

Automata & Complexity Theory

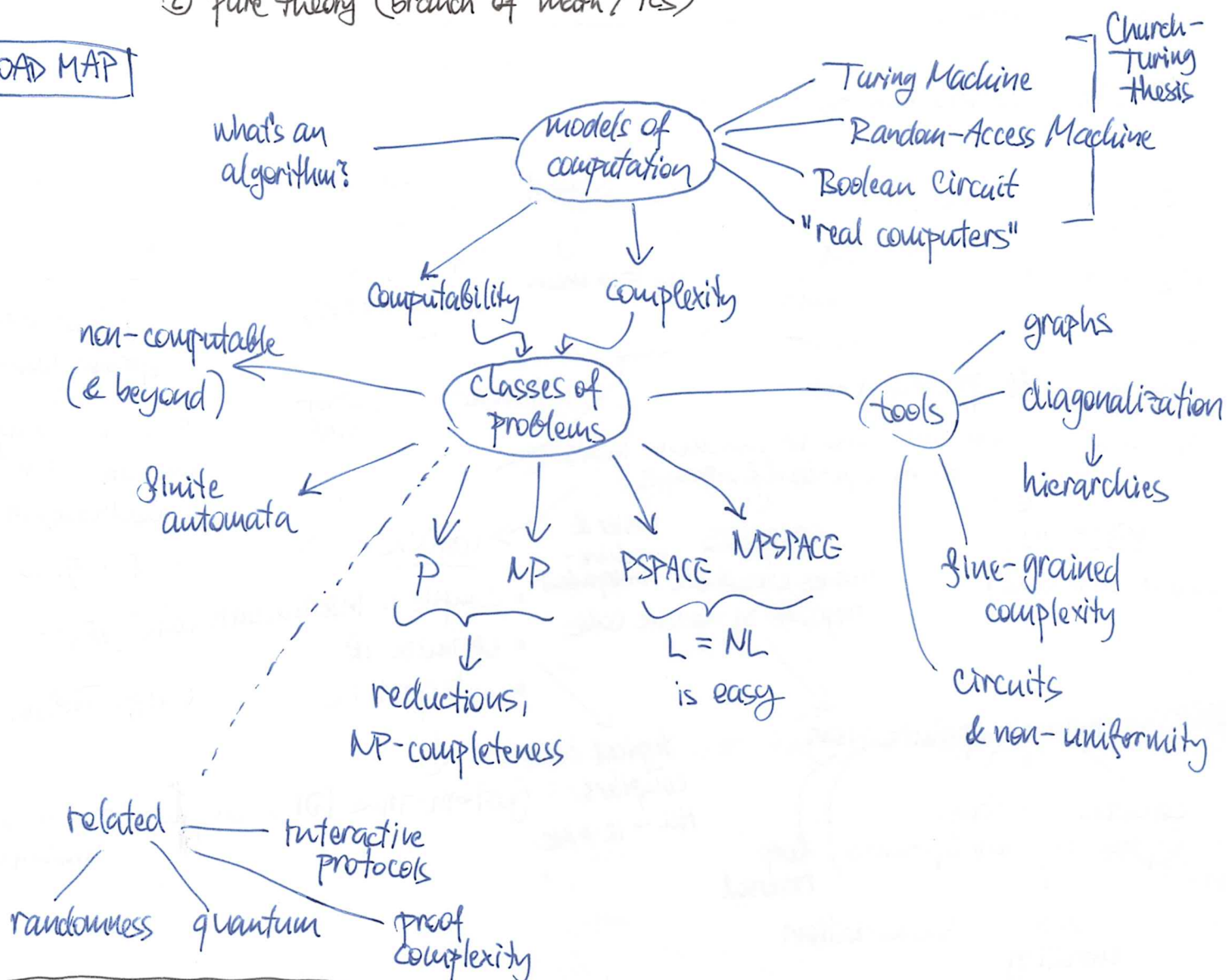
winter 2022 2023 2026

(1)

goal: build theory of computation & hardness of problems

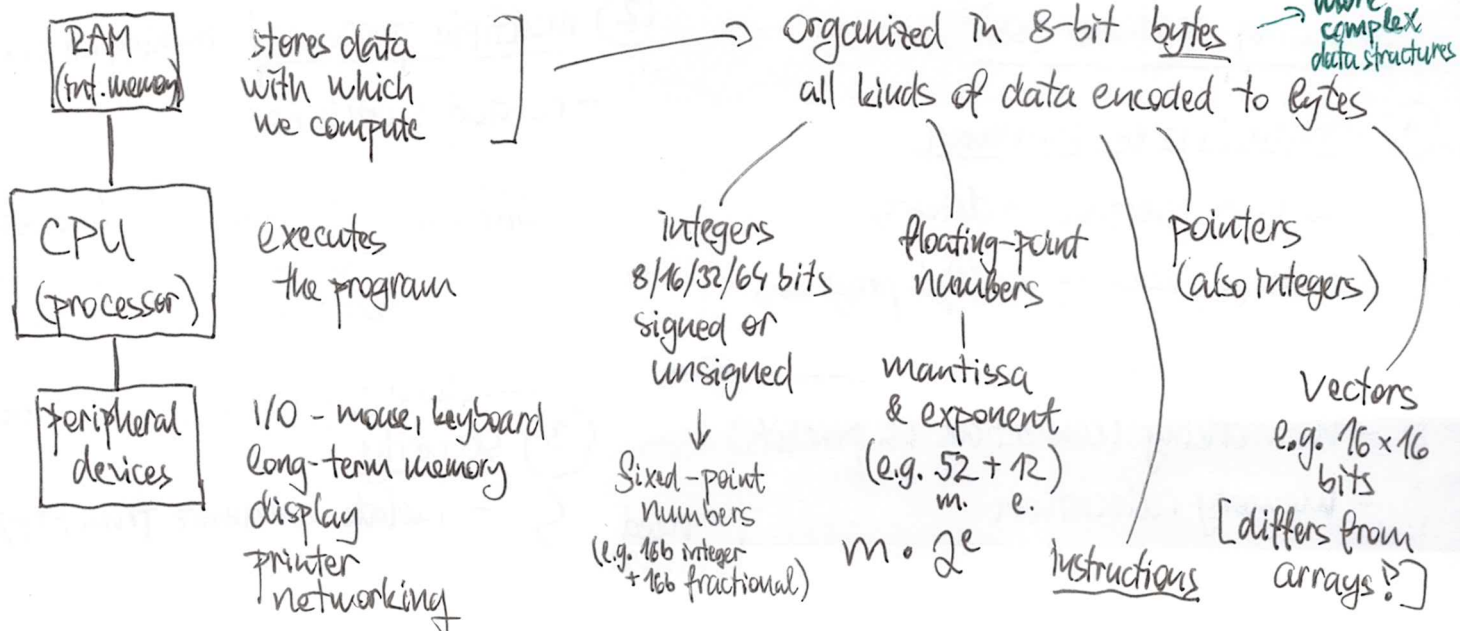
two views: ① an applied theory to help us use machines more efficiently
② pure theory (branch of math/TCS)

ROAD MAP



PHYSICAL COMPUTERS

& their architecture (typical case in 2022, just sketching)



machine instructions (program)

- stored in the same memory as data (Von Neumann architecture) (2)
- can modify itself
- just another interpretation of bytes

↓
they often work with several simple pieces of data (e.g., 64b numbers)

machine words

("64-bit CPU")

registers

inside CPU

(10s of them)

see example program ...

We seldom write programs in machine insns but in high-level languages

interpreters

(e.g., UNIX shell)

compilers

produce executable programs in machine code

easier & machine-independent

combined

- compile to intermediate code (IR)
- optimize IR
- interpret IR

e.g. Python

↓
just-in-time (JIT) compilers - e.g. Java, JavaScript

automatic optimization

- constant expressions
- common sub-expressions
- loop reversal
- loop unrolling
- vectorization
- & many others

typical compiler:
HLL → IR → MC

types of instructions

data movement

reg → reg

reg ↔ mem

arithmetical & logical

(+ - * / %

AND OR XOR NOT)

<< >>

integer

floating-point

control (jumps)

conditions test

flags

(e.g. "last arithmetic operation gave zero")

stack for storing temp. data & return addresses in RAM

SP register

program flow, PC register

Operating systems (OS)

① abstraction of hardware

- common interface → drivers
- manage access by multiple programs
- file systems (files, directories, mounts...)
- networking (connections vs. packets)
- memory allocation

② multiple processes "running at once"

- context switching

sleeping

yielding

timeslices

(pre-emptive multitasking)

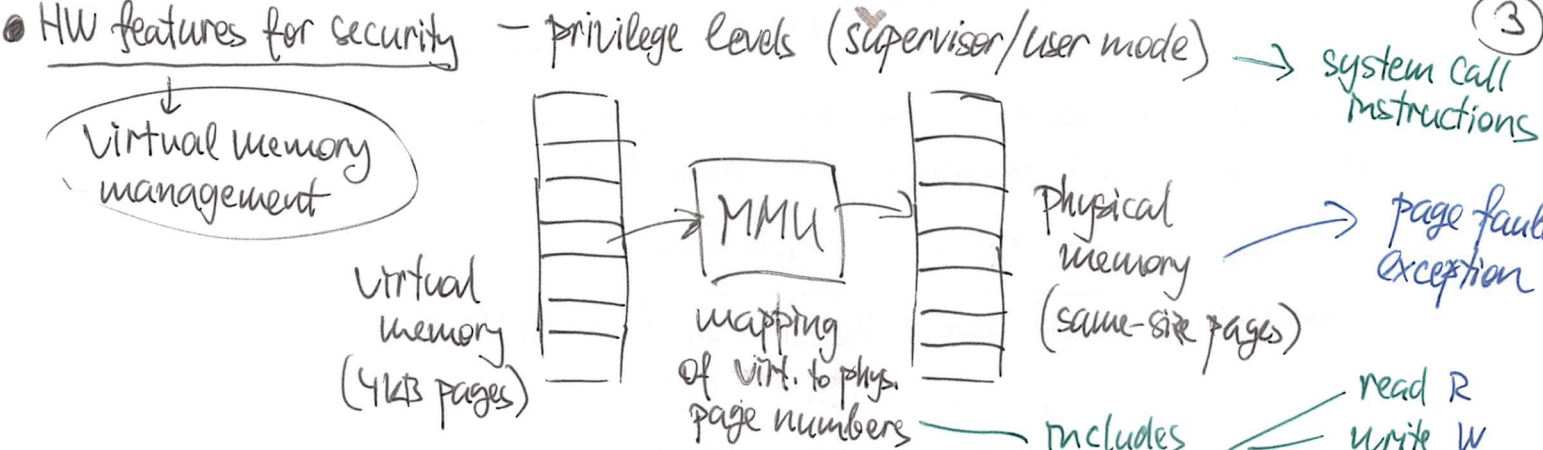
cooperative multi-tasking

- scheduling, priorities, real-time

③ security

need hardware support

- isolate hardware from programs
- separate programs/users



- Uses:
- protection of processes (private memory) ... RW(x) for 1 process
 - shared memory -- RW(x) in multiple processes
 - shared library ... R in multiple processes
 - lazy allocation ... read-only shared zeroes, copy on write
 - fork ... using copy-on-write mapping
 - SWapping ... store seldom-used pages to disk, read back on access

- caching - RAM is slow (CPU executes $\sim 10^9$ instructions/second, RAM latency is tens of ns)
- idea: small, very fast memory inside the CPU which remembers frequently used data] called a cache | can be better only for small memory (speed of light etc.)
 - caches 64B chunks of data (cache lines)
 - strategy:
 - write-through vs. write-back
 - when cache fills up: evict least-recently used item (LRU)
 - real caches have limited associativity → cache aliasing
 - multiple levels of caches
 - example: accessing a matrix row by row vs. column by column
 - ↳ sequential in memory
 - ↳ every access is a cache miss → very slow
- modelling of caches, cache-oblivious algorithms (not at this lecture)

- improving execution of instructions
- CPU works in cycles, historically 1 instruction took multiple cycles
 - pipelining

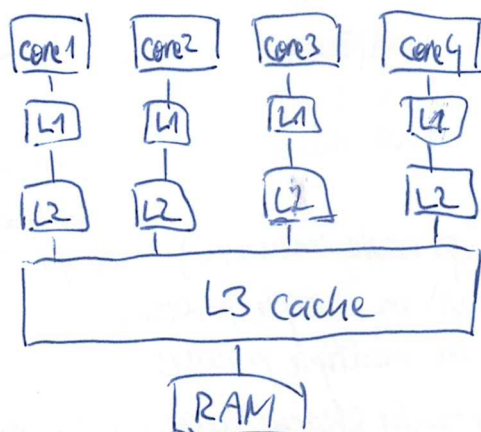
Fetch	Load	Comp	Store	ins.1
Fetch	Load	Comp	Store	ins.2
T1	T2	T3	T4	T5

all units of CPU always busy

problems: dependencies & (conditional) jumps
 - superscalar CPU: multiple units for different types of instructions, can run in parallel → scheduling instructions to units
 - jump prediction
- e.g.:
 - 1 fetch & decode
 - 2 load operands
 - 3 compute result
 - 4 store result
 all this is transparent to SW (well, almost: Meltdown & other bugs)

- multiple processors sharing memory (SMP = Symmetric Multi-Processing)
 - OS schedules processes on processors — real parallelism
 - hard to get right: locking in SW, cache coherency protocols in HW
- multi-core processors: SMP on a single chip

For example:



typical sizes

32 KB code + 32 KB data

256 KB unified

8 MB unified

16 GB

- multi-threaded cores: two cores sharing their execution units & caches
 - unclear benefits (can even make things worse!)
- virtual machines: simulating a whole machine within a process
 - including supervisor mode → the VM can run its own OS
 - including virtual peripherals
 - CPUs have special support for VMs (e.g., nested paging in MMU)

Relationship with theory

- will ignore most machine-dependent constants
 - concentrate on asymptotics ⇒ all machines (roughly) equal
 - use simple mathematical machines instead
 - I/O and caches need special treatment
- } rest of the semester