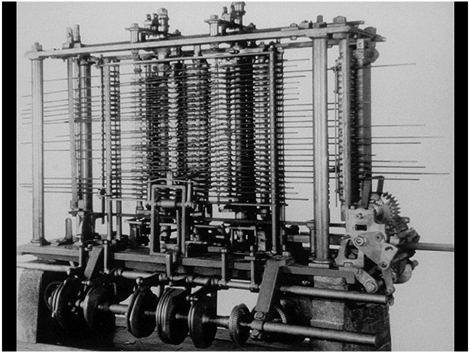
COMP1730/COMP6730
Programming for Scientists

The Computer

The Computer

- A computer combines repeated function execution and memory.



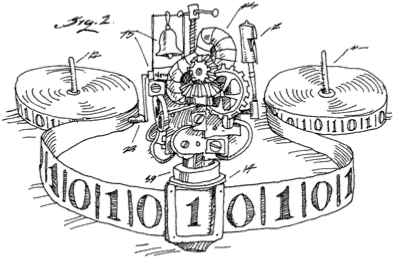
- In a programmable computer, the function is selected by program instructions.

Turing’s Machine

The Turing Machine (TM)

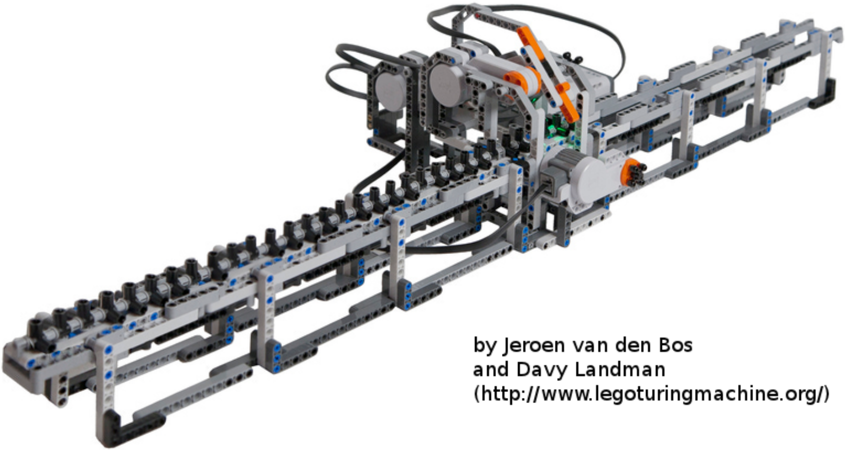
- A simple, abstract model of a (universal) computer.
- Capable of executing any program that can be implemented on any known digital computer hardware.
- Computable by a Turing machine is widely accepted to be the same as computable, full stop (Church–Turing thesis).

- A *finite* state.
- An *infinite* tape of discrete cells marked with letters from a fixed, finite alphabet.
- A current position on the tape.



- Next state, letter written (at current position) and movement (left, right) is a *finite* function of the current state and the letter read.

http://aturingmachine.com/



by Jeroen van den Bos
and Davy Landman
(http://www.legoturingmachine.org/)

http://legoturingmachine.org/

Execution cycle

1. Read the symbol (X) on the tape at current position (P).
2. Compute the *transition function*:
   - the next state (Q');
   - the new symbol (Y); and
   - the movement ($\Delta \in \{-1, 0, +1\}$),

   as a function of the current state (Q) and X.
3. Write Y at position P on the tape (replacing X).
4. Update the position to $P' = P + \Delta$.
5. Update the current state to Q'.

* ...and repeat.

- A TM is completely defined by its transition function.
- Possible inputs to the transition function are *finite* – can be written as a lookup table.

$T(Q, X) =$

	$\Box$	0	1
A	$(B, \Box, -1)$	$(A, 0, +1)$	$(A, 1, +1)$
B	$(Z, \Box, 0)$	$(B, 1, -1)$	$(A, 0, +1)$
Z			

The universal Turing Machine

- Consequently, the transition function of a TM can be *encoded* as a string of letters (even in binary).
- We can design a TM, *U*, that reads the encoding of any other TM, *M*, and “simulates” the execution of *M*.
- *U* is a *programmable* computer.

Digital Circuits

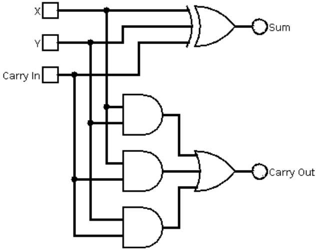
Reminder: Binary numbers

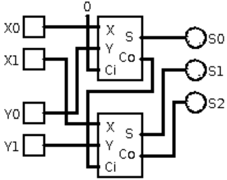
- A binary number is simply a number in base 2.
  - Also negative and fractional numbers.
- Electronic computers work with binary representation of data.
  - A single binary digit (*bit*) is represented by the presence or absence of current in a circuit.
  - 8 bits make a *byte*.
  - Fixed-width numbers (e.g., 32-bit, 64-bit) are often called (short or long) *words*.

$$\begin{array}{r} {\scriptstyle 1\,1\,1} \\ 0101_2 \\ +\,0111_2 \\ \hline 1100_2 \end{array}$$

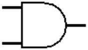
Combinatorial circuits

- A circuit computes binary outputs as a function of binary inputs.
  - Primitives ("*gates*") implement elementary functions.
  - Wires carry values.
- A circuit can be an element of a larger circuit (*abstraction*).





Elementary binary functions

AND ()

x	y	$x \cdot y$
0	0	0
0	1	0
1	0	0
1	1	1

OR ()

x	y	$x + y$
0	0	0
0	1	1
1	0	1
1	1	1

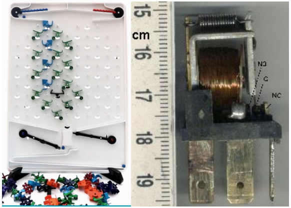
NOT ()

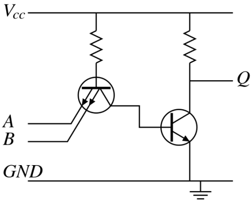
x	$\overline{x}$
0	1
1	0

- Any binary function can be written as a combination of AND, OR and NOT.
- Other primitives (XOR, NAND, NOR) are also used.

Realisations of circuits

- Mechanical (1800's idea, only used in toys)
- Electro-mechanical relays and valves (1930's)
- Transistors (1950's), integrated circuits (1960's), CMOS (1990's).
- Performance: size (density), speed, power/heat, cost.

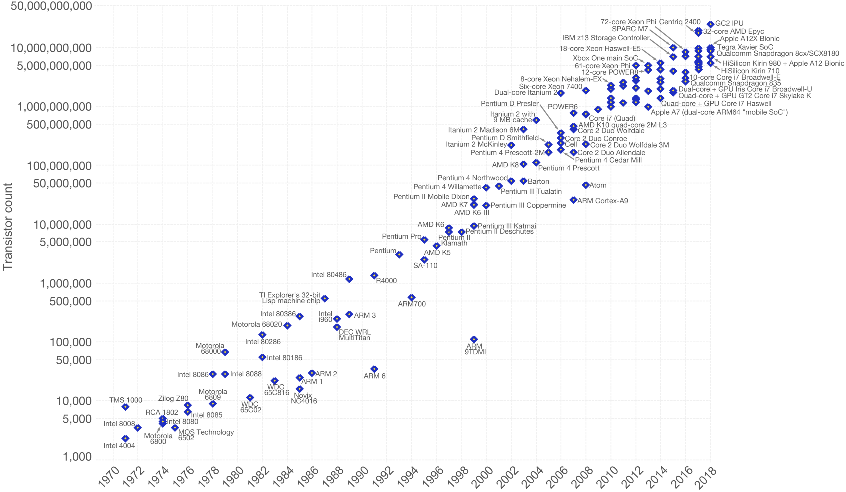




Moore’s Law – The number of transistors on integrated circuit chips (1971-2018)

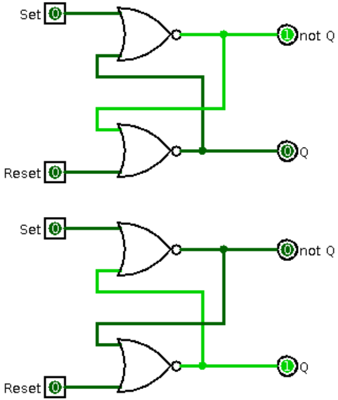
Our World
in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are linked to Moore's law.

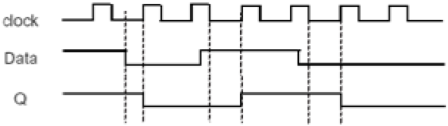


Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

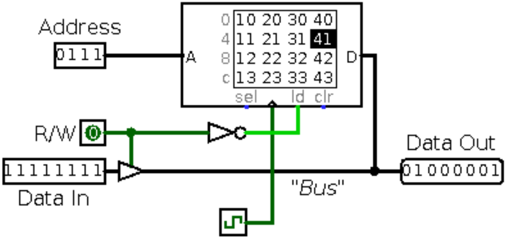
Memory circuits

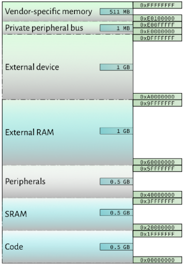


- A closed (feedback) circuit can *remember* a value.
- Problems with feedback (e.g., oscillation, races).
- A *clock* is a regular synchronising signal.



Addressable memory (RAM)



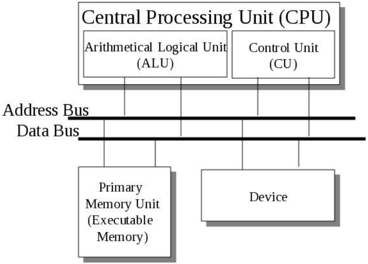


- Can store many *m*-bit words but only access (read/write) one word at a time.
  - The *address* determines which word.
- Like a very large array of fixed-width integers.

von Neumann’s Machine

von Neumann architecture

- Architecture of the modern digital computer.
- CPU implements a fixed (small) set of operations.



- Program and data are stored in memory.
- The CPU repeatedly reads and executes instructions from memory.

Execution cycle

1. CPU reads next instruction from memory address given by the program counter (*PC*).
2. The instruction details:
   - what operation to do ($+$, $-$, $\times$, $\leq$, copy, ...)
   - what operands to do it on (CPU register, memory, constant)
   - where to store the result.

- If instruction is a (conditional) *jump*, set *PC* to target address, else to $PC +$ instruction size.
- ...and repeat.

Example (x86 instruction set)

Instruction			
Byte 1	Byte 2–4		
00	Op. 1	Op. 2	
01	R/M 8	R 8	Add
02	R/M 16/32	R 16/32	Add
03	R 8	R/M 8	Add
	⋮		
28	R/M 8	R 8	Subtract
	⋮		
74	offset (8)	–	Jump if last $=$ was true
	⋮		
E9	offset (16)		Jump
	⋮		

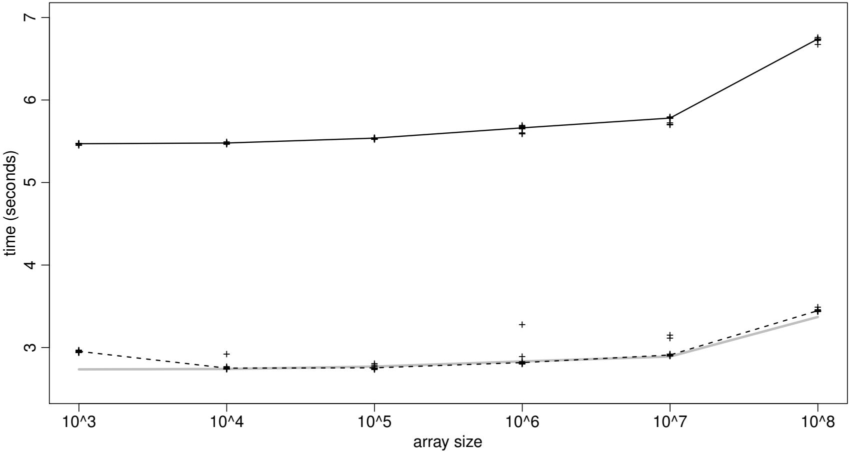
R/M 16	
00	memory at BX+SI
01	memory at BX+DI
	⋮
08	memory at BX+DI+offset (8)
	⋮
C0	AX register
C8	CX register
	⋮

Registers, memory and cache

- The CPU has a “working” memory: *registers*.
  - Registers are limited (word-sized) and few (typically a few 10’s) but fast (run at the CPU’s clock speed).
- Main memory is high-density (maybe 10^8 times larger) but slower than the CPU (often factor 10 or more).
- A *cache* is a smaller, high-speed memory between CPU and main memory.
  - Caching is transparent to the program.

The bus

- The bus transfers data between components: CPU(s), FPU, memory, peripherals (hard drive, graphics, network).
  - Components can run at different clock speeds.
  - Bottle neck: only one component can write to the bus at any time.
- Speed-ups:
  - Direct memory–peripherals paths.
  - Out-of-order and speculative execution: CPU executes independent or (possible) future instructions while waiting.


time (seconds)
7
6
5
4
3
10^3
10^4
10^5
10^6
10^7
10^8
array size

Assembler

- Programming languages that do not have (much of) an abstraction from the way that the CPU works are usually called “assembler”.
- Assembly languages typically provide mnemonic names for instructions and operands, address labels, and other conveniences.

```
  xor  ax, ax
  pop  bx
  mov  cx, [bx]
  add  bx, 4
@loop:
  add  ax, [bx]
  add  bx, 4
  sub  cx, 1
  jnz  @loop
  push ax
  ret
```

When and why does this matter?

- In certain cases, for performance.
  - “hand-written assembler is fast” is mostly a myth.
  - But some effects (cache, bus, etc) matter in some situations.
- Interfacing with hardware (e.g., GPU programming, embedded computing devices).
- Understanding (writing?) malicious code, and how to guard against it.