Modern real-world computers

Anton Luka Šijanec, Filip Štamcar

Faculty of Mathematics and Physics

11th November 2026

History: On computable numbers (Turing, 1936)

The behaviour of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite.

History: Computer as an occupation

- 17th century: A person who makes calculations or computations; a calculator, a reckoner; spec. a person employed to make calculations in an observatory, in surveying, etc. (Oxford Dictionary)



History: Electomechanical

We may now construct a machine to do the work of this computer. To each state of mind of the computer corresponds an "m-configuration" of the machine.

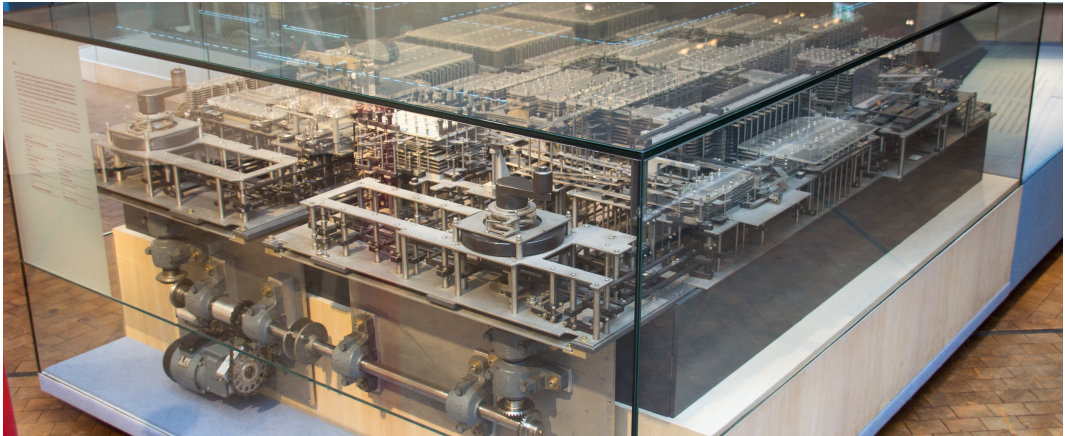
---

† If we regard a symbol as literally printed on a square we may suppose that the square is $0 \leqslant x \leqslant 1$, $0 \leqslant y \leqslant 1$. The symbol is defined as a set of points in this square, viz. the set occupied by printer's ink.

- 1937: Turing built a binary multiplier using relay switches, as a proof-of-concept tool for cryptanalysis of German ciphers.
- 1937: Claude Shannon's master thesis describes the similarity between Boolean logic and relay circuits.

History: Relay computer

- 1938: German Z1 by Konrad Zuse is completed: Reads instructions (Rechenpläne) from punched 35 mm celluloid film, ALU: 22-bit FP, 16-word r/w memory, distinct control unit, 1 Hz clock driven by a vacuum cleaner motor, 1000 kg
- No conditional branching. Unreliable.



Modern computer architecture (von Neuman/Harvard)

- Processor: Control and arithmetic/logic unit.
- Memory: Read from and written to by the processor. With a fixed-size address ($\in \mathbb{Z}_n$) addresses modifiable words of fixed-size ($\in \mathbb{Z}_m$).
  - Registers: Variables with special meaning to the processor, for example the instruction pointer.
- I/O: Mechanism to communicate with external devices, machines or humans.
- At each processor cycle (dictated by a clock oscillator), the processor fetches an instruction, acts upon it, and writes to memory accordingly. Additionally, it interacts with other devices (I/O) if instructed to do so.
- An instruction set includes arithmetic instructions, conditional jump instructions, ...

Random access machines

- We can model a theoretical computer by means of a random access machine. Formally, RAM is a multi-register counter machine supporting indirect addressing.
  - Registers: Infinitely many registers, indexed by natural numbers, each holding a numeric ($\in \mathbb{N}$) value.
  - Instruction table: Contains execution instructions from a given instruction set (common instructions: increment, decrement, clear to zero, copy, conditional jump, halt).
  - Instruction register: Specifies the instruction from the table currently being executed.
- Addressing modes: Direct and indirect.
- RAMs are Turing complete, since multi register counter machines are Turing complete.

Real-world computer architecture

- Processor: 64-bit word size, 16 registers, 64-bit address space for addressing memory, thousands of instructions (machine language), running at $\approx 10^9$ cycles per second. (x86-64)
- Memory: $\approx 10^{11}$ bits of memory (DRAM)

Real-world computer architecture

- Processor: 64-bit word size, 16 registers, 64-bit address space for addressing memory, thousands of instructions (machine language), running at $\approx 10^9$ cycles per second. (x86-64)
- Memory: $\approx 10^{11}$ bits of memory (DRAM)
- **Not** Turing complete, because they have finite state. They are merely finite state machines:
  - A state of a computer consists of memory content, register values, program code, etc.
  - The transition function is given by the specification of the processor.

Real-world computer architecture

- Processor: 64-bit word size, 16 registers, 64-bit address space for addressing memory, thousands of instructions (machine language), running at $\approx 10^9$ cycles per second. (x86-64)
- Memory: $\approx 10^{11}$ bits of memory (DRAM)
- **Not** Turing complete, because they have finite state. They are merely finite state machines:
  - A state of a computer consists of memory content, register values, program code, etc.
  - The transition function is given by the specification of the processor.
- Peripherals: Hard disks, USB sticks, network storage, etc.
  - Hard disks only finitely extend the state of a computer.
  - Attaching two computers with finite state to each other still produces a system with a finite state.

Extending a computer to Turing completeness

- Turing machine is a finite-state machine with a special environment – the infinite tape.
- We could define such a hypothetical infinite state extension for our real world computer.
- Examples:
  - Infinite seekable tape over USB.
  - Buying and attaching more memory when we run out.
- At any given time, a TM will have touched at most a finite amount of cells on it's tape.

Programming real world computers: Python

- Is Python Turing complete? Can one simulate a universal Turing machine in Python?
- Yes. We can implement a UTM in Python.
- Python (as a language) doesn't limit the number of available objects, we never run out of memory (theoretically).
- We showed that real-world computers are FSMs, so all real-world "implementations" of Python actually implement a subset of Python that isn't Turing complete (limited memory).
- We still accept them as valid implementations of Python.

```
def tm (inputword, transition, haltstates):
        if len(inputword) == 0:
                inputword = (None, tuple())
        tapeleft = tuple()
        taperight = inputword[1]
        tapehead = inputword[0]
        state = 0
        while state not in haltstates:
                if len(tapeleft) == 0:
                        tapeleft = (None, tuple())
                if len(taperight) == 0:
                        taperight = (None, tuple())
                tapehead, state, direction = transition[state][tapehead]
                assert direction in [-1, 0, 1]
                if direction == 0:
                        continue
                if direction == -1:
                        tapeleft, tapehead, taperight = tapeleft[1], tapeleft
                            [0], (tapehead, taperight)
                if direction == 1:
                        tapeleft, tapehead, taperight = (tapehead, tapeleft),
                            taperight[0], taperight[1]
        return tapeleft, tapehead, taperight, state
```

Programming real world computers: C

- There are arguments that C isn't Turing complete, because:
  - All types need to be of statically known finite size.
  - You can take a pointer to any object, pointers as types have a finite size, and compare equal if and only if they point to the same object.
  - Having a separate heap for every type (allowed) does not help, as there can only be finitely many types.
- As C allows recursion without limiting depth, we could implement a pushdown automata using function return values.
- C99 adds `va_copy` which allows iteration over variable function arguments multiple times. This might allow building a two-stack pushdown automata, which is equivalent to a Turing machine.

Sources

- Alan Turing, 1936: On computable numbers
- Leo Corry, 2017: Turing's Pre-War Analog Computers
- Marvin Minsky, 1967: Computation: finite and infinite machines
- ISO, 2010: ISO 9899: Programming languages – C
- Raul Rojas, 2014: The Z1
- https://memo.barrucadu.co.uk/c-is-not-turing-complete.html
- Python software foundation, 2026: Python language reference
- feersum, 2018: Is C actually turing-complete?

5.2 TURING'S ANALYSIS OF COMPUTATION PROCESSES

In his 1936 paper, A. M. Turing defined the class of abstract machines that now bear his name. A *Turing machine* is a finite-state machine associated with a special kind of environment—its *tape*—in which it can store (and later recover) sequences of symbols. We will describe these machines in greater detail in chapter 6 and the sequel. They are very simple. At each moment the (finite-state part of the) machine gets its *input* stimulus by *reading* the symbol written at a certain point along the tape. The *response* of the machine may *change that symbol* and also move the machine a small distance either way along the tape. The result is that the stimulus for the next cycle of operation will come from a different "square" of the tape, and the machine may thus read a symbol that was written there long ago. This means that the machine has access to a kind of rudimentary exterior memory in addition to that provided within its

Appendix

4 Values stored in non-bit-field objects of any other object type consist of $n \times$ **CHAR_BIT** bits, where n is the size of an object of that type, in bytes. The value may be copied into an object of type **unsigned char [**n**]** (e.g., by **memcpy**); the resulting set of bytes is called the *object representation* of the value. Values stored in bit-fields consist of m bits, where m is the size specified for the bit-field. The object representation is the set of m bits the bit-field comprises in the addressable storage unit holding it. Two values (other than NaNs) with the same object representation compare equal, but values that compare equal may have different object representations.

6 Two pointers compare equal if and only if both are null pointers, both are pointers to the same object (including a pointer to an object and a subobject at its beginning) or function, both are pointers to one past the last element of the same array object, or one is a pointer to one past the end of one array object and the other is a pointer to the start of a different array object that happens to immediately follow the first array object in the address space.[109)]

Appendix

11 Recursive function calls shall be permitted, both directly and indirectly through any chain of other functions.

2 The types declared are **`size_t`** (described in 7.19);

`FILE`

which is an object type capable of recording all the information needed to control a stream, including its file position indicator, a pointer to its associated buffer (if any), an *error indicator* that records whether a read/write error has occurred, and an *end-of-file indicator* that records whether the end of the file has been reached; and

`fpos_t`

which is a complete object type other than an array type capable of recording all the information needed to specify uniquely every position within a file.