B4M36ESW: Efficient software

Lecture 1: Introduction

Michal Sojka
michal.sojka@cvut.cz

CZECH TECHNICAL UNIVERSITY IN PRAGUE

February 17, 2025

Outline

About this course

https://esw.pages.fel.cvut.cz/

Teachers

Michal Sojka C/C++ (or Rust), embedded systems, operating systems
David Šišlák Java, servers, ...

Scope

- Writing fast programs
- Single (multi-core) computer, no distributed systems/cloud
- Interaction between software and **hardware**
- Programming languages: no runtime (C/C++/Rust), with runtime (Java)
- How general concepts apply to programs in different programming languages i.e. how to use **hardware** efficiently from C/C++/Rust and Java
- The course is not about comparing C/C++/Rust with Java,
  - but you should be able to make this comparison yourself at the end.

Grading

- Exercises
  - 8 small tasks
  - semestral work (implemented in any programming language)
  - Maximum 60 points
  - **Minimum 30 points**
- Exam
  - Written test: max. 30 points
  - Voluntary oral exam: 10 points
  - **Minimum: 20 points**

Lectures

- Slides accompany lectures, they are not self-standing documents.
- We would like to get your feedback:
  - questions (even stupid)
  - typos notifications
  - error reports
  - ...

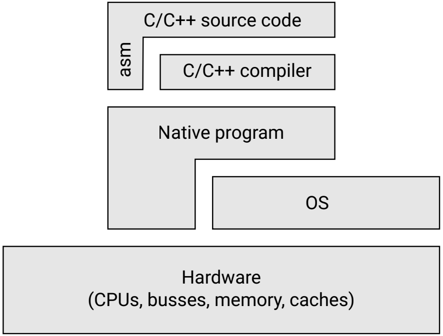
  Please use the feedback link on every slide or file the issue directly at `https://gitlab.fel.cvut.cz/esw/lectures/issues/new`.

Outline

Efficient software

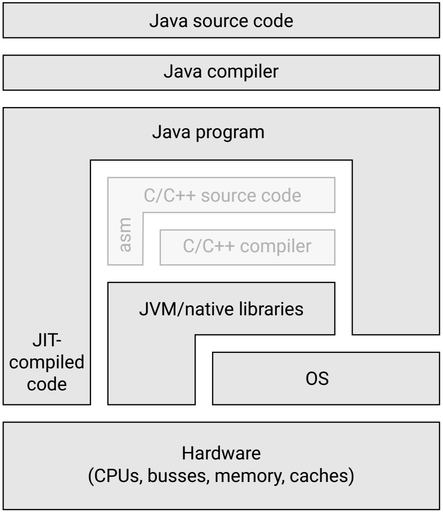
- There is no theory of how to write efficient software
- Writing efficient software is about:
    - Knowledge of all layers involved
    - Experience in knowing when and how performance can be a problem
    - Skill in detecting and zooming in on the problems
    - A good dose of common sense
- Best practices
    - Patterns that occur regularly
    - Typical mistakes

Layers involved in software execution



- In the end, everything is executed by hardware
  - Majority of this course is about how to tailor the code to use the hardware efficiently
- C/C++ source code is transformed into native (machine) code by the compiler
  - Compiler tries to optimize the generated code
  - Optimizations are often only heuristics
- Native code is executed directly by HW or invokes OS services

Layers involved in software execution



- Java source code is also compiled
- Java program can execute:
  - interpreted by Java Virtual Machine (JVM) or
  - natively after being just-in-time (JIT) compiled by JVM (AOT compilation also possible)
- JVM is a native program
- Java program can use native libraries (JNI)
- ... long way from source code to execution on HW

Fundamental theorem of software engineering

All problems in computer science can be solved by another level of indirection

... except for the problem of too many layers of indirection.

—David Wheeler

Layers of indirection in today's systems

Hardware

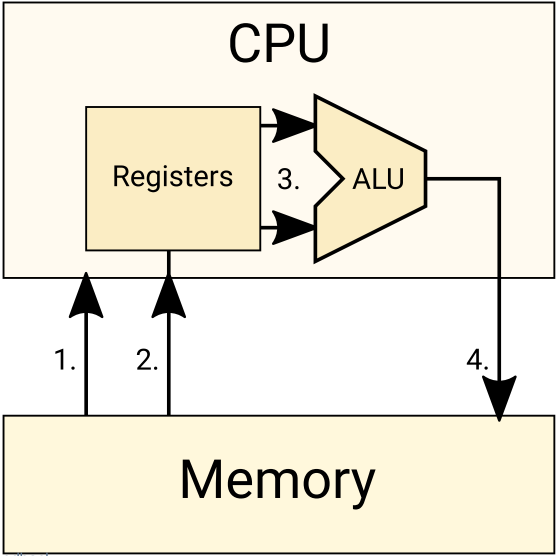
- microcode, ISA
- virtual memory, MMU
- buses, arbiters

Software

- operating system kernel
- compiler
- language runtime
- application frameworks

Outline

CPU – principle of operation



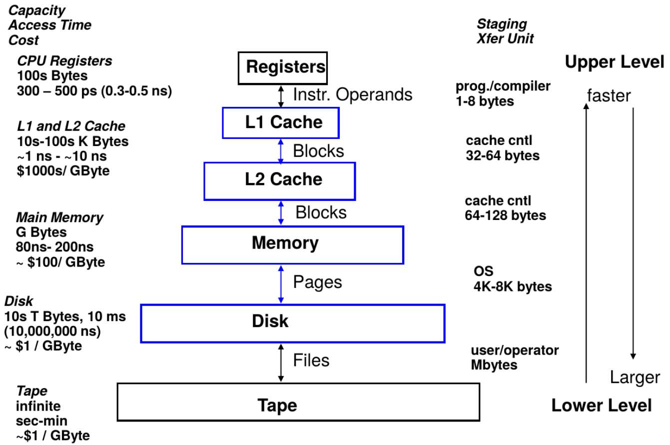
1. Fetch instruction from memory
2. Fetch data from memory
3. Perform computation
4. Store the result to memory (or register)

C code and machine code

```
int a, b, r;
void func() {
  r = a + b;
}
```

```
mov 0x100,%eax   ; load a
add 0x104,%eax   ; add b
mov %eax,0x108   ; store r
```

Memory



- Source of many performance problems in today's computers
- Reason: Memory is slow compared to CPUs!
- Solution: Caching $\Rightarrow$ memory hierarchy

Latencies in computer systems

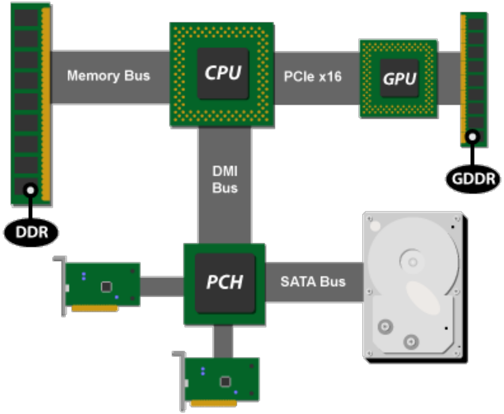
Event	Latency	Scaled
1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access (DRAM, from CPU)	120 ns	6 min
Solid-state disk I/O (flash memory)	50–150 μs	2–6 days
Rotational disk I/O	1–10 ms	1–12 months
Internet: San Francisco to New York	40 ms	4 years
Internet: San Francisco to United Kingdom	81 ms	8 years
Internet: San Francisco to Australia	183 ms	19 years
TCP packet retransmit	1–3 s	105–117 years
OS virtualization (container) system reboot	4 s	423 years
SCSI command timeout	30 s	3 millennia
HW virtualization system reboot	40 s	4 millennia
Physical server system reboot	5 m	32 millenia

Computer performance and laws of physics

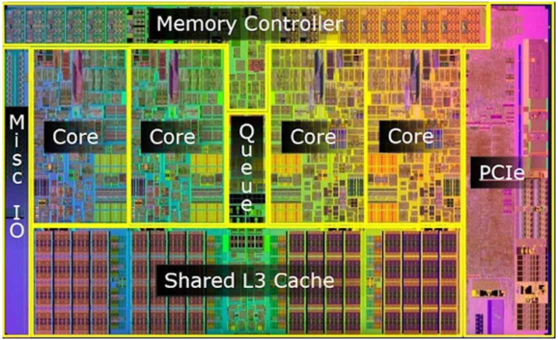
What distance does light travel in vacuum during a clock cycle of a 3 GHz CPU?

- 10 cm
- Speed of light in silicon is even slower.
- Each gate delays the electric signal a bit.
- It’s already difficult to pass the information quickly from one side of the chip to another.
- The layers between source code and hardware make it difficult to understand how the hardware is actually “used”.

Example: Intel-based system (single socket, 2009)



Intel's P55 platform
Source: ArsTechnica



Lynnfield CPU
Source: Intel

Outline

Making the hardware faster

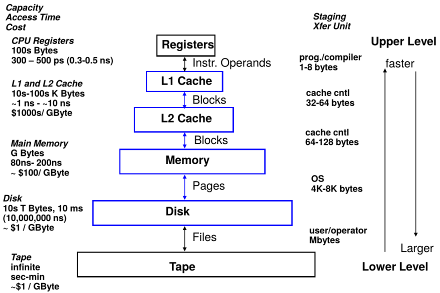
... and more tricky to use efficiently from software

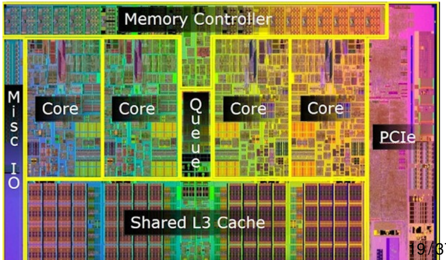
- Hardware designers intensively optimize their hardware
- These optimizations improve performance in common (average) cases
- Using the HW in “uncommon” ways can drastically degrade the performance
- The layers between source code and hardware complicate understanding how is the hardware actually “used”
- What are the features that can be problematic from performance point of view?
- We will look at them in more detail in the rest of the lectures.

Outline

Caches

- Principle
  - Smaller but faster memory
  - Take advantage of spacial and temporal locality of memory accesses performed by the code.
- Problems
  - Random Access Memory (RAM) is no longer RAM from performance point of view
  - Management of multiple copies of a single data...
    (known as **cache coherence**)



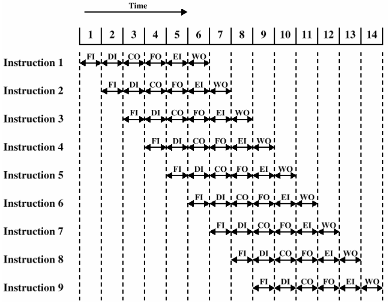


Outline

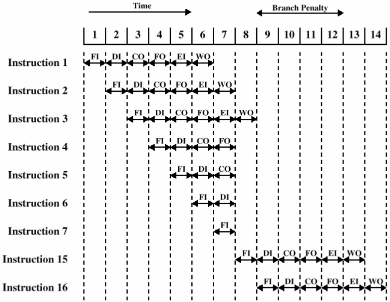
Pipelining, branch prediction

Branch = if then/else

Branch not taken



Branch taken



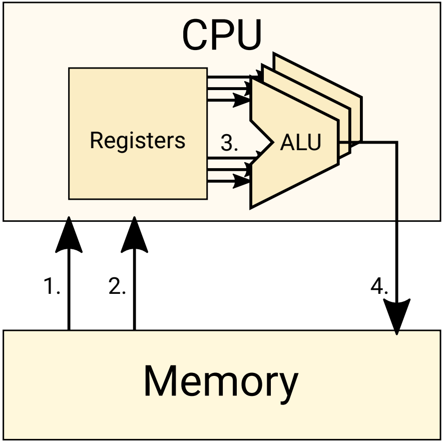
Example pipeline stages:

1. FI = Fetch instruction
2. DI = Decode instruction
3. CO = Calculate operands
4. FO = Fetch operands
5. EI = Execute instruction
6. WO = Write output (result)

- The **branch predictor** tries to predict the branch target and condition value (true/false)
- If it fails, we pay branch penalty
- Here, the branch penalty is a *few cycles*, but it is much more severe in case of superscalar CPUs.

Superscalar CPUs

HW tries to execute several instructions in parallel.



Instruction stream

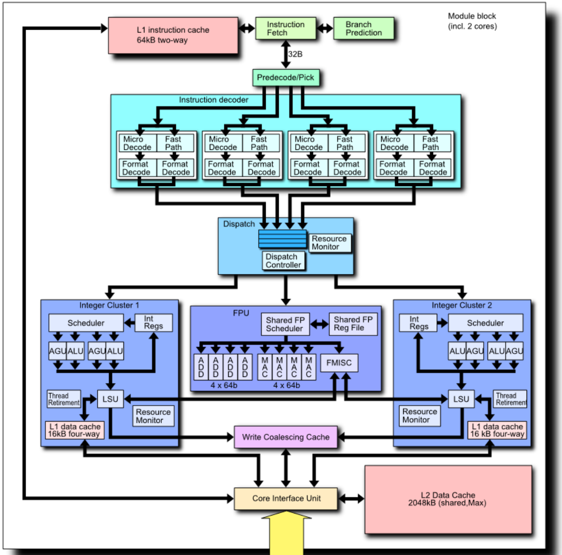
r = a + b
s = c + d
t = e + f
u = g + h
v = u + i

Superscalar execution

r = a + b; s = c + d; t = e + f
u = g + h
v = u + i

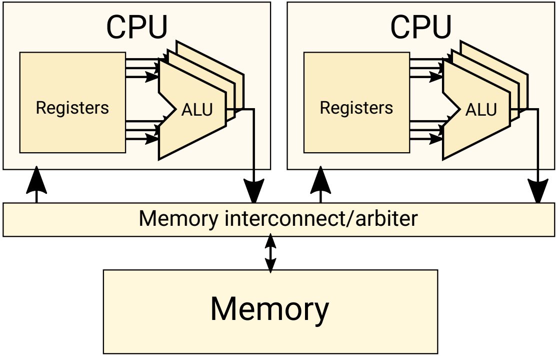
- Efficient SW goal: Order instructions in a program to use all execution units (e.g. ALUs) in parallel
  - Task for the compiler
  - Complicates reading of assembler (and debugging)

Example: AMD Bulldozer CPU



Outline

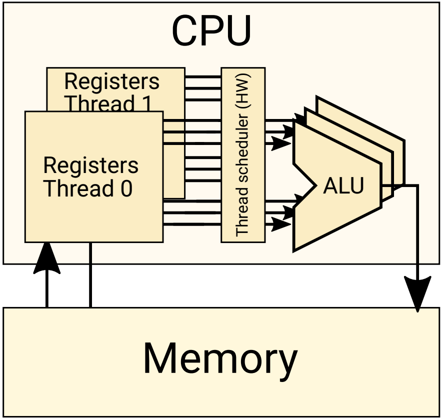
Multiple CPUs



- Computers usually run multiple programs simultaneously
- Let's execute them simultaneously on two CPUs
- The CPUs can be on
  - single chip $\Rightarrow$ multi-core
  - multiple chips $\Rightarrow$ multi-socket
- Performance problems: synchronization
  - Communication between the cores (via shared cache or memory interconnect) is slow
  - What we mean by communication?
  - Access to shared data in the memory. Examples:
    - Mutex – e.g. to ensure mutually exclusive access to shared data structure in memory
    - `synchronized` keyword in Java
    - ...

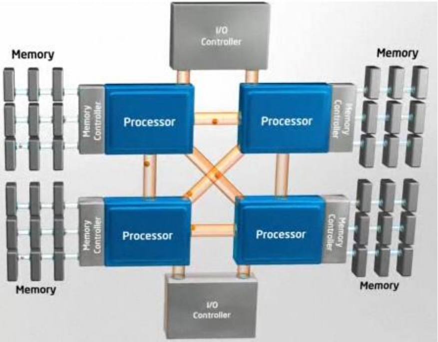
Simultaneous multi-threading (SMT)

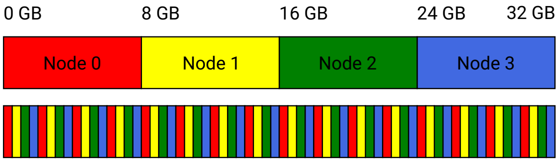
Hyper-threaded CPU



- “Cheaper variant” of parallelism
- Duplicate just the registers, not the execution units (ALU)
- Add a HW scheduler to simulate parallel execution
- When one HW thread waits for memory, the other can execute
- From SW point of view, SMT looks like a multi-core CPU
- Imperfect instruction-level parallelism (superscalar CPU) is improved by task-parallelism
- Hyper-threading is not popular today due to recent security related HW bugs.

Non-Uniform Memory Access (NUMA)





- Multi-socket system
- Each socket has locally connected memory
- Other sockets access the memory via inter-socket interconnects (slower, ca 15%)
- Software sees all memory
- SW (OS) should allocate memory local to the CPU where it runs, apps could help

←Two possible mappings of memory addresses to memory nodes

Out-of-order execution

Instruction stream

r = a + b
s = c + d
t = e + f
u = g + h
v = **u** + i

a and c are not cached, the rest is:

Superscalar, out-of-order execution

t = e + f; **u** = g + h
r = a + b; s = c + d; v = **u** + i

From a single CPU point of view, everything is correct.

- Complicates synchronization
- Other CPUs can see results of computations in different order

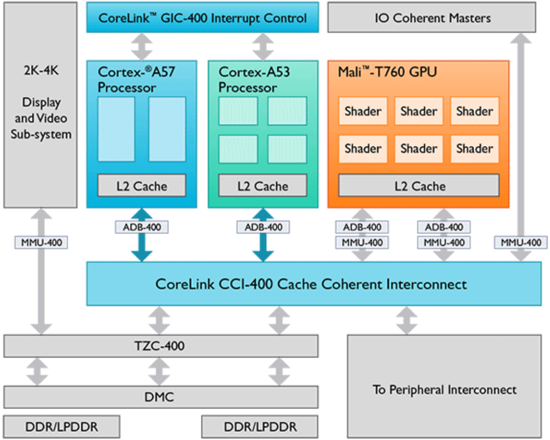
When order matters?

lock = 1
r = a + b
s = a - b
lock = 0

The above example will likely not work, because accesses to "lock" may be reordered.

Embedded heterogeneous systems

Different CPUs/GPUs on a single chip



Source: ARM

Outline

Energy efficient software

- Today, we no longer want just fast software
- We also care about heating and battery life of our mobile phones
- Good news: Fast software is also energy efficient

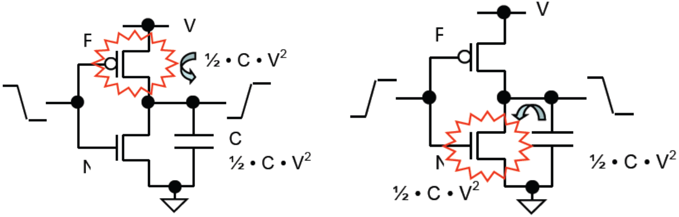
Power consumption of CMOS circuits

Two components:

- Static dissipation
  - leakage current through P-N junctions etc.
  - higher voltage $\rightarrow$ higher static dissipation
- Dynamic dissipation
  - charging and discharging of load capacitance (useful + parasitic)
  - short-circuit current

$$P_{total} = P_{static} + P_{dyn}$$

Dynamic power consumption and gate delay



Charging the parasite capacities needs energy

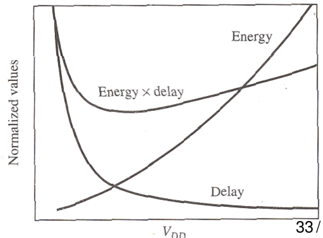
Power consumption

$P_{dyn} = a \cdot C \cdot V_{dd}{}^2 \cdot f$

Gate delay

$t = \frac{\gamma \cdot C \cdot V_{dd}}{(V_{dd} - V_T)^2} \approx \frac{1}{V_{dd}}$

Low power $\Rightarrow$ slow



Methods to reduce power/energy consumption

- use better technology/smaller gates (HW engineers)
- use better placing and routing on the chip (HW engineers)
- reduce power supply V_{DD} and/or frequency = Dynamic voltage and frequency scaling (OS job – apps can help)
  - raising it back takes time (ramp-up latency)
  - deciding optimal sleep state to take requires knowing the future
  - recent Android versions have API for “predicting future”
- reduce activity (clock gating = switch off parts of the chip that are not used) [job for OS and HW, apps can help]
- use better algorithms and/or data structures (SW engineers)

Outline

C/C++ compiler

- Generates native code from C/C++ source code
- Popular compilers: GCC, Clang (LLVM), icc, MSVC, ...
- Perform many “optimization passes”
  - Those will be covered in a separate lecture
- For now, very brief overview of what you might need today

Compiler flags (gcc, clang)

- Documentation is your friend:
  - Command `(p)info gcc`
  - `https://gcc.gnu.org/onlinedocs/`
  - Clang's flags are mostly compatible with gcc
- Generate debugging information: `-g`
- Optimization level: `-O0`, `-O1`, `-O2`, `-O3`, `-Os` (size), `-Og` (debugging)
  - `-O2` is considered "safe", `-O3` may be buggy
  - Individual optimization passes: `-free-ccp`, `-fast-math`, `-fomit-frame-pointer`, `-free-vectorize`, ...
  - Find out which optimizations passes are active for given optimization level: `g++ -Q -O2 --help=optimizers`
- Code generation
  - `-fpic`, `-fpack-struct`, `-fshort-enums`
  - Machine dependent:
    - Generate instructions for given micro-architecture: `-march=haswell`, `-march=skylake` (will not run on older hardware)
    - Use only "older" instructions, but schedule them for the given µarch: `-mtune=haswell`, `-mtune=native`,
    - `-m32`, `-minline-all-stringops`, ...