syntax rule. Terms explicitly defined in this International Standard are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not defined in this International Standard are to be interpreted according to ISO/IEC 2382−1. Mathematical symbols not defined in this International Standard are to be interpreted according to ISO 31−11.

3.1

argument
actual argument
actual parameter (deprecated)
expression in the comma-separated list bounded by the parentheses in a function call expression, or a sequence of preprocessing tokens in the comma-separated list bounded

§3.4.2 General 3

Default behavior for newer C compilers ☹

Default behavior for older C compilers, and *What You Actually Want.*

The role of "standards"

- Language standards aim to specify unambiguosly what any program in the language does when compiled and executed.
  - Java: "write once, run anywhere"
  - Formal semantics
- The C standards should be viewed as rather different
  - Behavior frequently described as "*implementation dependent*"
  - What does this mean?

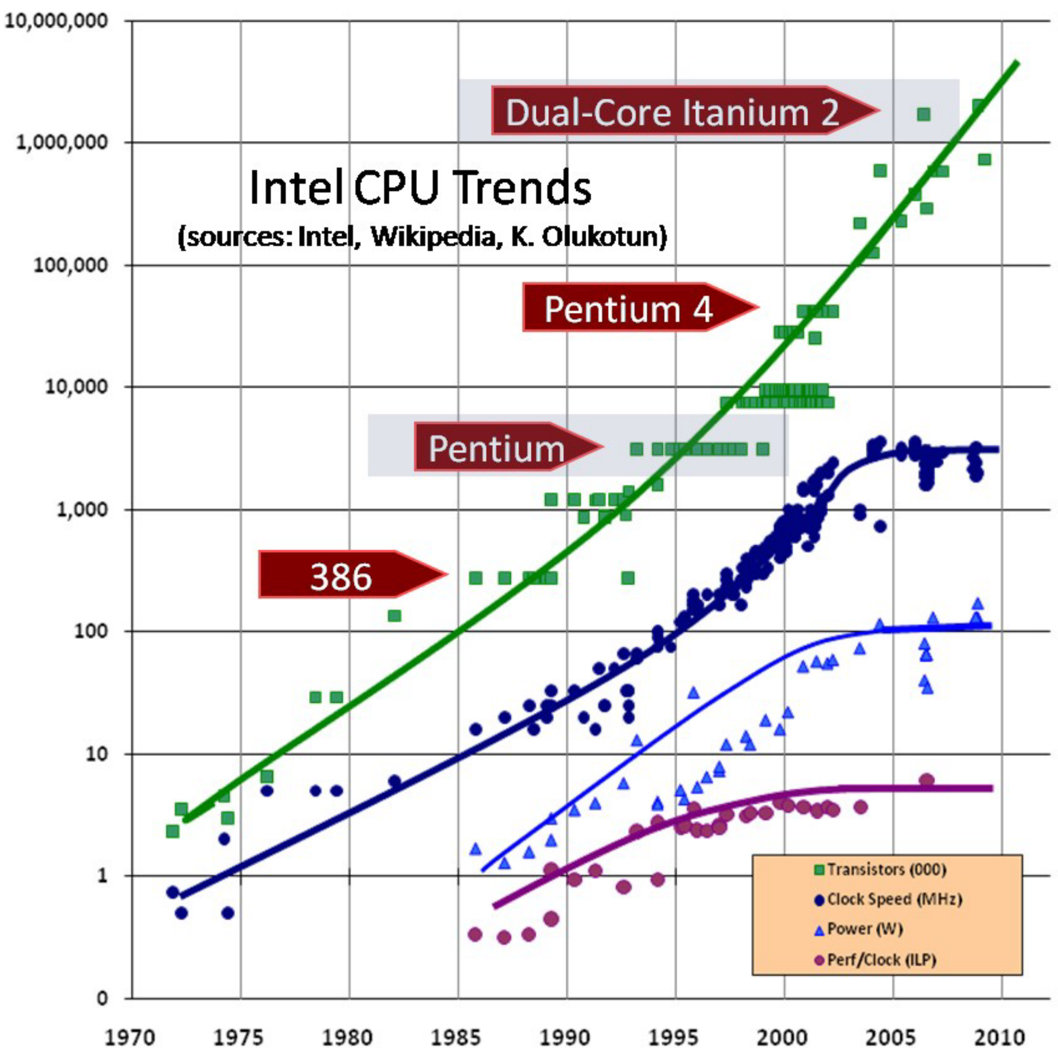
A program is a set of instructions to a compiler that tell it what assembly language to generate.

Summary

1. `int`s are not integers. `float`s are not real numbers.
2. You've got to know assembly.
3. Memory matters. RAM is an unrealistic abstraction.
4. There's much more to performance than asymptotic complexity.
5. Computers don't just evaluate programs.
6. Programs are not complete semantic specifications.

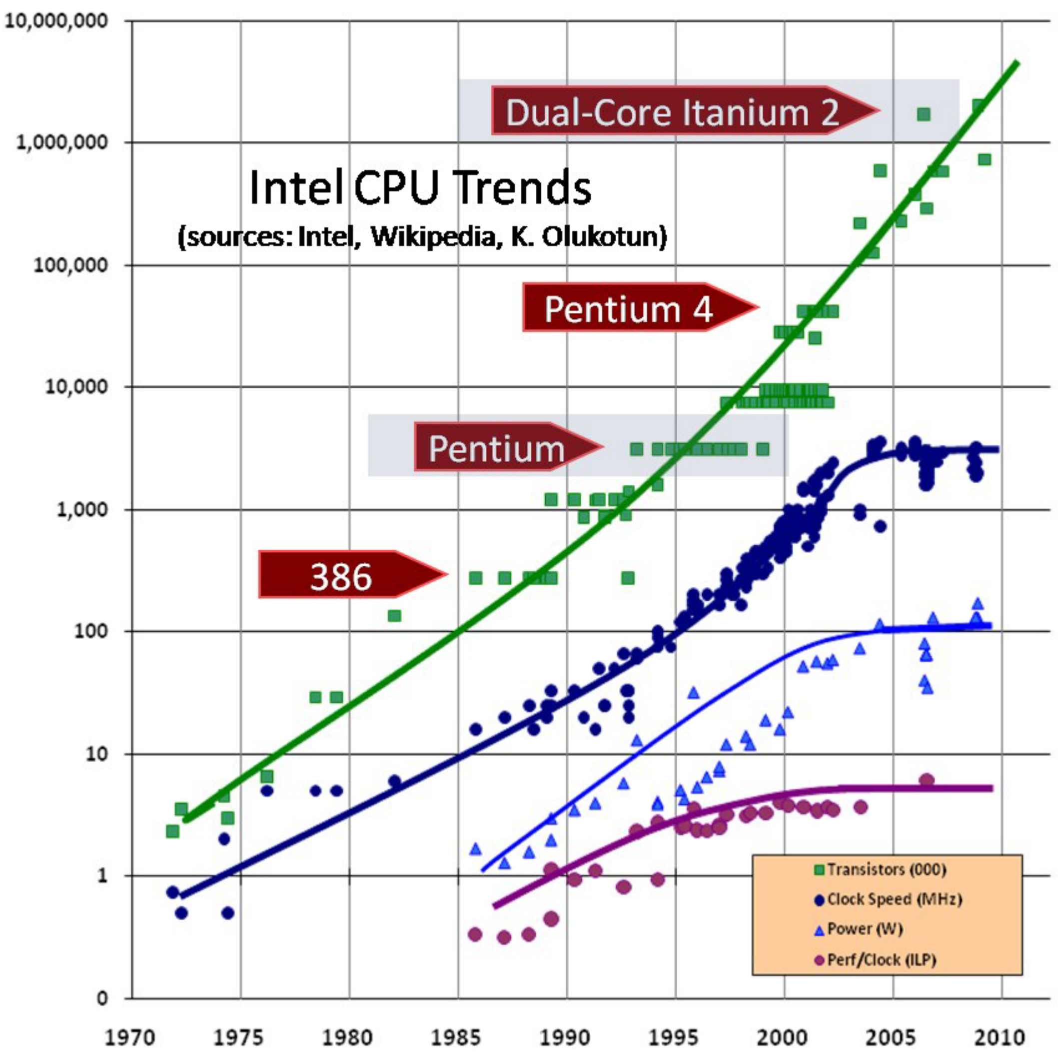
Interesting times…

- Processors are not getting faster.
- Performance-wise, progress in computer architecture has halted.
  - In fact, it's going backwards due to security concerns.



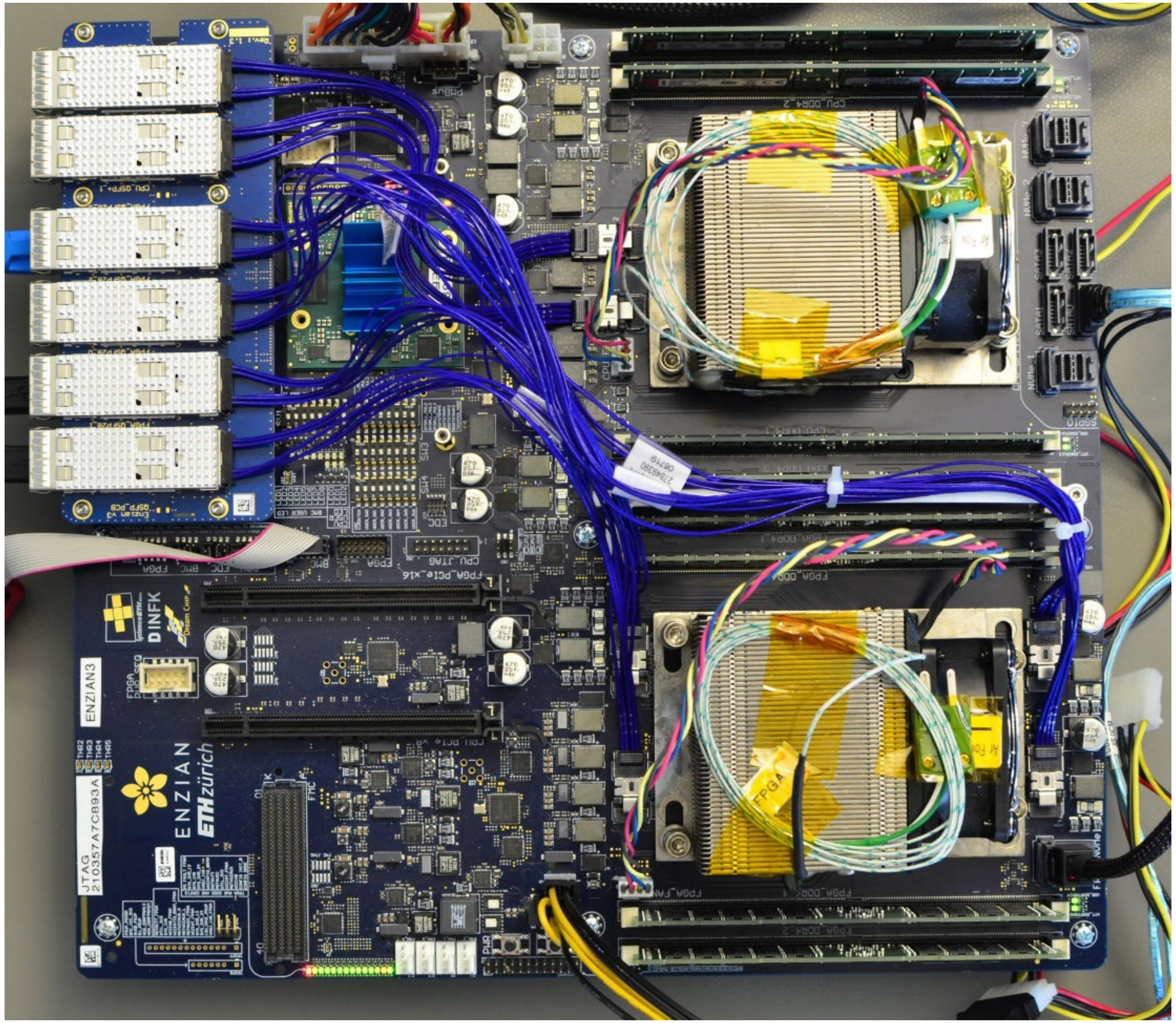
Interesting times…

- Lots of companies, universities, etc. are building new kinds of computers.
- How can these be programmed?
- This is a systems software problem.



Computers are looking different!

- Cavium / Marvell ThunderX-1 NP:
  - 48 x ARMv8 cores at 2GHz
  - 128 GB DDR4
  - 2 x 40Gb/s network
- Xilinx UltraScale+ VU9P
  - 512 GB DDR4
  - 4 x 100 Gb/s network
- NVMe, SATA, PCIe on both sides
- Native coherence between FPGA and CPU



ENZIAN

Systems@ETH zürich

1.4: What we'll assume you know

Systems Programming and Computer Architecture

Courses already

- Programming & software engineering
- Parallel programming
- Data structures and algorithms
- Digital Circuits
- Discrete Mathematics

What we’ll assume you know #1:

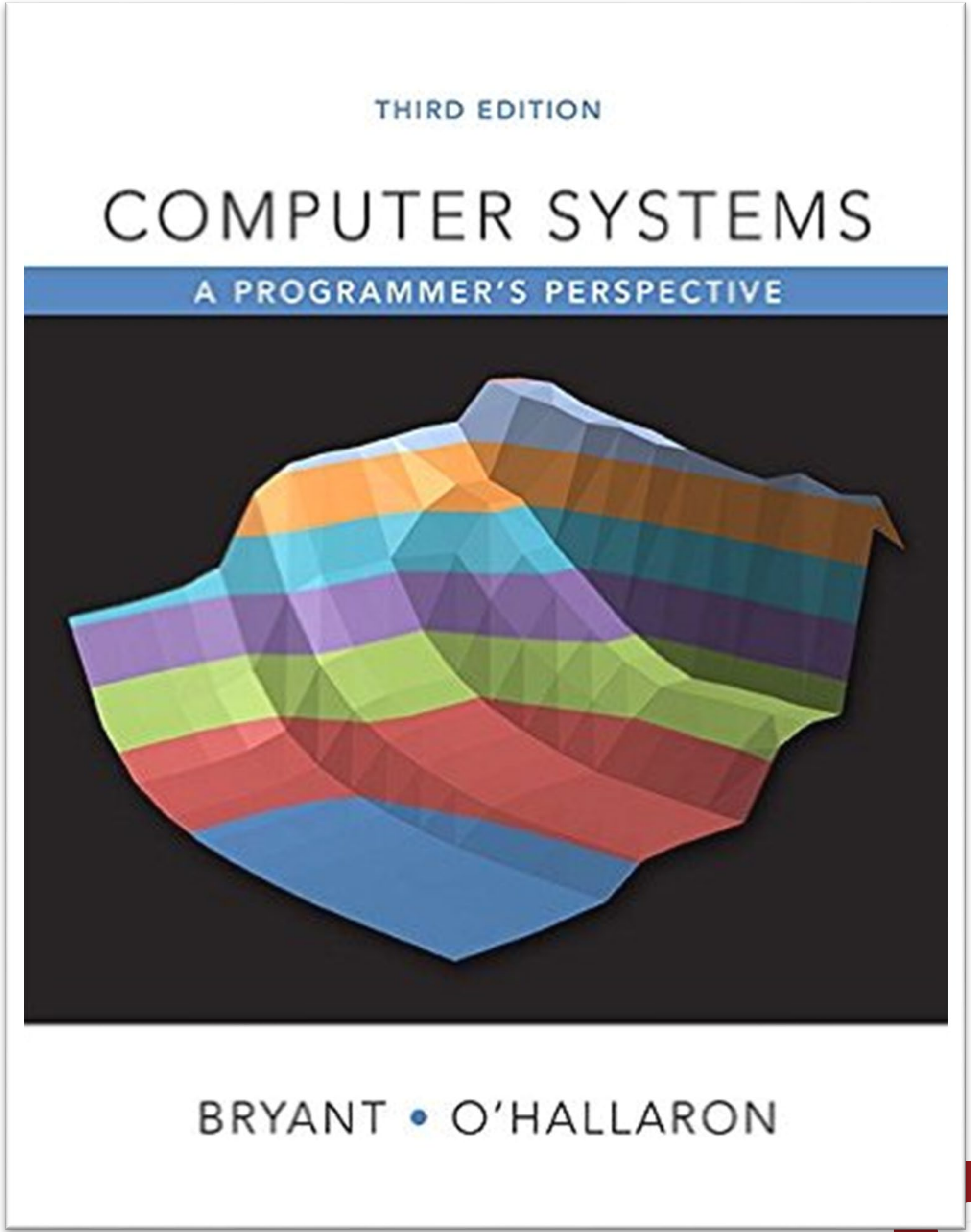
- Memory, addresses, bytes, and words
  - Binary, and Hexadecimal notation
  - Byte-ordering (Big/Little Endian)
- Boolean algebra
  - and, or, not, xor
  - Generalized Boolean algebra: bitwise operations on words as bit vectors
- How to write programs
- Languages
  - Java (but we won’t use it)
  - Some C or C++ (but not much)
  - Some assembly language

What we'll assume you know #2:

- Processor architecture, pipelines
  - Registers
  - Addressing modes
  - Instruction formats
- Basic memory systems
  - cache architectures
  - virtual memory
  - I/O devices
- Software engineering
  - Object-orientation
  - Design-by-contract
  - Strong typing
- Concurrency and parallelism
  - Threads
  - Locks, mutexes, condition variables
  - Parallel programming constructs

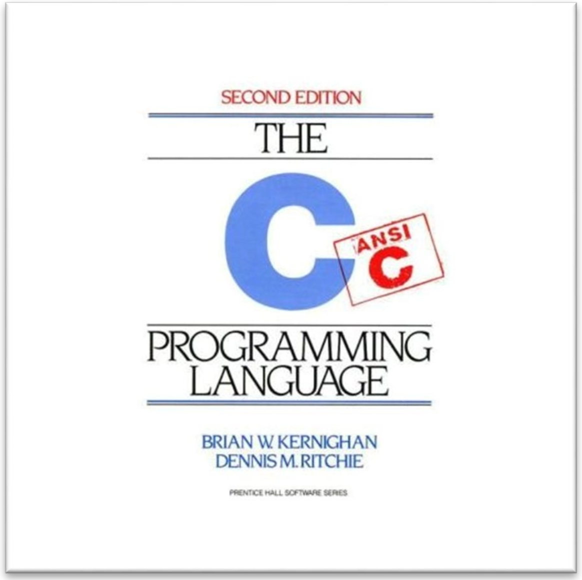
Textbooks

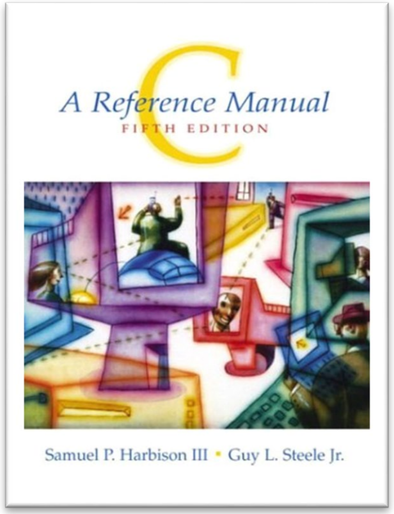
- Randal E. Bryant and David R. O'Hallaron,
  - Computer Systems: A Programmer's Perspective, Third Edition (CS:APP3e), Pearson, 2016
  - http://csapp.cs.cmu.edu



Books on C (there are many)

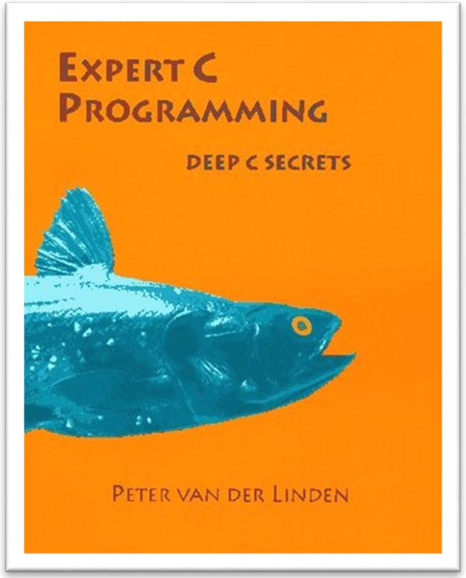
- Brian Kernighan and Dennis Ritchie,
  - “The C Programming Language, Second Edition”, Prentice Hall, 1988(!)





- Samuel Harbison and Guy Steele
  - C: A Reference Manual
  - 5th edition 2002

- Peter van der Linden
  - Expert C Programming: Deep C Secrets, 1994



OK, let’s start