Lecture 1
Introduction

Computing platforms

Novosibirsk State University
University of Hertfordshire

D. Irtegov, A.Shafarenko

2018

What is a [computing] platform?

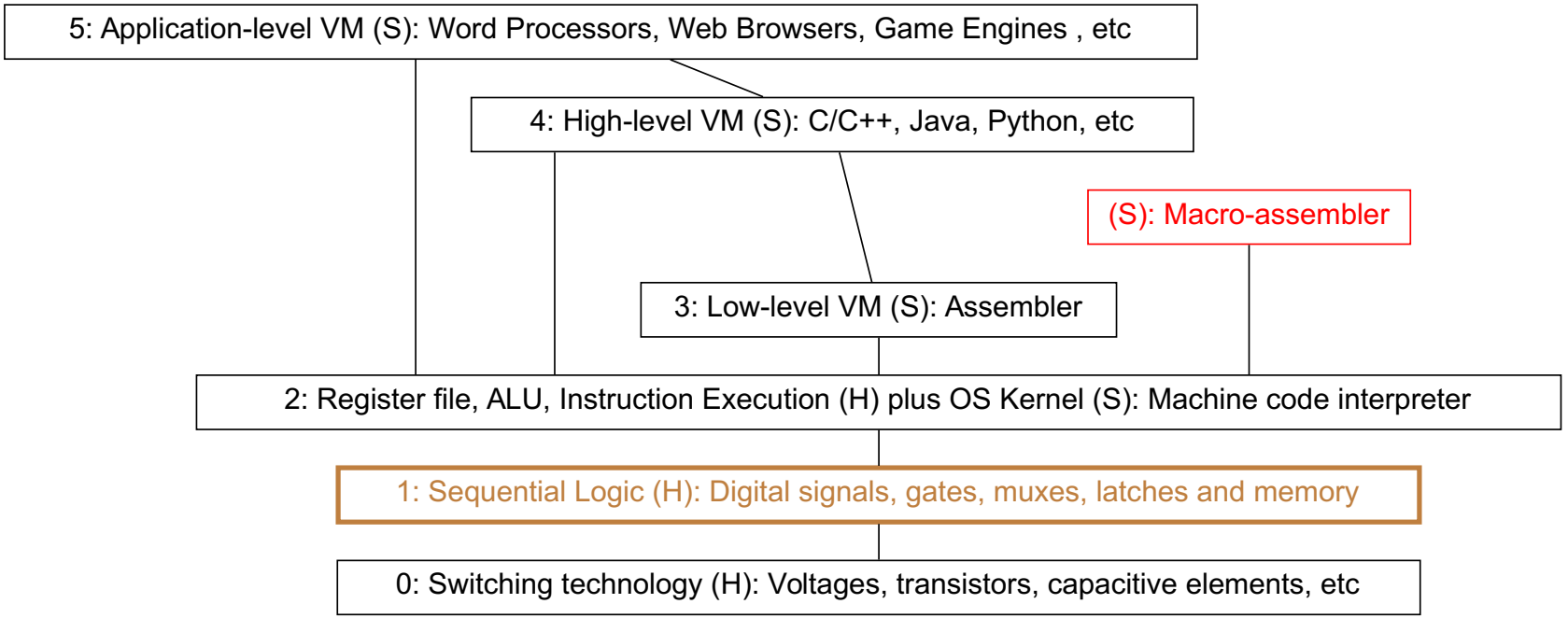
So, what we mean by Platform

- Platform is something you can use to build something else
- In this course, we somewhat extend this concept
  - We call a Platform something you can use
  - So we can group application and system components into a single category
- In computing, many platforms are actually *virtual machines* built on top of another platform
- But you cannot build a purely virtual machine
  - There must be something real at the bottom

Computer languages

- Digital computers are language machines
  - You write a program in some language
  - Computer runs (executes) it
- Every platform has a language (some have many)
- Build something on a platform
  ~= write a program using a platform language
- But platform is not A language,
- It is an environment (virtual machine) to run programs in a specific language
- And often a toolset to aid in creating programs (debuggers, profilers, IDE, etc)

Platforms hierarchy



It is even more complex

- Modern technology allows us to build anything from anything
- Application-level platform can have it's own programming language ("scripting language"), and people can use it to build something of even higher level
  - Example: Web browsers and JavaScript
- You can use high-level platforms to emulate lower-level ones
  - Example: Virtual machines in narrow sense (hypervisors, like VirtualBox or VMWare)
  - You can write a program in JS, emulating a hardware CPU capable of booting Linux or Unix (try to calculate a platform level of C program running on this system) and because of optimizing JiT it can have a decent performance
- We won't discuss it now, let's deal with simple case first

Why so many levels?

- Short answer:
  - Modern computer systems are very complex
  - Hierarchy is one of best ways to deal with complexity
- Another short answer
  - You can simplify programming
    (=> write more complex programs)
    by making computer to help you write programs
  - Then it got out of control
- Long answer:
  - Probably you will understand it
    as you learn several platforms of different levels

What is virtual machine

- In this course, virtual machine is a platform that is different from lower level platform
  - Other textbooks and documents can use different definitions
- Usually, higher level platform is easier to program
- So, most programming is done in Level 4 and higher
- Other reasons to create virtual machines:
  - To run programs from old or different platforms
  - To run several platforms on single physical machine
  - To aid in testing and debugging
  - Etc

Why study low level platforms?

- Virtual machines are not perfect
- VM tries to isolate you from details of lower-level platform, but it is impossible
- Specifics of low-level platforms (resource limitations, performance bottlenecks, logical constraints, etc) leak to higher levels
- If you know low-level platforms, you will understand higher-level ones (like C language or operating systems) much better.

Language translation

- Create VM ~= create a mechanism to translate platform language to the lower-level platform
- Two main techniques:
  - Compilation
    - You take source program and generate a new (supposedly equivalent) program in target language
  - Interpretation
    - You take source program and interpret is statement by statement
- Lowest level language must be interpreted, because (by definition) there is no lower-level native language
- There are hybrid techniques, like JiT compilation
  - But we won't discuss them now

Advantages of compilation

- Compiled program is not different from native language program.
  Actually, it IS a native program
- It means, there is no runtime overhead
  - compiled program can be suboptimal,
    this can be seen as form of the overhead
- Modern compilers are very good
  at writing optimal programs,
  on average, better than humans

Advantages of interpretation

- Interpreter is often simpler than a compiler
- You can do runtime checks (implement a “sandbox”) and run untrusted programs
- Handling runtime errors is much simpler
- Development and debugging cycle can be faster (no compilation phase)

Software and hardware

- Software and hardware are equivalent
  - With the exception that pure software cannot run (it must have some hardware below)
  - To run software, hardware must be programmable
- Any specific function can be implemented both in hardware and in software
  - In this course we will learn how to implement some interesting functions in hardware
  - I hope this will give you the idea how to implement even more complex functions, essentially everything you can imagine
- There are tools to describe hardware designs in high-level languages, similar to high-level programming languages, and automatically build circuits based on these descriptions (silicon compilation)
  - Most modern hardware is designed by tools like that

Reasons to choose

- Reasons to choose between hard- and software are mostly economical
- Hardware is usually faster,
  but this is not always important
- Software is cheaper to develop, copy and modify
- Complex designs practically always have bugs
  - You cannot use them if you cannot modify (fix) them
- Multilevel organization allows you to move between hard- and software implementations without breaking higher-level platforms

Von Neumann computer

- A programmable computing platform with random access program memory
- Typical example of Level 2 platform
- Usually an interpreter
  - there were attempts to implement so called binary compilation, but they failed and there are reasons for that
- One of most widely used types of programmable hardware
  - We will briefly discuss some other types of programmable hardware in next semester
  - Anything called a ”computer” has [at least one] von Neumann CPU inside
  - Many pieces of modern hardware you do not call a “computer” also have von Neumann CPU[s] inside

More on von Neumann computers

- Von Neumann computer interprets values in memory as commands (instructions)
- Values in memory are numbers
  - In binary computers, they can be viewed as bit strings (thus the term “binary code”)
  - But some von Neuman computers were not binary
  - For example, ENIAC was decimal

CPU commands

- Every valid command has a number (opcode)
- List of all valid commands (together with description of their semantics) is known as
  ISA (Instruction Set Architecture) or
  machine language
- ISA is not to be confused with *assembly language*, which is Level 3 platform in our classification
- Program in machine language is a sequence of numeric (binary) commands, known as *binary code*
- Assembly language program is a human-readable representation of machine language program

Why random access is important

- When program is loaded in random access memory, every command has an address
(a number of memory cell where this command is located)
- Von Neumann CPU has so called jump (branch) instructions that transfer execution to any specific address
- So, the execution of program is not [strictly] sequential
- Jump instructions are used to implement constructs like loops, conditional statements, subroutine calls, etc
- Most of this semester is dedicated to studying how this actually works

CPU families

- People want to run programs for old CPUs on newer machines
- Series of CPU with the same or compatible ISA
  - Compatible means that never versions of ISA have all the same commands as older one, may be with addition of some new commands.
- Oldest widely used ISA is IBM System/360, first developed in 1965.
  - Still fully supported by IBM System/z computers
- Probably second oldest (between widely used now) is MCS-48 family of 8-bit microcontrollers, developed by Intel in 1976 and licensed by other manufacturers.
  - Every PC-compatible motherboard has MCS-48 circuit, typically implemented as part of so-called ”south bridge”

Popular ISA: ARM and x86

- x86 - CPU family compatible with Intel 80386, released in 1987.
  - Older Intel CPU like 8086/88 and 80286 are NOT members of x86 family
  - Originally 32-bit, extended to 64-bit (x86_64 or simply x64) by AMD in 2003
- ARM (originally Acorn RISC Machine) was developed by Acorn Computers in 1985 for use in Acorn Archimedes microcomputer.
  - In 1990s it was licensed by other manufacturers mostly for embedded applications
  - In 2000s it started to gain popularity in smartphones, and circa 2010 it surpassed x86 by worldwide usage
  - Originally 32-bit, extended to 64-bit in 2011.

Why not study commercial ISA?

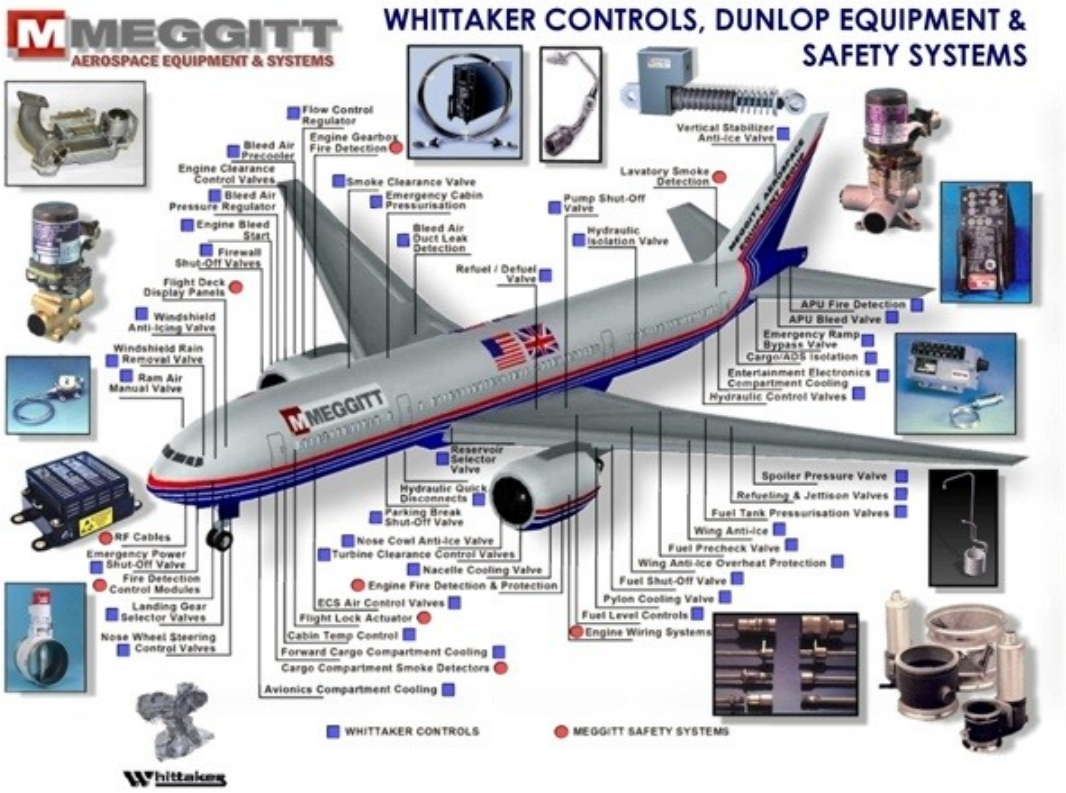
- Well, nobody forbids you to study ARM or x86 or MCS-48/51 or Atmel AVR or etc
  - On interview, I've heard some of you actually studied Arduino
- But commercial ISA have long (and sometimes tragic) history and are constrained by backward compatibility
- Many important details and even concepts of x86 or ARM cannot be understood without knowing peculiarities of 1980s hardware and intellectual fashion of that period
- x86 is also constrained by attempts to maintain assembly-level and OS-level compatibility with 80286, 8086 and even earlier 8080/85 ISA
- This will distract us from basic concepts we need to understand

Welcome CdM-8

- CdM-8 is a 8-bit Level 2 platform (von Neumann CPU) designed by A. Shafarenko
- Two implementations:
  - Software emulator cocoemu, written in Python
    - Works together with CocoIDE
  - Gate-level (Level 1) circuit file for Logisim
  - No silicon and copper hardware implementation (somebody want to volunteer?)

Why CdM-8 (an analogy)

Modern aircrafts are pretty complex



So let’s start from something simpler

Why it is good idea

- It can fly
- It is fun to fly
- It can be disassembled to sticks and gadgets and assembled back (and it is also fun)
- It can illustrate basic concepts like lift and drag
- It can illustrate some not so basic concepts like stability, weight discipline and stall recovery
- As you understand basics, we can move to something more complex

Advantages of CdM-8

- Very simple design, good for illustrating both Level 2/3 and Level 0/1 concepts
- No backward compatibility burden and related complexity
- Fully functional (for a CPU with 256 bytes of total memory)
- Advanced macroassembler with some capabilities of Level 4 platforms (thus the name Level 3 ½)
- Nice IDE with ability to see all program and data memory in a single window (impossible even on 16-bit CPU!)
- Extended version with interrupts, memory banks and system/user mode
- Ability to extend the design with Logisim circuits

Two types of von Neumann CPU

- Manchester architecture
  (for “Manchester Baby”)
  - Stores program and data in same memory
  - In some textbooks, only Manchester computers are called von Neumann computers
- Harvard architecture
  (for Harvard Mark I, which was not even stored-memory computer)
  - Uses separate banks of memory for program and data
  - If you remember, ENIAC was actually a Harvard computer
- Some CPUs can be switched from Harvard to Manchester by single fuse or just by reconnecting memory banks
- Some textbooks and documents use other definitions
  - For example, some CPUs use separate caches for code and data, and this is called Harvard architecture even if these caches are backed by same main memory

Course materials

- http://fit.nsu.ru/~fat/Platforms
- Short documentation on CdM-8 ISA and long book, containing full description of CdM-8 platform and toolchain and other useful and interesting info.
- To run CdM-8 tools (cocoide, cocas, cocoemu), you need Python 2 (2.6 or 2.7 recommended). Cocoide depends on tkinter library (installed by default on Windows and MacOS)
- For your convenience, Logisim archive is included. You also can download it from official site http://www.cburch.com/logisim/ , it is free.

How we will work

- Your main teaching guide is Cocomaro the Robot.
- It is a mail robot which will send you emails
- Every email will contain a task description and instructions how to submit a solution
- After your send a correct solution for a task, Cocomaro will send you next one
- Cocomaro checks for new mail every three minutes 24/7 (if it does not, you should alert me)
- If you have questions, you are free to ask me and other teachers, personally during practice lessons or by email

Course grading

- Tasks sent by Cocomaro will not be graded, so cheating is pointless
- We will have assessment tests on most of practical lessons
- Assessments will include questions and solving tasks similar to ones that were sent by Cocomaro
- You will solve tasks in controlled environment, with cheating prohibited

Course grading

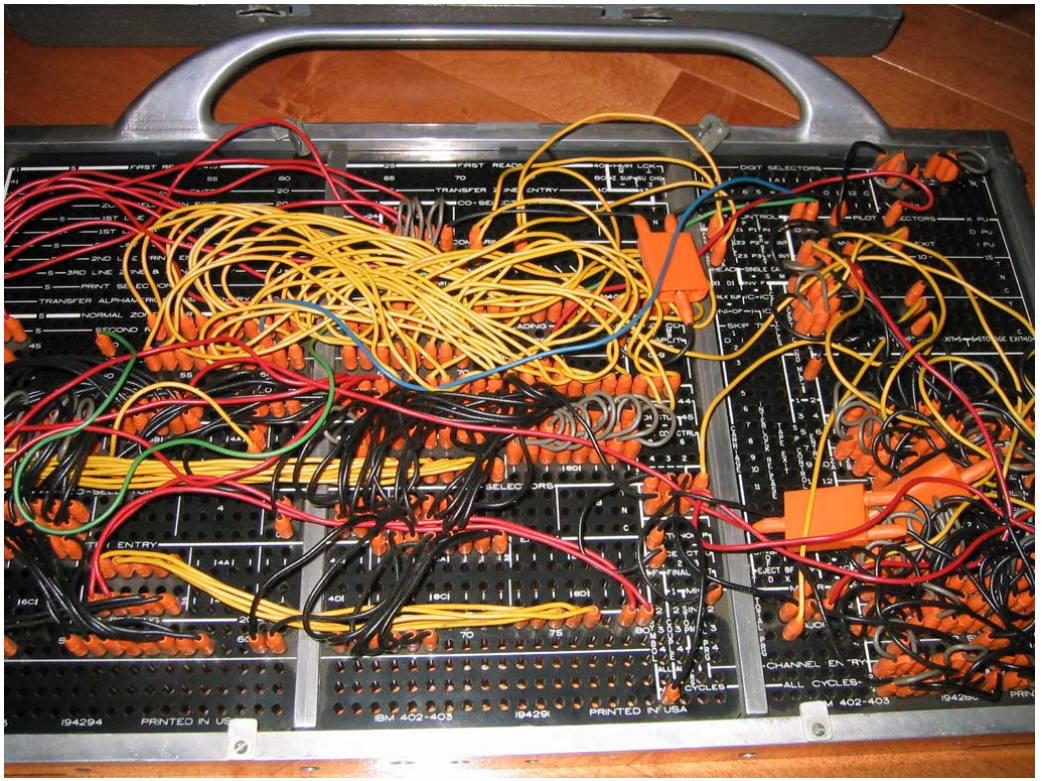
- First semester is graded fail/pass
- To pass, you must successfully complete ~25% of all tests (I hope this will be simple)
- Second semester is graded 2/3/4/5
- You do a group project
- Groups are formed semi-randomly, so every team will have students of different level (based on assessment tests)
- Group project results are main component of the final grade
- If you are not happy with project results, you can try to get a higher grade by passing the exam

Some interesting facts

- One of the first working von Neumann computers was created programmatically
- Originally it was the first fully-electronic computer called ENIAC
- Design started in 1943, delivered in 1945
- Used for military research, including numerical modelling of thermonuclear explosives
- It was programmed by so called plugboards

Plugboards

- Also known as patch panels
- Used in 1930s to “program” IBM tabulators (machines for processing punch cards)



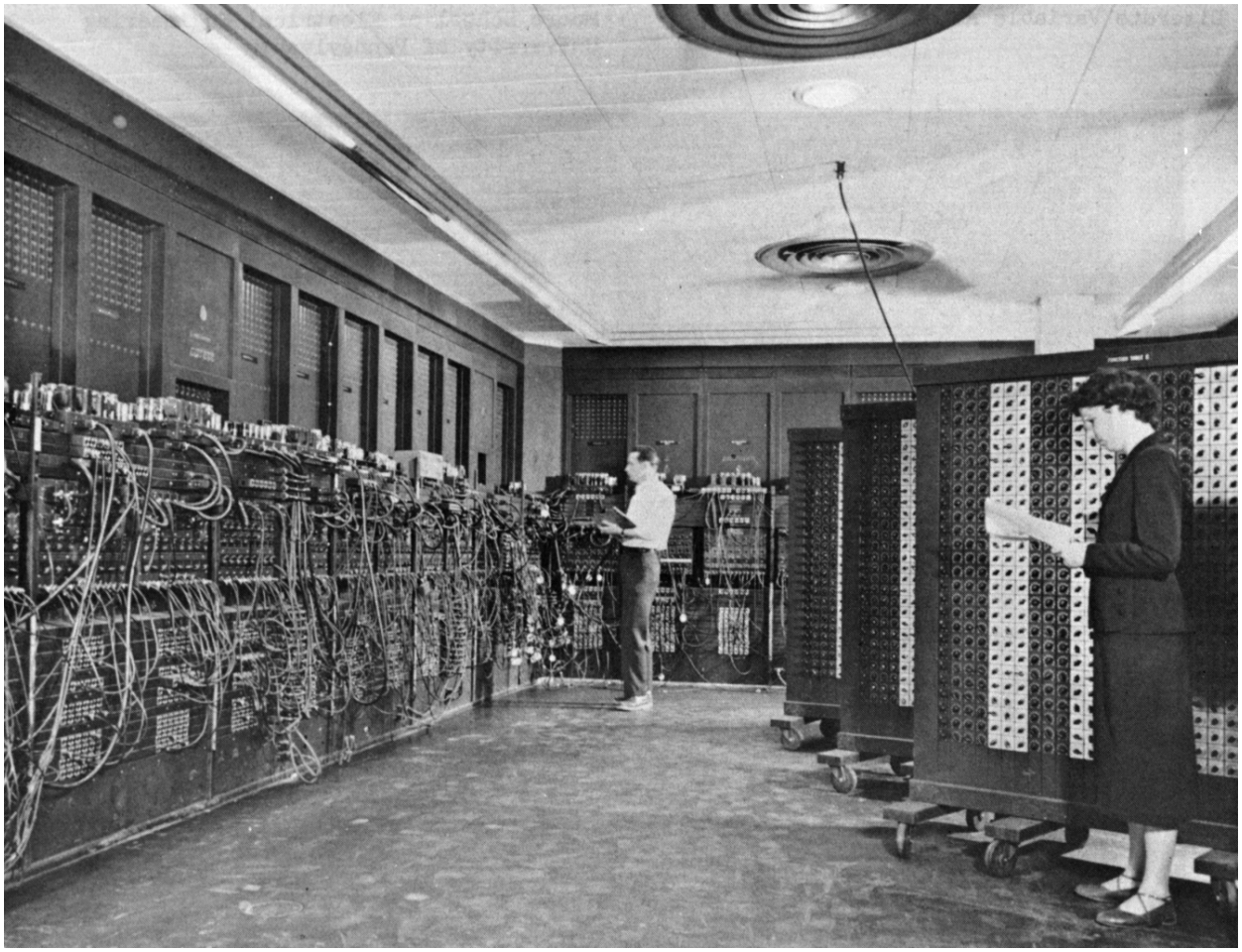
Plugboards and tabulators

An IBM 407 Accounting Machine with control panel (plugboard) inserted, but not engaged.

ENIAC

On the left:
Arithmetical units with plugboards

On the right:
Function tables (sort of read-only random-access memory, every cell is a mechanical 10-position switch)



ENIAC as von Neumann CPU

- In 1948, ENIAC was converted to a stored-program computer by creating a ”program” (plugboard wiring) to use function tables as program memory
- Von Neumann was not inventor of this idea, but he worked as a consultant in EDVAC project and participated in ENIAC conversion
- EDVAC was operational in 1949 and was not a true von Neumann machine
- EDVAC used serial (not random access) program memory based on mercury delay lines

First true von Neumann computer

- “Manchester baby” or SSEM
- Created in Victoria University of Manchester, England
- Von Neumann had nothing to do with that project
- Run first program 21 June 1948
  three months before converted ENIAC
- Had random-access memory based on so-called Williams tube (read Wikipedia...)
- Used same memory for program and data
- Was a testbed for Manchester Mark I and later Ferranti Mark I (first serially produced and commercially available stored-memory computer)

What happened next

- After stored program computer proved to be a good idea, people started to write program tools to simplify programming: assemblers and linkers
- Then other people invented higher level languages, like autocode, Fortran and Lisp
- And then it got out of control and now we have multilevel platforms writing programs for themselves