



# Sequence 1.1 – Introduction to Compilers

P. de Oliveira Castro  S. Tardieu

# How to tell machines what to do?

**Figure 1:** Traditional Loom

# How to tell machines what to do?

**Figure 2:** Jacquard Loom (1804)

# How to tell machines what to do?

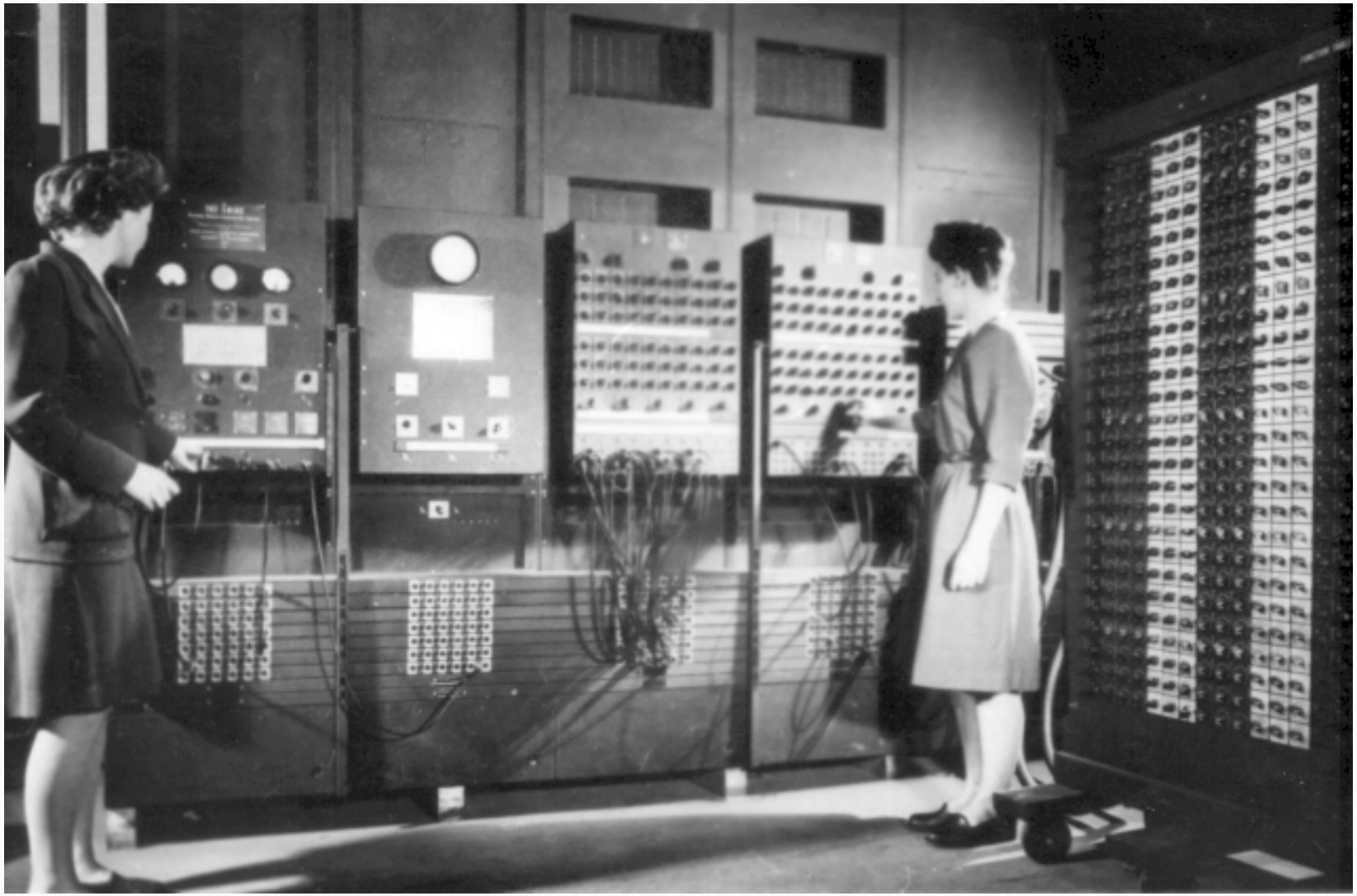

**Figure 3:** ENIAC (1943)

# How to tell machines what to do?

**Figure 4:** Macintosh Plus (1986)

# How to tell machines what to do?

- Switches or connection boards
- Perforated cards
- Stored programs in memory (tapes, disks, electronic memory)

Each constructor has its own *instruction set*.

# What is an instruction set?

- Each instruction available in the machine such as addition or memory moves is encoded as a binary number:
  - In the ARM 32 instruction set, 1110000110100000 0000 000000000001, moves the content of register `R1` into `R0`.
  - In the Intel x86 instruction set, 10001001 11 000 011, moves the content of register `eax` into `ebx`.
- Remembering binary codes by heart is difficult; we prefer *mnemonics*, forming an *assembly* language:
  - `mov $r0, $r1`
  - `mov %eax, %ebx`

# No single instruction set!

- Mnemonics (often called *instructions*) are much easier to remember and write.
- But each processor has unique features that require different mnemonics:
  - A single x86 instruction can combine arithmetic and memory operations.
  - ARM 32 arithmetic instructions operate only on registers.
- How can we write a portable program that works across all architectures?
  - Mnemonics are *low level*, easily understood by the machine, but specific to a given processor.
  - We would prefer a *high level* language easily understood by humans and targeting many processors.

# Interpreters

- Define a common *high level* language.
- For each architecture we write in assembly an *interpreter*:
  - It reads the high-level program instruction by instruction.
  - For each instruction it calls an assembly procedure.
  - Control-flow instructions (tests, loops) jump to another place.

**Advantages**

- Only write a single program (the interpreter) in assembly.
- High-level programs are portable across all architectures.

**Disadvantages**

- Slow: instead of being directly executed each instruction must be read, decoded and simulated.
- Hard to optimize: instructions are read on the fly, the interpreter cannot easily simplify or combine patterns.

# Interpreters and Virtual Machines

- Interpreters are still used today for many languages (Python, Ruby, PHP).
- Another option is to write a *Virtual Machine*, an efficient interpreter for a common assembly language (Java HotSpot VM, DotNET).
- Modern virtual machines and interpreters use some tricks to optimize the code and speed-up the execution, but they still add an additional software indirection layer.

# Compilers

- While an interpreter translates the common language on the fly, a *compiler* translates the whole program to assembly before the execution.
- Compilers read a high level language and produces an *equivalent* assembly program.
- The notion of equivalence is tricky. A simple definition is that two programs are equivalent when they produce the same observable effects when fed with the same inputs.

# Compilers

**Advantages**

- Cost of translation is payed only once.
- No need to ship either the source code or the interpreter.
- Many optimizations can be applied while compiling.

**Disadvantages**

- Less flexible and dynamic than an interpreter for distributing code.
- Some optimizations can only be applied at run time.

# Image credits

- *Traditional Loom*, photography by dagrimshaw, CC0 1.0 Public Domain
- *ENIAC*, U.S. Army Photo, Public Domain
- *Jacquard Loom*, photography by D. Monniaux, CC SA 3.0
- *Macintosh Plus*, photography by B. Patterson, CC BY 2.0