AKADEMIA GÓRNICZO-HUTNICZA
IM. STANISŁAWA STASZICA W KRAKOWIE

Introduction to Computer Science
Lecture 01

Version: 2024

Marek Wilkus Ph.D. **http://home.agh.edu.pl/~mwilkus**
Faculty of Metallurgy and Industrial Computer Science
AGH UST Kraków

Conditions

- Positive grade of exercise courses.

- While lectures are not compulsory, exercise courses ARE.
- Two absences without documents are allowed.
- Positive grades of 3 tests during exercise courses.
  - Final exercise grade – average of 3 grades from tests.

- ALL ABSENCE-RELATED DOCUMENTS MUST COMPLY WITH AGH UST RULES AND MUST BE SUPPLIED IN 2 COURSES AFTER THE LAST COURSE OF ABSENCE AT MAXIMUM. MISSING TESTS MUST BE CORRECTED AT THE FOLLOWING COURSES.

Topics

- Historical introduction,
- Modern computers and their architectures,
- Basic principles of operation of modern CPUs.
  - Logic, number systems, commands.
- Algorithms:
  - General rules for description of algorithms,
  - Basic data structures,
  - Basic algorithms and their implementations.
- Low-level programming – assembly language:
  - Introduction to NASM,
  - Simple programming.
- Operating systems design and operation,
  - Memory management in operating systems,
  - Data management in real-life applications.
  - Additional aspects of modern OSes.

Exercise courses - laboratories

- Introduction
- Introduction to processors,
- Programming,
- C++ simple programming
- [test 1]
- NASM,
- Implementing simple data structures in C++,
- Implementing simple algorithms (C++),
- [test 2]
- Automation of tasks – high level languages,
- Operating system administration – high-level languages,
- [test 3]

Bibliography

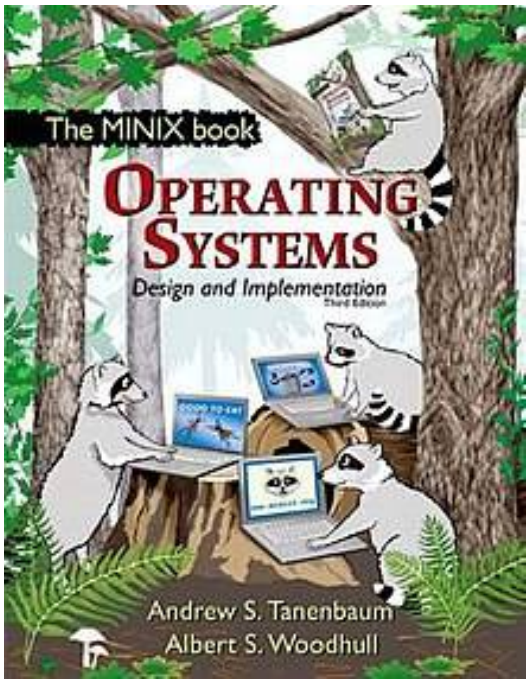
- Harel David - Algorithmics: The Spirit of Computing

- Patterson, David A. and John L. Hennessy. Computer Organization and Design: The Hardware/Software Interface, Fourth edition

- **Additional books:**

- Tannenbaum, Woodhull – Operating Systems Design and Implementation (The Minix book)

- Baber R. L. - The Spine of Software: Designing Provably Correct Software - Theory and Practice – PDF in http://www.cas.mcmaster.ca/~baber/Books/Books.html

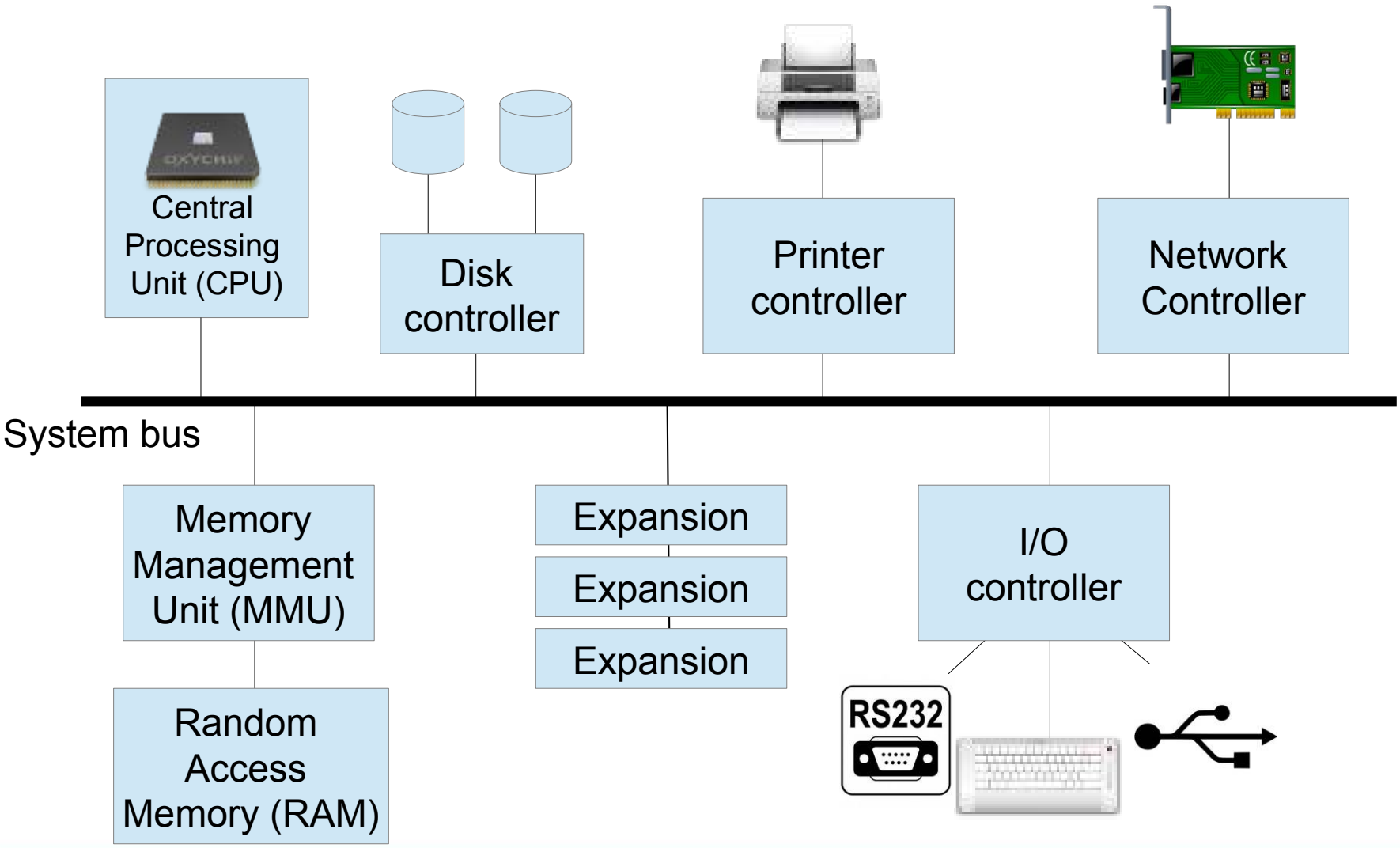


Introduction

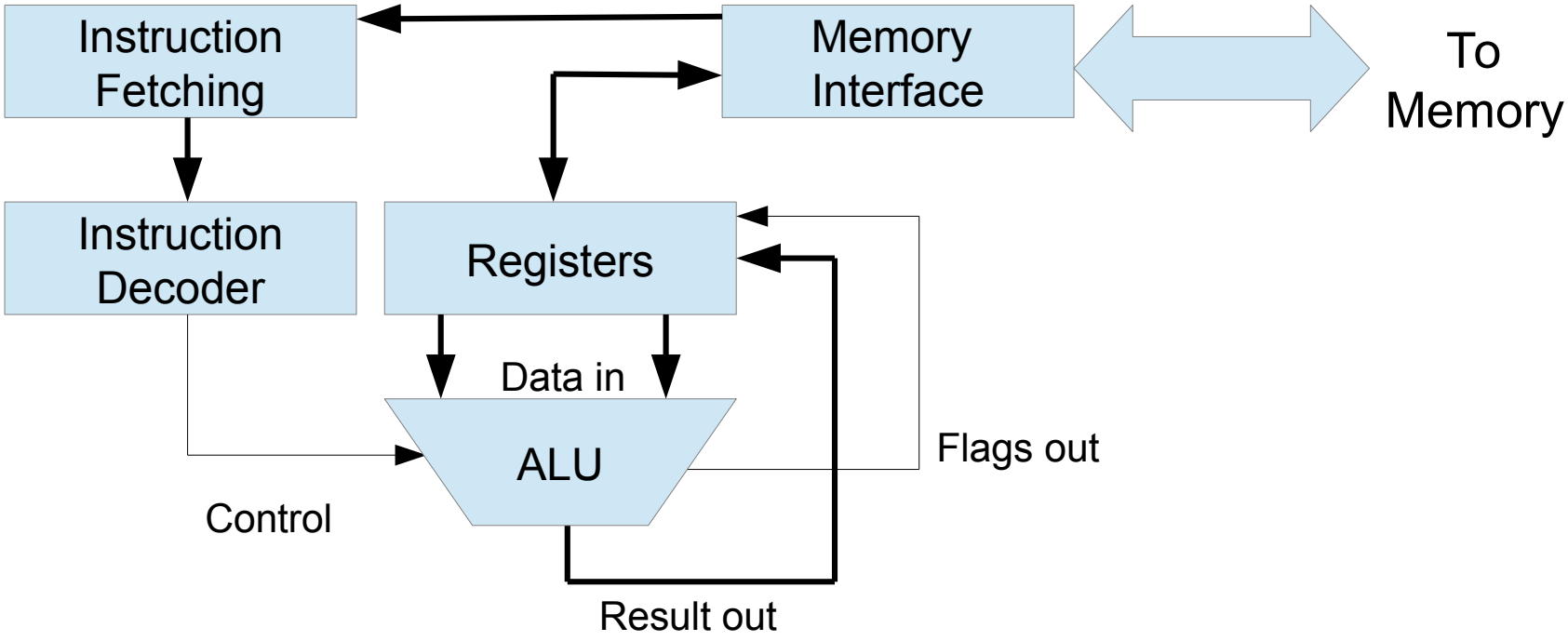
- **Computer Science –** The interdisciplinary study of computation, information and automation.
  - Algorithmics,
  - Computation theory,
  - Information theory,
  - Mathematical modelling,
  - Systems theory,
  - Electronics,
  - Engineering,
  - Computer architecture
  - ...

PART 0: What are computers made of?

Computer system hardware basics



Central Processing Unit basics



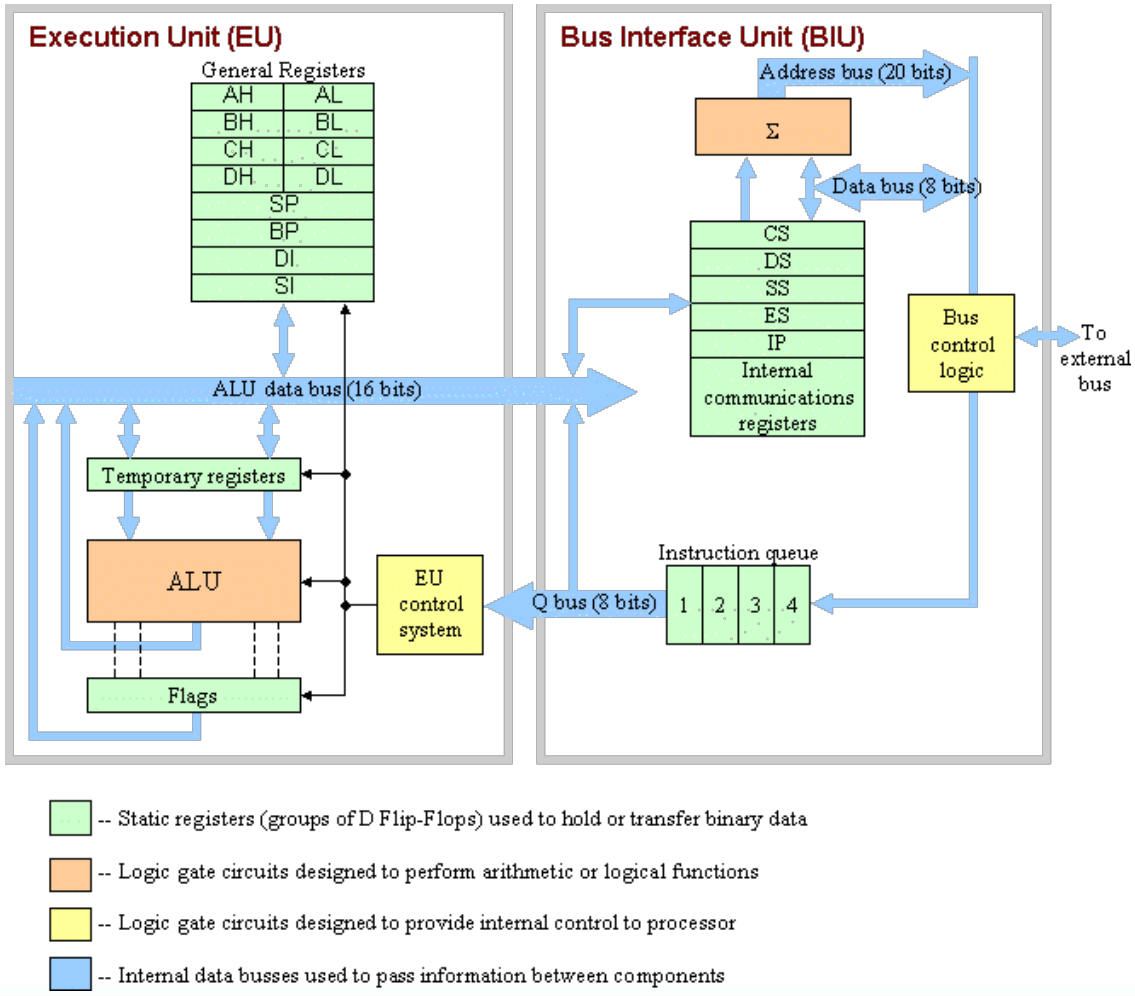
An oversimplified CPU diagram.

CPU building blocks

- **ALU** – Arithmetic-Logic Unit – this block executes arithmetic (like add, subtract, division, modulo etc.) and logic (eg. AND, OR, XOR, etc.) operation on input arguments stored in registers and outputting the result into registers.
- **Registers** – are a storage space for data.
  - **Flags** register is a special register storing information about the recent state of the ALU/CPU (e.g. "result is zero" flag, "divided by zero" flag, etc.)
  - **Program Counter** or **Instruction Counter** is a special register storing current position in program.
  - **General-purpose** registers are for storing input and output data.
- **Instruction fetcher/decoder** – is getting instructions from memory, increasing PC, feeds them to ALU.
- **Memory Interface** interfaces the CPU to the outside world.

Central Processing Unit basics

- A (very simple) real-world CPU (8088):



Source:http://faculty.etsu.edu/tarnoff/ntes2150/uproc/arch8088.htm

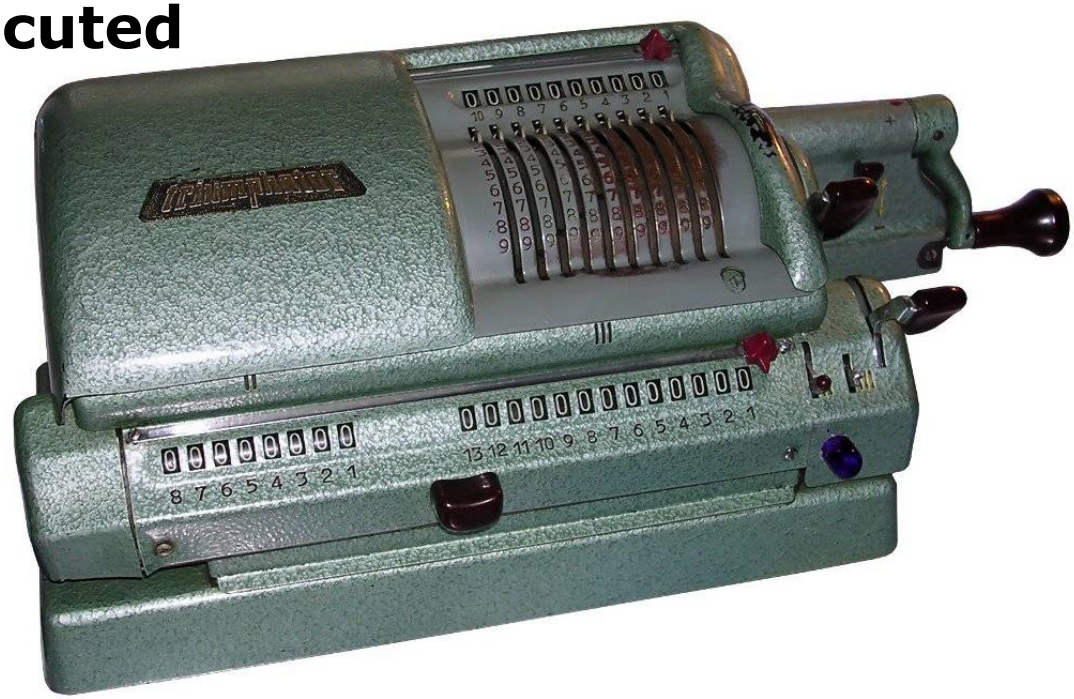
PART 1: History of the computer

A brief history of computers
1. Prehistory

- The main objective of computers was **to automate the calculation process.**
- Until 20th century the calculator was a position of an employee, who performed calculations using **arithmometer**.
- Calculations were performed according to the **"program"** and the results were verified and applied.
- This situation was known since at least mid-1800s – when simple arithmometers and calculating aids became accessible.
- But is it possible to **automate** the process **entirely**?

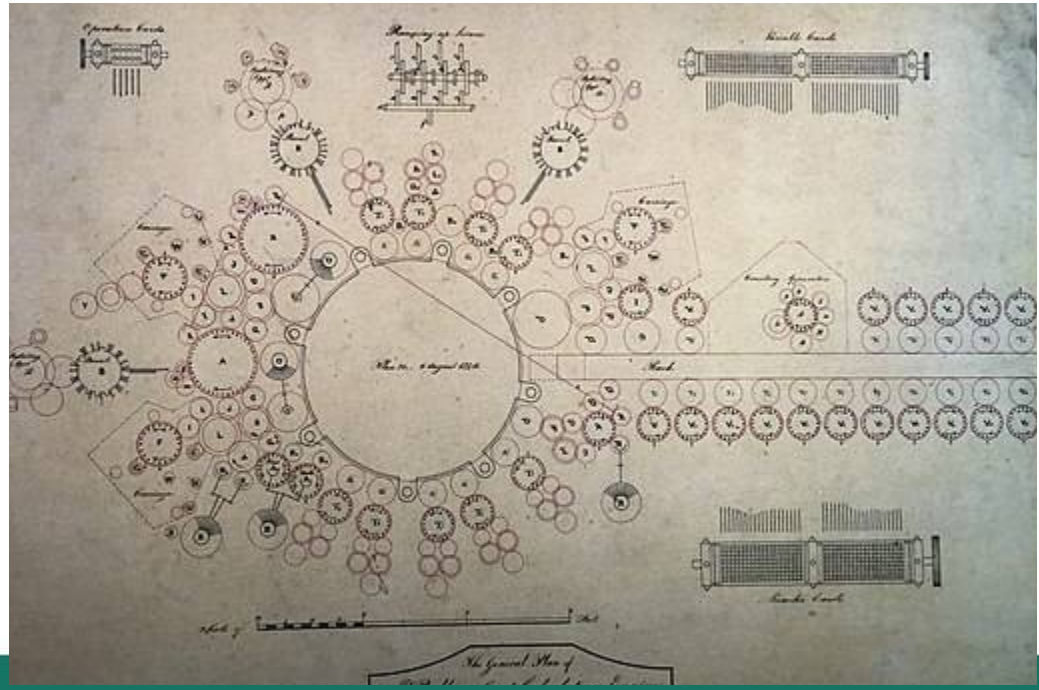
Prehistory: The arithmometer

- ...is a device which allows to perform basic arithmetic operations on numbers.
- Although operation on these is quite demanding, it can be seen that **an algorithm can be described using a list of operations and conditionals** which can be **blindly executed on the machine** to get the result.
- So the machine has a **set of commands** and the operator can extend it by actions like, for example, "**repeat ... until ...**"



Controlling the arithmometer

- 1837: Charles Babbage – "Analytical engine" – a design of mechanical general-purpose computer which used perforated cards as a program medium. Never fully built (but reproductions exist).
- 1920s – algorithms designed to be used specifically in programmable arithmometers (e.g. cracovian calculus, numerical integration).
- Until 1940s – dedicated, partially-programmable devices.



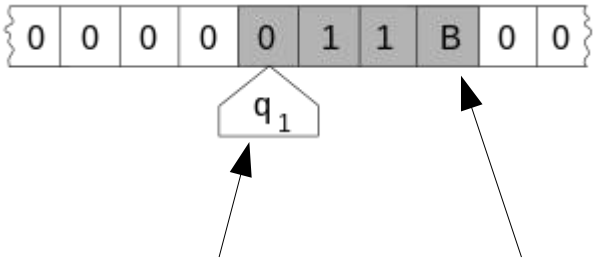
Turing machine (1930s-1940s)

- What is a **smallest** thing which makes a computer?

...an unlimited memory capacity obtained in the form of an infinite tape marked out into squares, on each of which a symbol could be printed. At any moment there is one symbol in the machine; it is called the scanned symbol. The machine can alter the scanned symbol, and its behavior is in part determined by that symbol, but the symbols on the tape elsewhere do not affect the behavior of the machine. However, the tape can be moved back and forth through the machine, this being one of the elementary operations of the machine. Any symbol on the tape may therefore eventually have an innings.

Alan Turing, "Intelligent Machinery", 1948

- Memory
- Registers
- Instruction set
- State



Notice the "program" vs its data!

What Turing machine needs?

- A finite set of **states** it can be in (q_0, q_1, q_2 …) .
- A finite set of **symbols** it can read and write to the tape, (e.g. 0, 1, Blank).
- A special **blank symbol** for non-used tape cells.
- The input symbols of the machine's alphabet (e.g. 0, 1).
- A **transition function** which determines how in the current **state** and current **tape symbol** the machine will react:
  - Move to new state
  - Change tape symbol
  - Move the head left/right.
- The **initial state**.
- A set of final states for the problem we're trying to solve.

What Turing machine can do?

- When we decide about transition function, we can **program** it.
- If the function would replace symbols with previous ones, it can, for example, **sort** – implement an algorithm.

- It can then be proven that Turing machine can execute a program on its data.

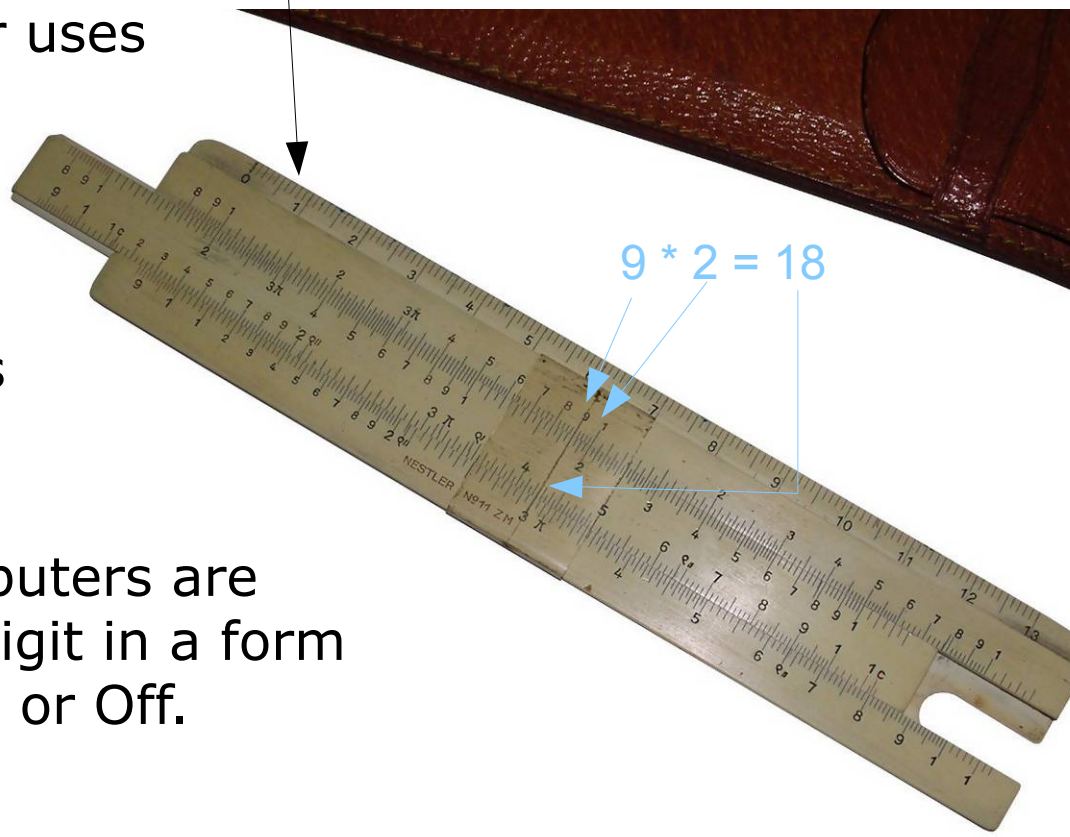
- So **every real-world design** of a computer **can be simulated on a Turing machine** with a very complex, but obtainable, states and functions. This has nothing to do with speed or being easy to program.

- Today, if a programmable system or programming language can simulate a Turing machine rules
(except the infinity of the tape of course) it is called **Turing-complete.**

Analog, discrete, digital, binary...

- The first machines exploiting physical properties, like length, elongation, mass, pressure or electric voltage/current, are using **continuous** variables and are called **analog.**

- The pin-wheel arithmometer uses a **discrete** number of steps (10 per digit – 0..9), so a digit can be represented as one of 10 states.
- Such devices are sometimes called **digital** devices.



- However, most current computers are **binary** – they represent a digit in a form of **binary** digit – 1 or 0, On or Off.

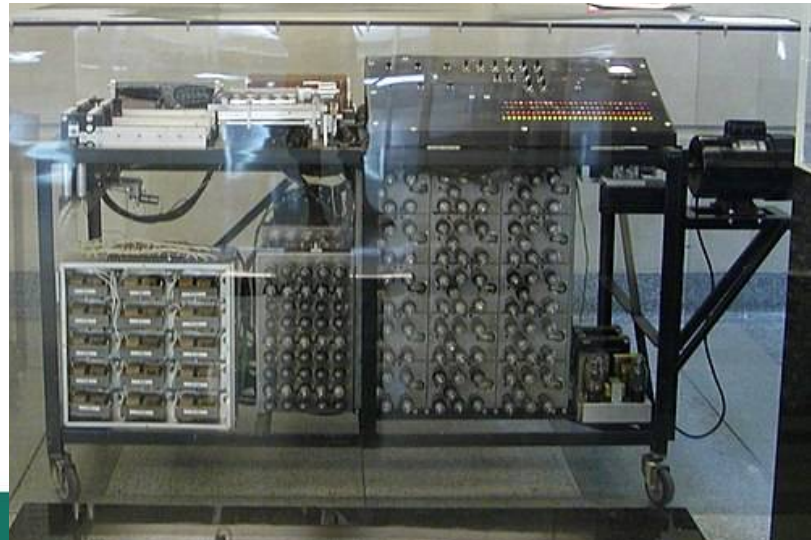
Binary calculus

- 1850s - George Boole - Binary logic - “Boolean algebra”.
- 1937 - Claude Shannon - “ A Symbolic Analysis of Relay and Switching Circuits” - The initial theoretical work of digital circuit design.
- 1937 - George Stibitz - Working 1-bit adder using electric current and 2 multi-contact relays (Model K).

- 1953 - Maurice Karnaugh - The “Karnaugh Map” - the method to describe any logic equation using logic functions which can be built using logic building blocks.

Electronic computers

- Relay-based, then tube-based, then transistor-based computers used a **binary, decimal,** or, more rare, **biquinary** (one of five digit for 0..4, and one of two choosing is it 0..4 or 5..9) number representation. It is just more simple for implementation and the binary logic is well defined.
- 1936 - The first complete electronic ALU – Arithmetic-Logic unit – known as Atanasoff-Berry Computer (ABC).
- 1941 – Zuse Z-3 – the machine operates on high-level programming language.
- 1943 – Colossus Mark 1 – Electromechanical.
- 1943-45 – ENIAC – fully programmable, tube-based, decimal.



First computers: ENIAC, 1940s

- Can we use more counters in the arithmometer?
  - Yes - there were such arithmometers for matrix calculations – one crank, 10 arithmometers for example.
- Can we make a super-fast, electronic ring counter?
  - Yes - with electron tube technology.
- Can we improve this "super-arithmometer" somehow to automate human operator?
  - Yes – we can switch which counters to transfer, from which counter and **when** – e.g. when some other counter reaches zero.

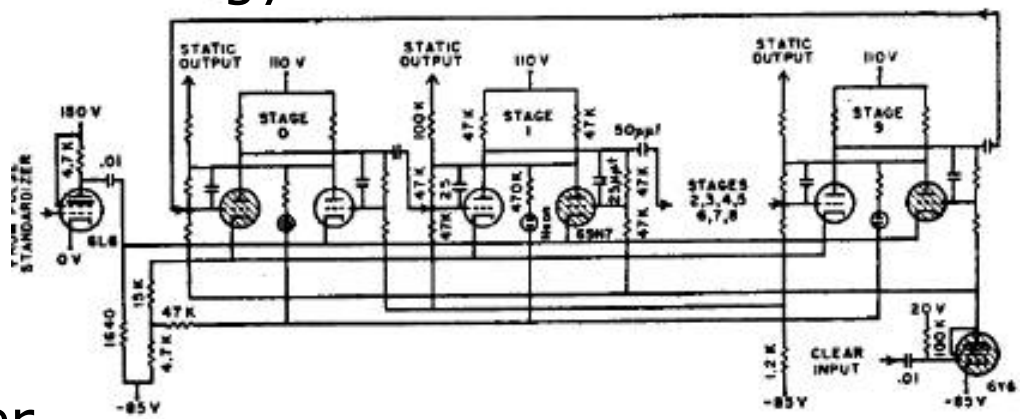
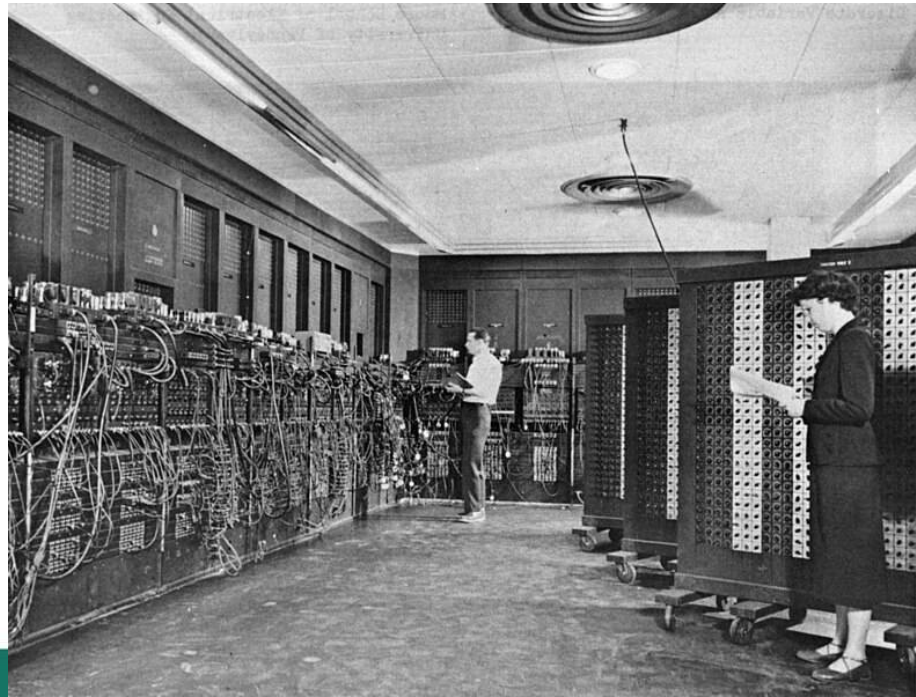


Fig. 1—Decade ring counter.

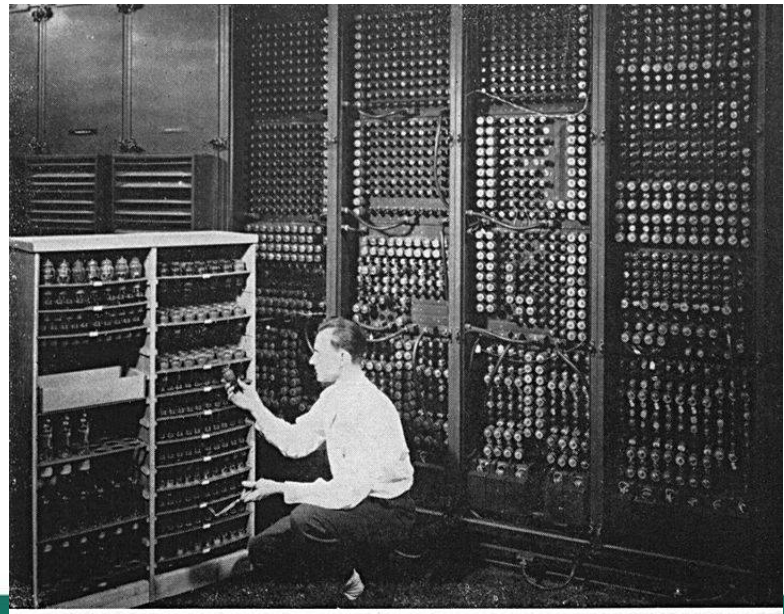
First computers: ENIAC (1940s)

- ENIAC is not the first computer, but is the first **fully programmable** one**.**
- The program was entered by switches,
- The data was entered by the “feeders” – bank of constants (also switches), which could “spin” a 10-digit number to any counter inside the machine.
  - Later, some values could be quickly “blown in” using perforated cards.
- Transfer which counter to which and where – defined by **function table**.
  The **program**.



ENIAC – the reality

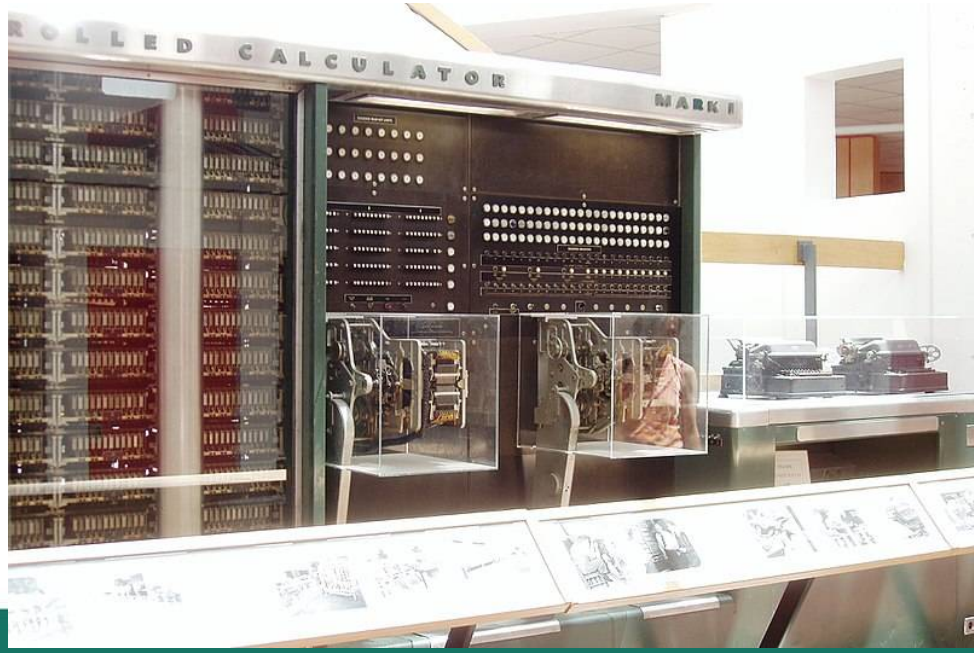
- Electronic Integrator and Computer.
  - J. Eckert, J. W. Mauchly, 1943-45. University of Pennsylvania.
  - Used for calculating ballistic curves, meteorological predictions, material calculations.
- It was given as a project assumption to abstain from using relays. So they used ~18800 vacuum tubes.
  - Vaccum tubes must be hot to work. This needed 140kW of power.
  - If the voltage was not right → tube blown → computations useless.
  - However, the device could "step", but the "step" was in fact one transfer of the number...
  - ...and the ENIAC could transfer in parallel, between multiple blocks.
  - So there was no such thing as "clock cycle".



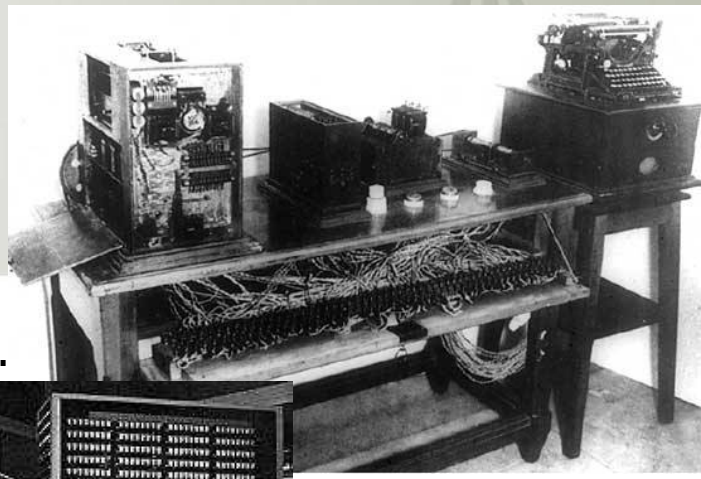
Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

Other machines, before ENIAC, programmable ones.

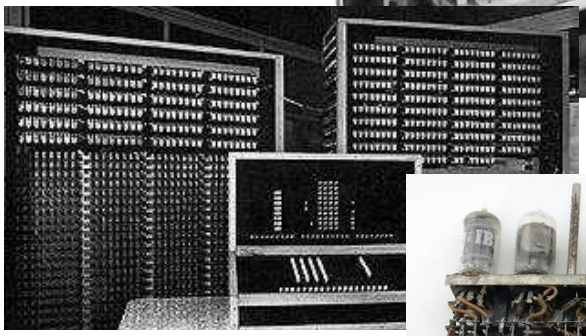
- Konrad Zuse's Z3 – Germany, 1941 – fully programmable, relay-based.
  - Notoriously underfunded, used a few times for aviation-related calculations.
- Colossus – Great Britain, 1943 - serial machine, programmable, tube+relays+Strowger counter based.
  - Used for cryptoanalysis.
  - Considered as fully "electronic"
  - Kept secret unti 1970s.
- Harvard Mark 1 – 1944, USA – fully programmable, relay-based
  - Program in the main memory,
  - No support for loops (Z3 and Colossus had it)
  - 765000 relays.



Evolution of electronics

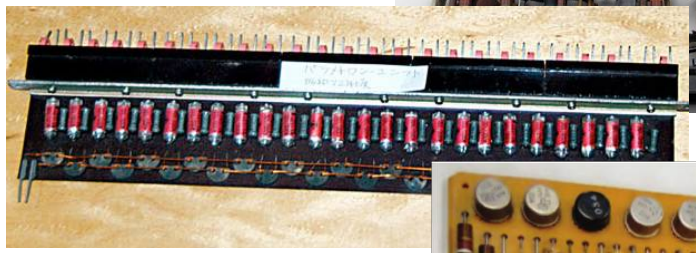


1) Mechanical-electromagnetic based devices (e.g. Leonard Torres y Quevedo's calculator). - 1920s-30s.



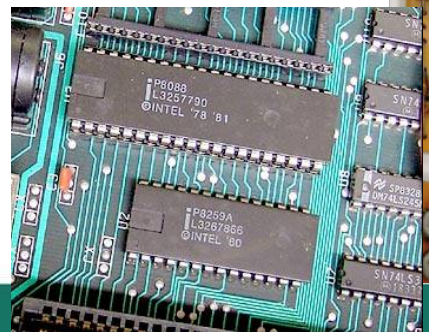
2) Electromagnetic relays (e.g. Zuse's Z2 computer, parts of the first Colossus). Slow and error-prone. 1930s-40s.

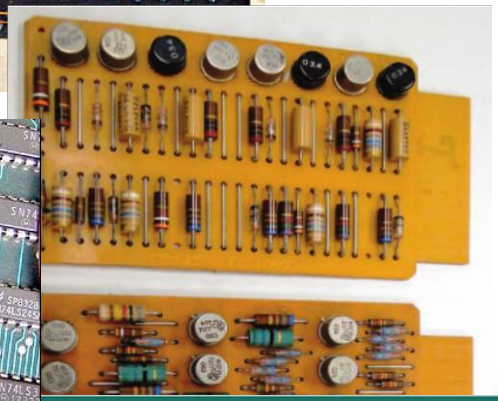
3) Vacuum tubes (e.g. ENIAC) – faster, but more power-consuming and unreliable – late 1930s-1950s.



4) Electromagnetic cores – Complex in synthesis, difficult in assembly (e.g. Ferranti Orion, Parametron) – late 40s-late 50s.

5) Transistors – since 1950s.

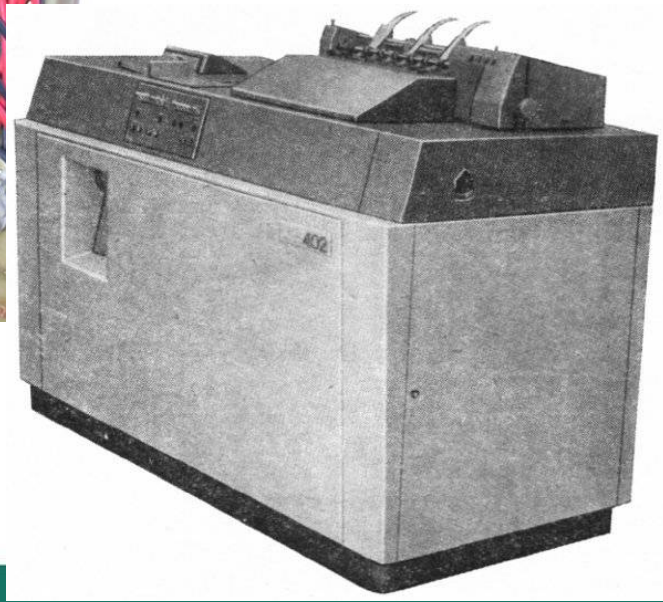




6) Integrated circuits – since late 60s.

Do we need a computer to process data?

- For some time, it was much easier to process data with non-programmable (or programmable in a very limited way) machines.
- These were much more accessible, easier to operate and were built to specific order.
- Although these machines were automatic, they are usually **not considered computers.**
- Finally, they became more and more general-purpose, fusing with emerging minicomputers.



Further development

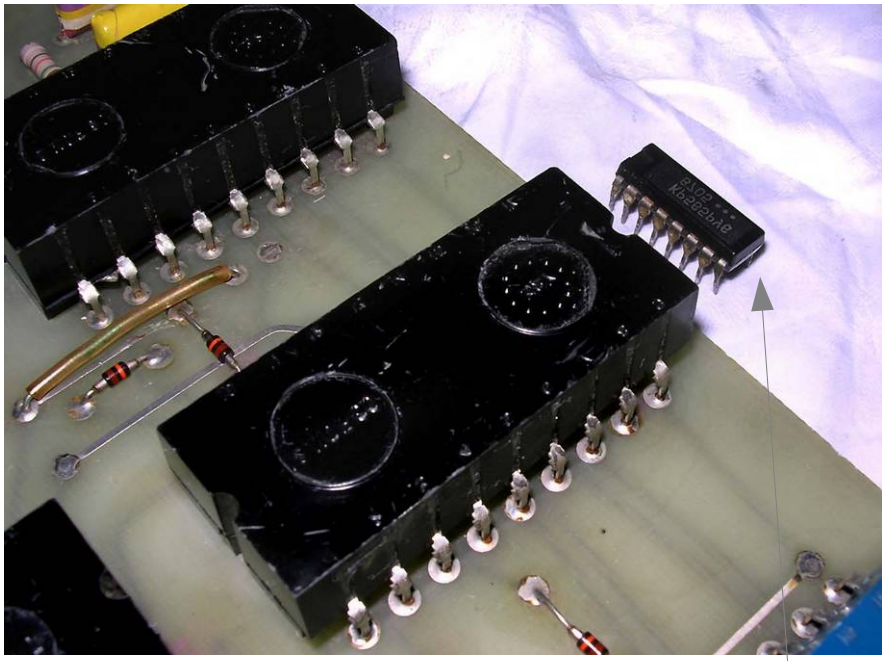
- Invention of a **transistor** allowed to abandon the difficult bi-quinary or 2/5 coding, focusing on better representation of numbers in a word. Computers became much smaller, more reliable and power-efficient.

- Theoretical description of an **integrated circuit** (Ambroziak et al, 1955) and its production (J. Kilby, 1958) allowed to make computers even smaller.

- In 1970s, computers got more popular, however, most foundations of computer science have been developed in computers early age.

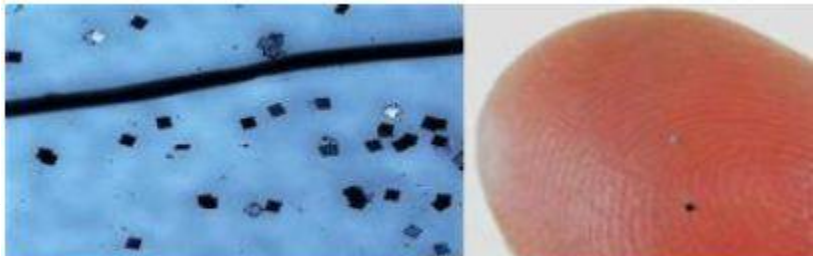
Moore's law (1965)

- The number of transistors in the IC doubles about every 2 *(or 1.5 – depending on interpretation)* years.

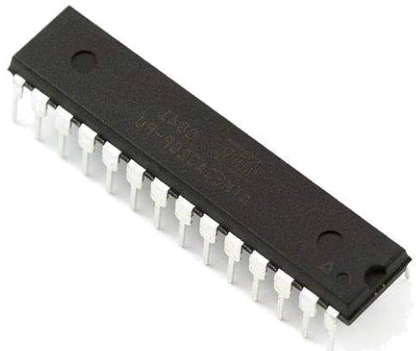


196x: 6 T/chip

198x: 10 000 T/chip
Smaller chips



200x: >100 millions T/chip,
even smaller devices.



199x: >million T/chip

Minicomputers

- In 1970s, widely accessible integrated circuits allowed to construct an entire computer of base building blocks – the **logic gates**.
  This started an era of minicomputers.
- Minicomputers had own instruction set, specific architecture, and were built with hundreds of small integrated chips.

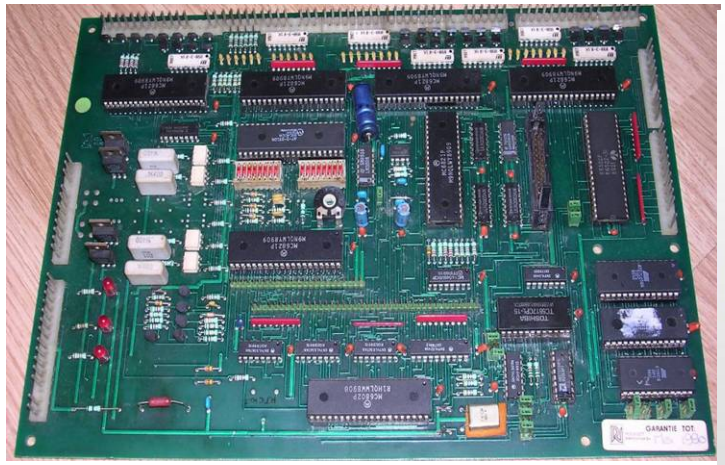
- The memory, initially ferrite-core, has become a semiconductor in mid-70s.

"There is no reason anyone would want a computer in their home."
Ken Olsen, founder of DEC, 1977

- These machines started to be used as a programmable logic control units, general-purpose data processing machines and were widely accessible.

Microcomputers

- Invention of the **microprocessor** – a chip containing a whole CPU, and sometimes even additional circuits, allowed to make computers cheaper and more accessible.
- In a few years, they became more and more cheap.
- In 1980s, popularization of home computers and BASIC programming language allowed many people to have, use and program computers at home.



Universal control board to build Into any machine.

1980s industrial PLC

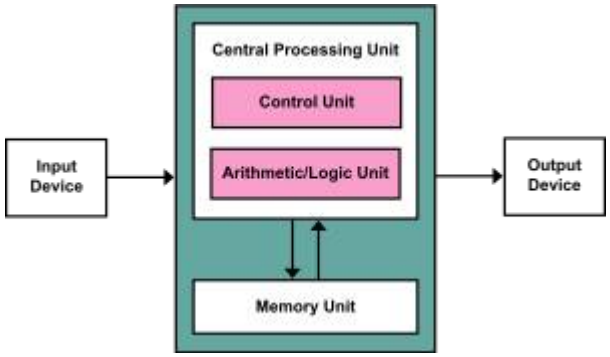
1980s home microcomputer

A case of the microcomputer

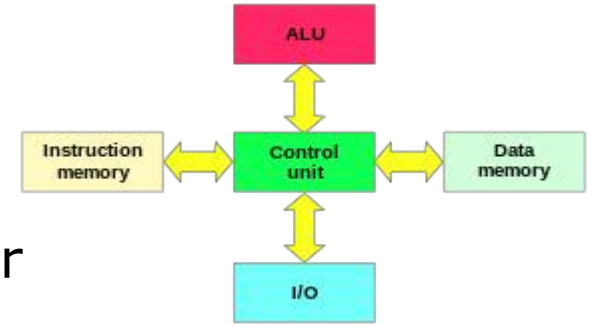
- In 1981, IBM's ESD – Experimental Systems Division, redesigned an IBM data terminal to be used as a computer.
- The design was open for expansions and well-described in the documentation.
- It used Intel 8088 8-bit microprocessor and had 64kB of RAM at first, later expanded to maximum of 640kB.
- It was called "IBM 5150 **Personal Computer"**
- Later, lots of PC clones dominated the market.

Computer architectures

- In von Neuman computer, **data and program** resides in the **same** memory.
- It is then possible to modify program by program, to dynamically program the machine, but it is more complex.



- The Harvard computer's architecture has **separate memory for program and for data.** This way it is more logically simple to implement, however, it has bigger footprint and is less flexible.



- Some modern microcontroller CPU cores are von Neuman in fact, but the short instruction counter makes them Harvard-like.

How the computers process data? Flynn's taxonomy (1)

- Single Instruction, Single data (SISD)
  - A sequential computer doing one program's step a time with one data portion a time.

- Single Instruction, Multiple Data (SIMD)
  - A computer doing a single instruction a time, but applies it in parallel to larger number of data.
    - Common CPU, different or split memory units (Arrays),
    - A set of CPUs, common memory unit (Pipelines),
    - Set of independent CPUs commonly controlled (Associative processors).

How the computers process data? Flynn's taxonomy (2)

- Multiple Instructions, Single data (MISD)
  - Many processing units perform various instruction on a single data in the common memory.
  - If the instructions are the same, we get **fault tolerance.**
  - If the instructions are different, we can see a lot of similarities with some aspects of modern AI solutions.

- Multiple Instructions, Multiple Data (MIMD)
  - Independent processors execute different instructions on different data.
  - Used in **distributed systems**.
  - Applied in multi-core processors.
  - Units may have a shared memory, or other equivalent communication method.

Part 2: How computers store numbers?

So, how computers store numbers?

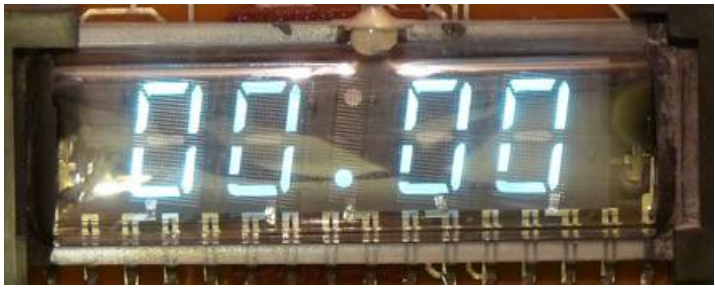
- We use a **decimal** system – it has a base of 10, so 0..9 – 10 digits.
- Computers operate on 2 states: 0 or 1, high or low – so a **binary** system is used there.
- Most numbers in computers have a **fixed bit width**. It means that the number may have a specific number of non-significant zeros to fill the space.

Binary encodings can be different

- When it comes to large numbers, methods of encoding can be different.
  - Historically, relay-based machines used biquinary format frequently because it was easier to implement using relay-based selectors.
  - If a small device performs operations in binary, but displays the result in decimal, it may be more handy to use BCD - encode **every decimal digit** using its own binary value.

BCD

- How many bits do we need to encode 0..9?
  - 0, 1, 10, 11, 100, 101, 110, 111, 1000, 1001 - 4 bits
  - The data buses in systems based on BCD seem to look unusually wide.



- However:
  - As we still operate in decimal, the "(0.3*3)+0.1 != 1" effect does not happen!
  - It is really, really, easy to encode separate digits to, e.g. 7-segment displays.
  - Notice that we can **expand** the code to full 4 bits to get one hexadecimal system digit.

Gray code

- Many older machines (and newer encoders) used switches to enter binary numbers. Switching, e.g., 1 to 2 required **two** flips:
  - 001 → 010 (rightmost bit, then middle are toggled)
  - During this switching, suddenly a stray 000 or 011 may appear, disrupting the machine's operation.
- Or when binary data is sent analog way, is it possible to **detect errors** when simply counting?

- The main question is: Is it possible to use a binary numeral system in which the number of **bit flips** when increasing binary value is **minimal**?

Gray code basics

It is possible to easily construct Gray code for **any** number of bits. We always start with a 2-bit list of permutations under a simple Gray code:

00, 01, 11, 10

Now reverse it:

00, 01, 11, 10 | 10, 11, 01, 00

Old values are starting with 0:

000, 001, 011, 010 | …

New will start from 1:

000, 001, 011, 010 | 110, 111, 101, 100

And we got a Gray code for 3 bits.
4, 5, 6 bits - the same way

Decimal	Binary	Gray
0	0000	0000
1	0001	0001
2	0010	0011
3	0011	0010
4	0100	0110
5	0101	0111
6	0110	0101
7	0111	0100
8	1000	1100
9	1001	1101
10	1010	1111
11	1011	1110
12	1100	1010
13	1101	1011
14	1110	1001
15	1111	1000

Decimal system

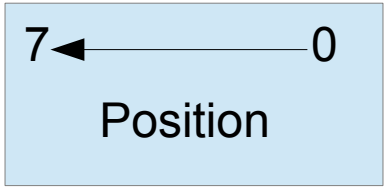
- 193 number in **base 10** is:

$$1 * 10^2 +$$
$$9 * 10^1 +$$
$$3 * 10^0 = 193$$

- Notice that we count the powers of the **base** from the **least significant** digit. It's similar in many CPUs.

Binary system

- We don’t have 0..9, we have only 1 and 0.
- So, let’s have a 8-bit byte: 11000001



1 * 2^7 +
1 * 2^6 +
0 * 2^5 +
0 * 2^4 +
0 * 2^3 +
0 * 2^2 +
0 * 2^1 +
1 * 2^0 = 193

Powers of 2:

0	1	2	3	4	5	6	7	8
1	2	4	8	16	32	64	128	256

Summing up

- Having a numeral system of a base **b**, the value of any number composed of digits:

  $\mathbf{d_n \ldots d_3 d_2 d_1 d_0}$

  can be described with the formula:

$$x = d_{n-1} b^{n-1} + d_{n-2} b^{n-2} + d_{n-3} b^{n-3} + \ldots + d_1 b^1 + d_0 b^0$$

- And we can use this scheme to **any** number system.

Operations on binary numbers

- We can add and subtract binary number digit-by-digit. The carry will be encountered more frequently than in decimal.
  - Note that 1+1=10, so it's 0 and carry 1.
  - During subtraction it's 0-1=1 and carry is 1.
  - Remember that limiting the value to e.g. 8 bits will **trim** the **overflow**!
  - ...and subtracting larger value from smaller one will result in **underflow**, and the carry will not disappear until the end of subtracting.

```
  1  11
 10100110
+  110011
----------
 11011001
```

```
  11  11
 10100110
-  110011
----------
  1110011
```

Operations on binary numbers

- The multiplication is analogical, however, there is one more trick:
  - Shifting bits by 1 to the right → division by 2 (rounds down).
  - Shift to the left → multiply by 2.

In C/C++, the bit shift operators are >> and << (until they're overloaded by stream operators)

Other number systems we may find

- A very popular **hexadecimal system**, is a base-16. Digits are:

  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

- Notice that 0xFF = 255, which is also 0b11111111, and in modern computers this 8-bit number is a **byte**.

- Quite rarely used **octal** system, fits to e.g. describe a byte in 3 digits.

- Ah, and 0x… is a hex number, 0b… is a binary.

- When we calculate probabilistic values, higher-base systems become useful as it is possible to represent permutations or combinations by just counting in these systems.

The byte

- Generally, the maximum value of a non-negative integer in binary of bit length **n** is 2^n-1.

- Maximum value of byte (8 bits) is then 2^8-1=255.
- Maximum value of an unsigned word (16 bits) is then 2^{16}-1 = 65535.

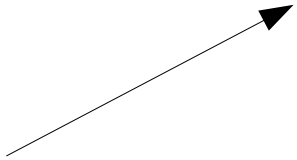
- If the value is **signed**, in modern programming languages it means that **there must be a bit(s) for the sign.**

A quick conversion base 10 → base ...

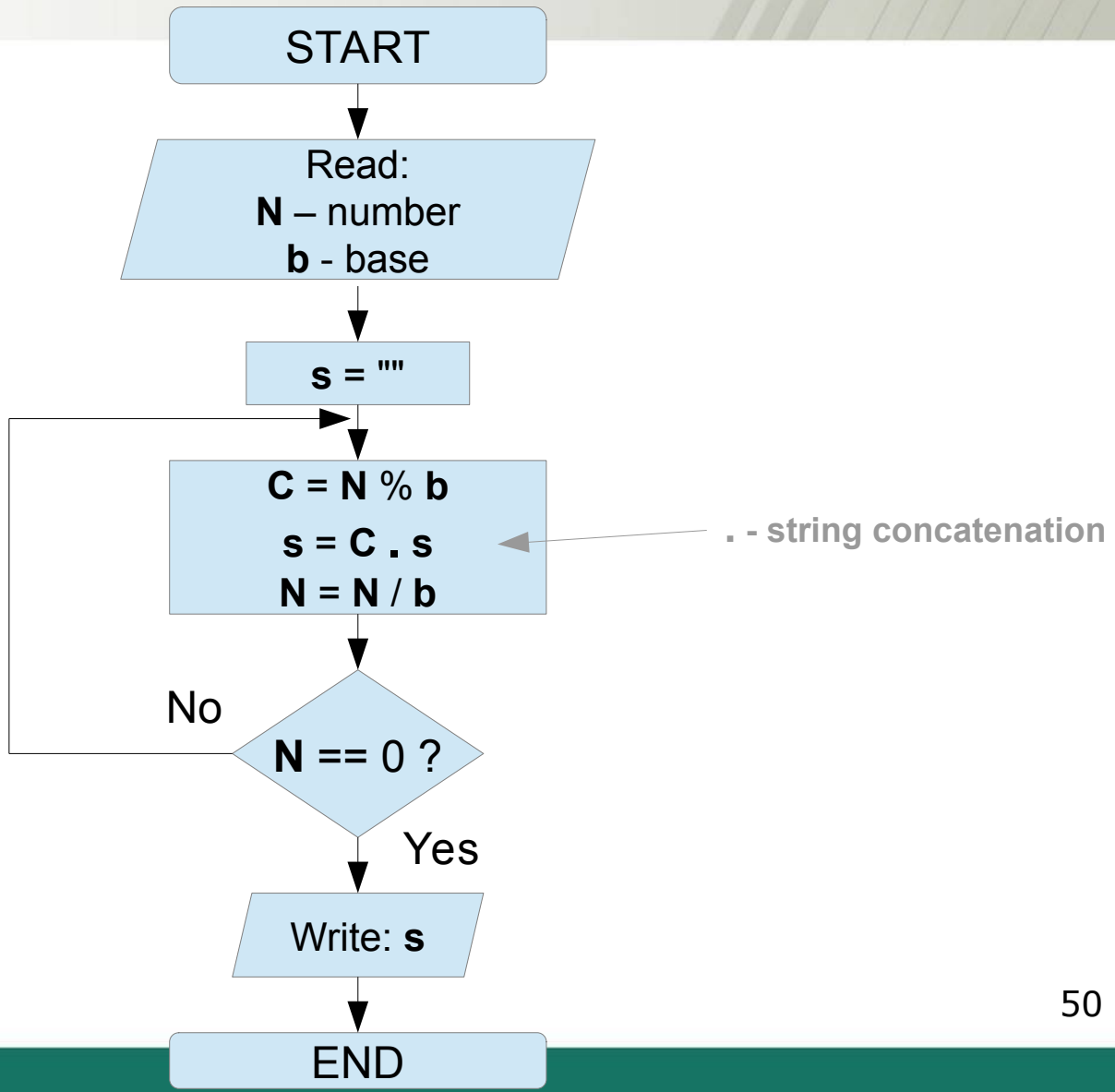
- And now we **divide without remainder.**
- Convert 18994 to hex:

- 18994 / 16 = 1187 18994 % 16 = **2**
- 1187 / 16 = 74 1187 % 16 = **3**
- 74 / 16 = 4 74 % 16 = 10 (**A**)
- 4 / 16 = 0 4 % 16 = **4**

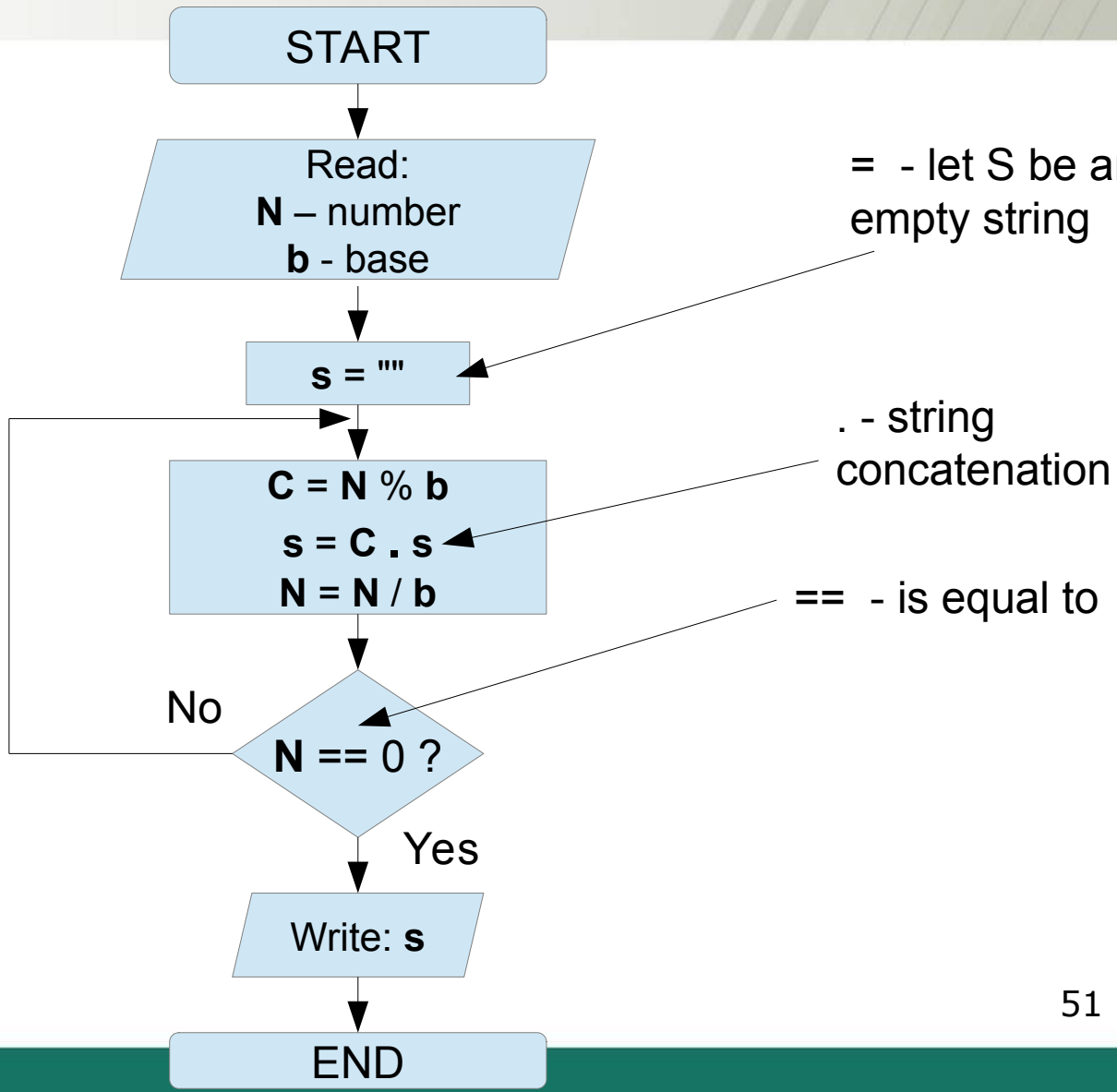
- The hex number is: 0x4A32



The algorithm:



The algorithm:



OK, that was for integers.

- How computer can represent a number with **decimal part**?
- The most simple approach: Use some digits for decimal part:

10^3 10^2 10^1 10^0 10^{-1} 10^{-2} 10^{-3}

2137,451

Integer part **Decimal part**

Now in binary

- Note that a base-10 digit in a decimal part **is usually not exactly a** binary value.
- We usually have a **fixed number of binary digits** for the remainder.
- It means that there are specific base-10 fixed-point numbers which **will not be reflected perfectly** using fixed-point binary.

It also means that further in programming, **you should not compare the floating-point numbers using arbitrary comparisons like ==.**
This may not work reliably.
Use x-y<threshold instead!

An example

- Convert decimal 37.21 to binary, 8 bits for the remainder.
  Integer = 37 Remainder = 21

  37 / 2 = 18 37 % 2 = 1
  18 / 2 = 9 18 % 2 = 0
  9 / 2 = 4 9 % 2 = 1
  4 / 2 = 2 4 % 2 = 0
  2 / 2 = 1 2 % 2 = 0
  1 / 2 = 0 1 % 2 = 1
- 37 in binary is 100101

An example

- Convert decimal 37.21 to binary, 8 bits for the remainder.
  Integer = 37 Remainder = 21
- 37 in binary is 100101

0.21 * 2 = 0.42 → 0
0.42 * 2 = 0.84 → 0
0.84 * 2 = 1.68 → 1 (so take the % 1)
0.68 * 2 = 1.36 → 1
0.36 * 2 = 0.72 → 0
0.72 * 2 = 1.44 → 1
0.44 * 2 = 0.88 → 0
0.88 * 2 = 1.76 → 1 reached the precision. It's **00110101**

Convert it back

00110101 =
$0 * 2^{-1} +$
$0 * 2^{-2} +$
$1 * 2^{-3} +$
$1 * 2^{-4} +$
$0 * 2^{-5} +$
$1 * 2^{-6} +$
$0 * 2^{-7} +$
$1 * 2^{-8} = 0.125 + 0.0625 + 0.00390625 =$0,19140625

It is not possible to reflect some values using such description methods. We ran out of precision.

How about negative numbers?

- The most simple – SM (Signed Magnitude): Make one specific bit (e.g. the most significant one) responsible for the sign.
- It is lit (1) – it's a negative. It's not lit (0) – the number is positive.

- A small problem: There is "-0" and "+0" and we cannot do anything with it.

- Because we symmetrically "mirror" the range, the **n**-bit word has a range:

From: $-2^{n-1}+1$

To: $2^{n-1}-1$

Problem with arithmetic

- With SM, you have to keep this "minus bit" all time somewhere and calculate its value additionally. This is time- and logic-consuming.
- Can we store the information about the negative number somewhere else?

- The solution is **1C system** – one's complement system.
- Let's assume that the most significant bit is a sign bit (1 – negative), but if a number is negative, other bits **are stored negated.**

Complementary 1C system

- It is then easier to perform arithmetic operations on these numbers:
  - The addition and subtraction can be done just in columns including the sign bit, however, if we carry beyond the sign bit, add 1 to the result.
  - Subtraction is made by adding the bit-negated value.

Code	Value
000	0
001	1
010	2
011	3
100	-3
101	-2
110	-1
111	"-0"

2C system

- If we add 1 after negating the bits in 1C, we will:
  - Be able to have a negative weight on a sign bit – arithmetic is a bit easier.
  - Instead of +0 and -0, the range will be shifted accordingly.
  - EXAMPLE: For 8-bit byte, it will be not -127..0..+127, but -127..0..128.

Now the numbers are out of order!

Base10	-3	-2	-1	0	0	1	2	3
SM	111	110	101	100	000	001	010	011
1C	100	101	110	111	000	001	010	011
2C	101	110	111	000		001	010	011

2C finally solves
the "-0" problem.

Now if we skip the sign, we will get the following base10 numbers of these codes:

	4	5	6	7	0	1	2	3

We don’t have the growing order of binary values and many operators would require troublesome sign decoding.

System with bias

- Notice the order of wrongly decoded values: 4, 5, 6, 7, 0, 1, 2, 3.
- Let's shift the values to the left to obtain the order: 0, 1, 2, 3, 4, 5, 6, 7.
- To do it, we have to **subtract** the specific constant called **bias.**

- To process the values with bias, it is needed to know:
  - The width of the word (in our example 3 bits),
  - The bias value – here **4** fits best.

Floating-point values

- We theoretically can emulate floating point operations with integer operations.
- The question "where to shift the decimal point" is solved as a sub-problem then.
  - ...which takes time!
- Modern computers have a hardware **floating point unit** which allows to calculate floating point variables.

- The measure of results "spread" of the same value representations is a **precision**. In floating point arithmetic, we talk about precision as a measure of the **detail**.
  - The **accuracy** is the measure of the results to the "real" value at all.

IEEE754

- Is „a set of representations of numerical values and symbols” - means how to store numerical values.
- The number can be described using:
  - A **sign** - written as a bit,
  - A **mantissa** - written as p bits,
  - An **exponent** written in the rest of bits for a word.

- The standard also defines +∞, -∞ and two „Not a Number” descriptions rarely used as intended.

Mantissa and exponent

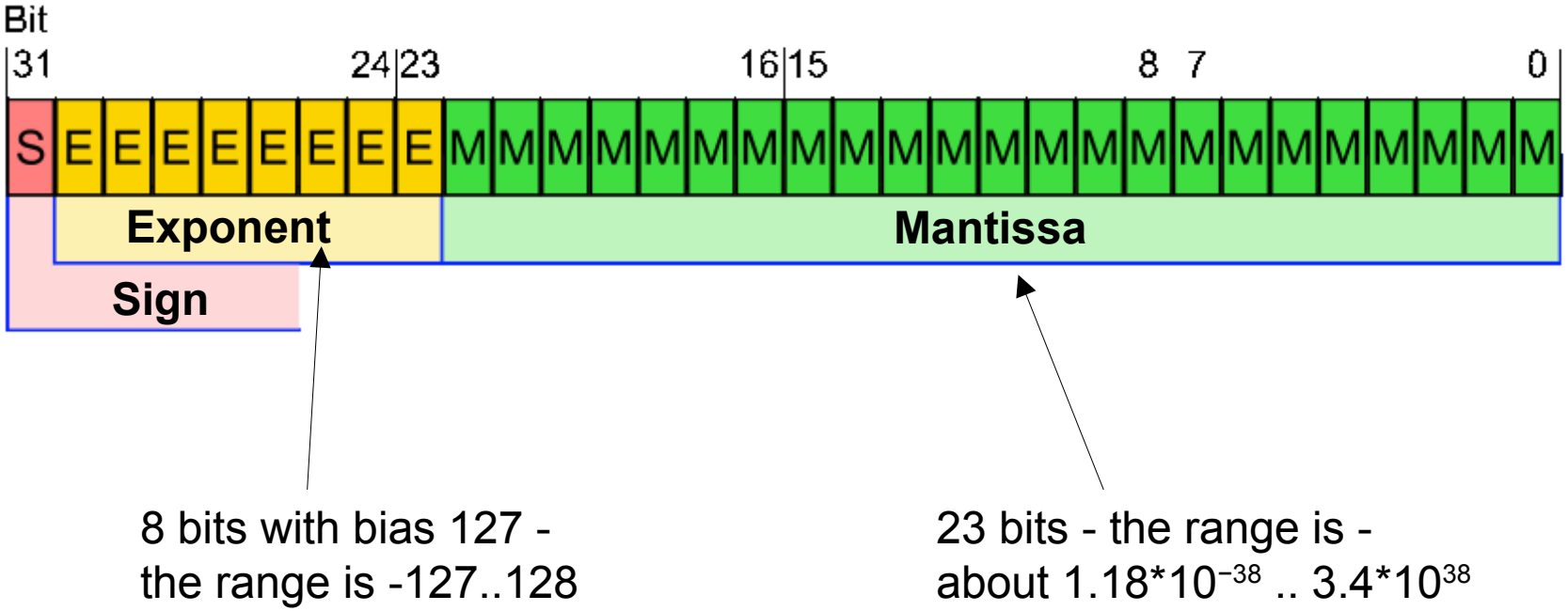
- 2137.451 can be written as:
  - $2.137451*10^3$
  - $21.37451*10^2$
  - $213.7451*10^1$
  - …

mantissa

exponent

- Although there is an ambiguity in binary description too, the value is chosen to fit into the mantissa part.

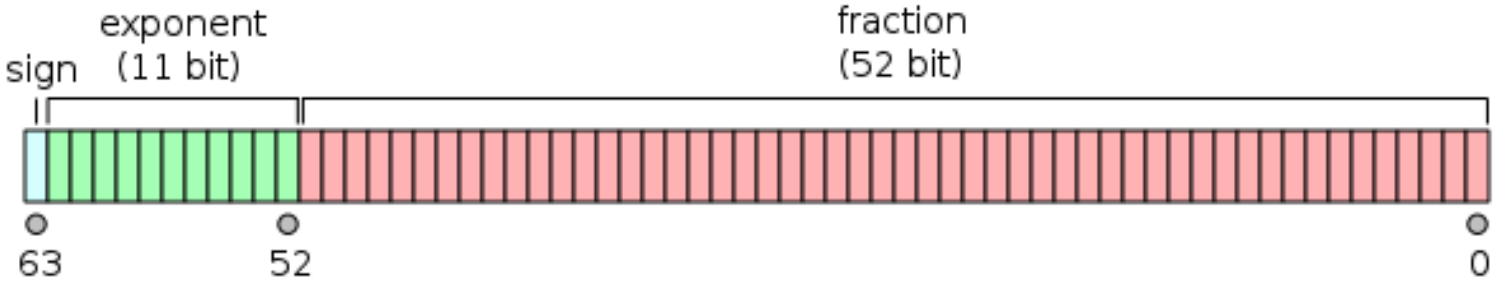
A IEE754 single-precision number



If this is not sufficient, there is also a **double** precision (64-bit wide) numbers.

Double

- If we need a better precision:
  - 52 bit for mantissa
  - 11 bit for exponent
  - 1 bit for sign



Additional types

- 80-bit floating-point – internal for floating-point operations inside Intel's FPU. Called "Extended precision".
  - Why do we need more precision for intermediate results?
  - How to organize storage of 80 bits in memory? - usually it is aligned to handy 96 or 128 bits.
- A full 128-bits "long double" – available in some compilers.

PART 3: A quick review of bitwise operators

Bitwise operators

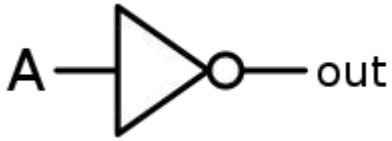
- Digital computers store and process information using binary digits: **bits.** The operators performed on bits are performed by the hardware on electric signals.
  - HIGH level – usually represents binary 1,
  - LOW level usually represents 0,

- ...which has nothing in common with activation of an electronic device. The device can be "Active low", means something turns on when the signal is low, not high.

- ...And in the electronics, the voltage levels of "1" and "0" may have totally different relation (RS232 – 1 is negative voltage, 0 is positive).

NOT - negation

- The NOT, bitwise complement is a binary operator that returns the complementary bit:
  - NOT 1 = 0
  - NOT 0 = 1



  - NOT 0100 = 1011
- Notice the leading zero!

- In C++, we negate by !
- In logic equations, usually with ~ or ¬

AND

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

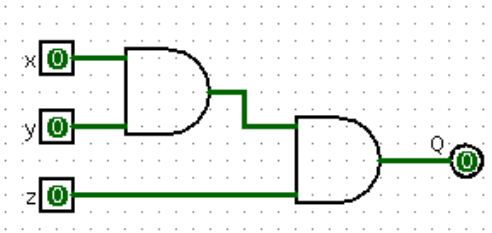
- Two argument AND operator:
  The result will be 1 if and only if all inputs are 1.

- Typical relations:

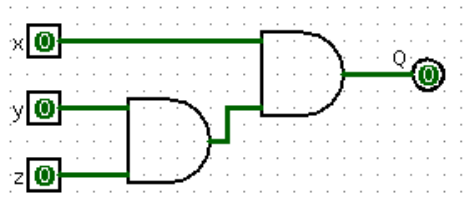
  x AND y AND z = (x AND y) AND z = x AND (y AND z)

- Notice the "masking" usage:

  0110 – is the 2th bit set?



==



	01**1**0
AND	00**1**0
	00**1**0 ← it is non-zero, the bit is set.

OR

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

- The result is 1 if **any** of the arguments are 1.
- Notice that OR-ing a word using "masking" allows to set bits without touching any 1s set.

x OR y OR z = (x OR y) OR z = x OR (y OR z)

but:

x AND (y OR z) = ((x AND y) OR (x AND z))

XOR

- Exclusive OR:

  1 only if inputs are different

- Notice that "masking" use of XOR allows to flip selected bit values in word. 1 becomes 0, 0 becomes 1 – without prior knowledge about value of this bit.

- x XOR x = 0
  - So a nice shortcut for zeroing a register...

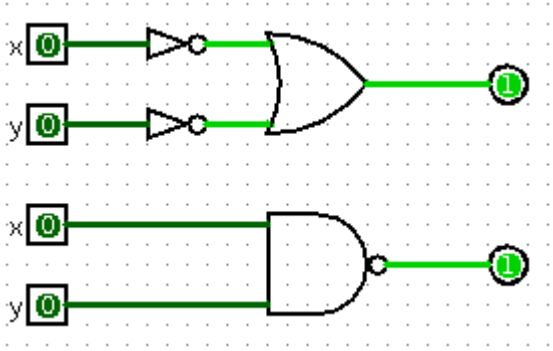
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Gates with negation in it
The NAND gate

A	B	A NAND B
0	0	1
0	1	1
1	0	1
1	1	0

- The output is 0 if and only if both inputs are 1.
- Used very frequently.

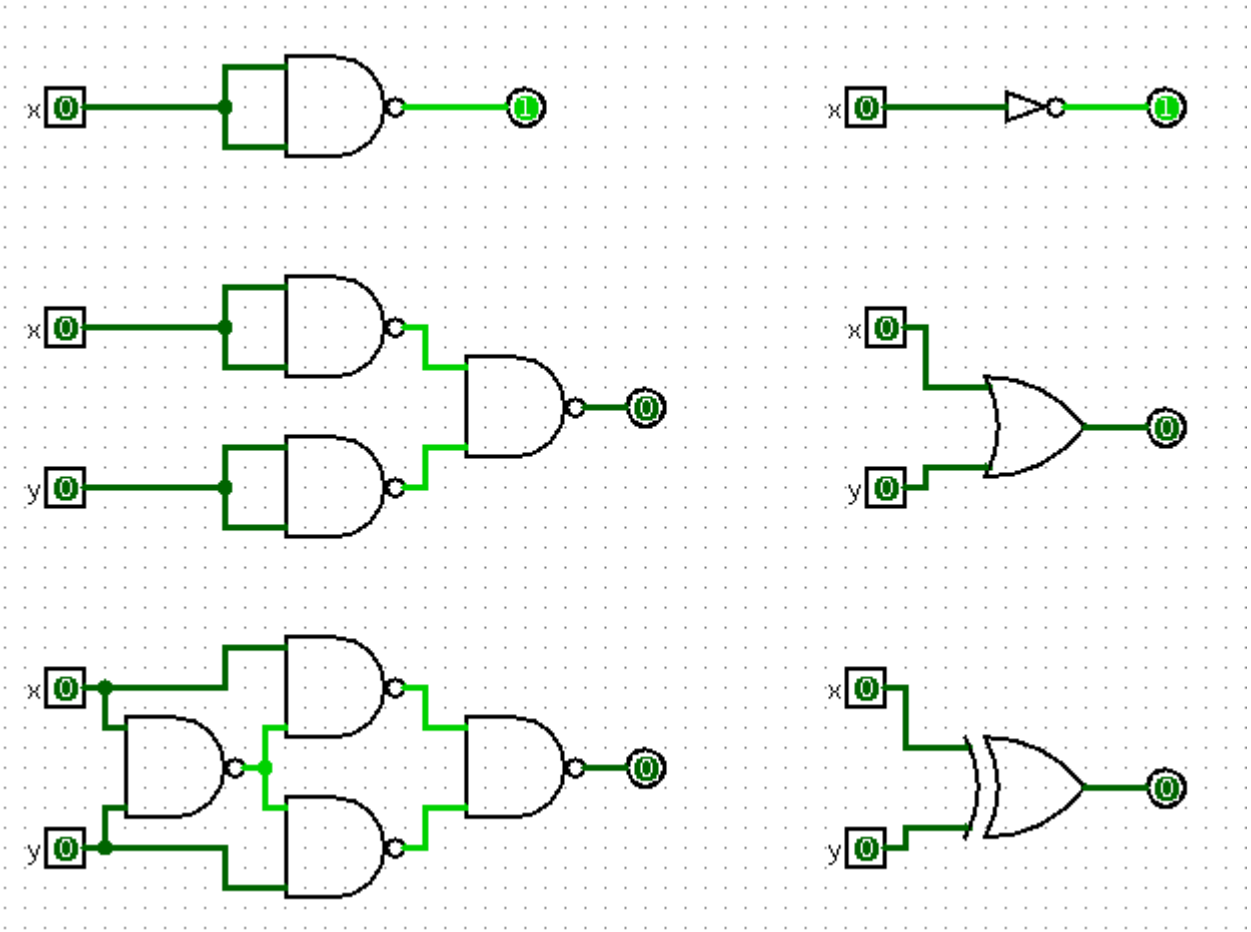
- Notice the application of DeMorgan's law:
  - A NAND B = !A OR !B



- **Functional completeness!**

These are logically equivalent

Functional completeness of NAND gate:

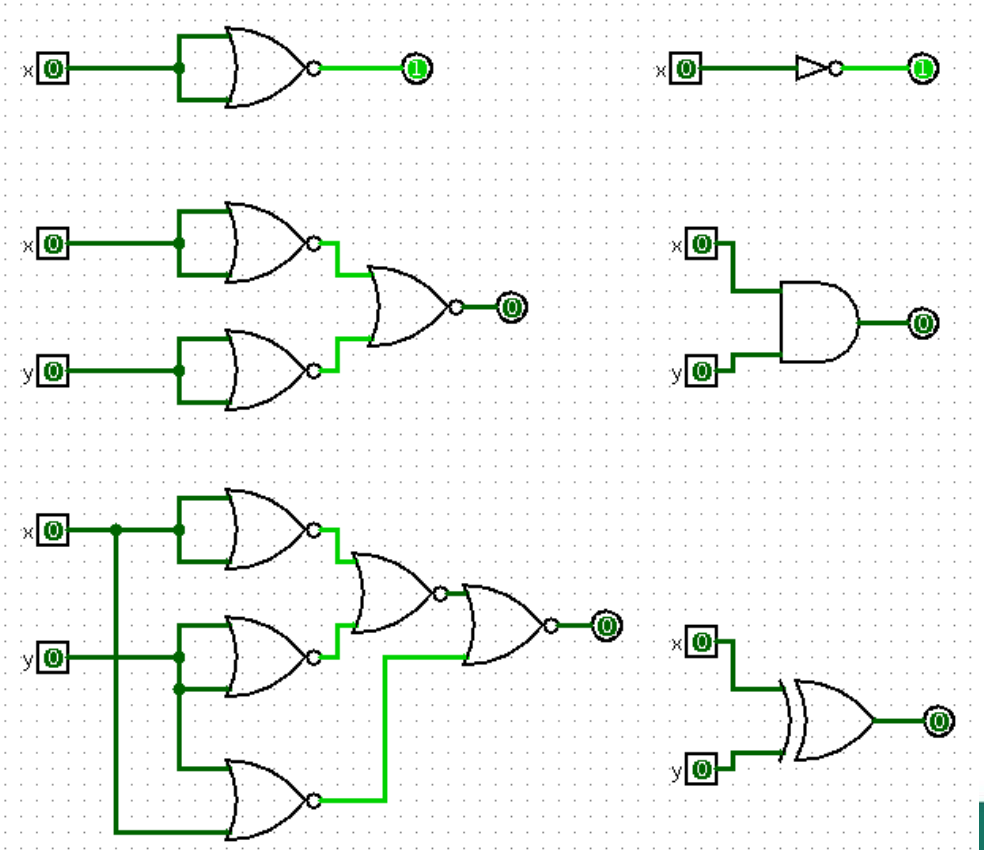


NOR Gate

- Generally, the HIGH is only when both A and B are LOW.

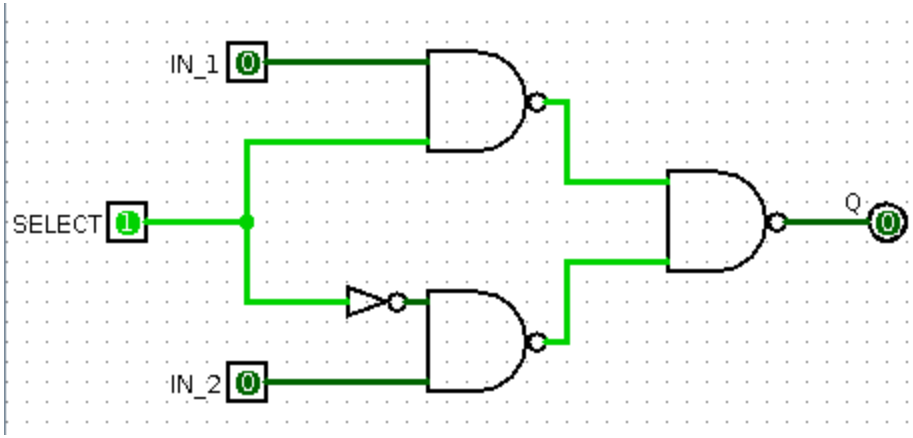
- Also functional complete:

A	B	A NOR B
0	0	1
0	1	0
1	0	0
1	1	0



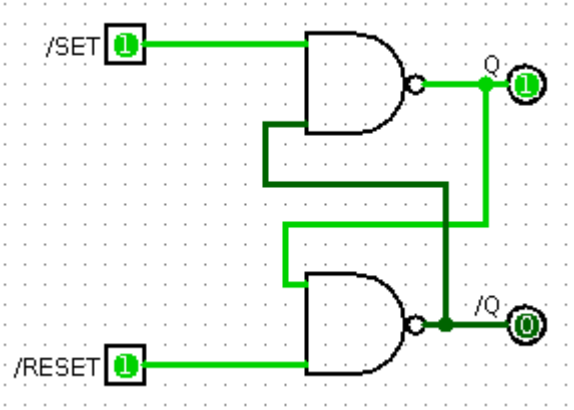
Another needed instrument

- Many times it is needed to implement the switching**.** The circuit used to do it is called a **multiplexer**.
- The 2-input multiplexer will pass the state from input 1 on its output, or the state from input 2, depending on a state of input **control line**.
- Can be made of gates:



How to remember the state?

- The most simple memory circuit is called a **flip-flop**.
- Can be done using NOR or NAND gates:

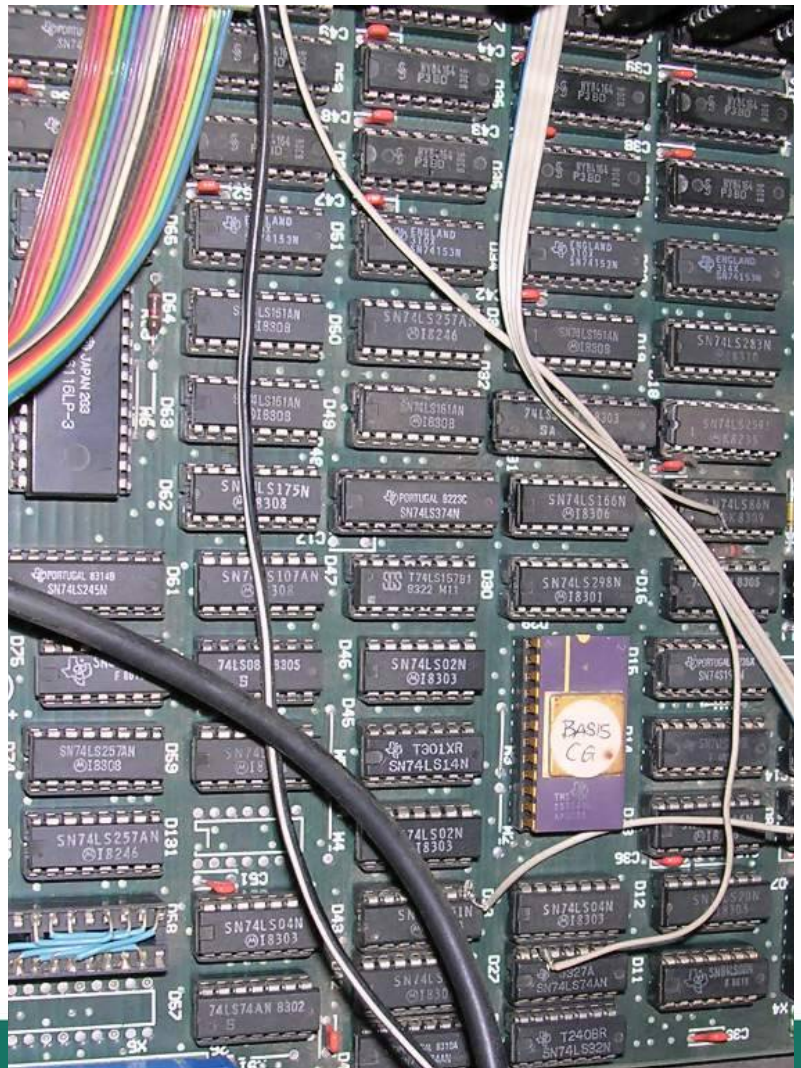


- The negative pulse on /S, toggles one state. Another pulse will do nothing on it.
- The state can be changed only with a negative pulse on /R – then, the state changes to the other.
- Always /Q == !Q.

- Not very cheap/simple/efficient as computer's RAM, but fast enough to work as some cache.

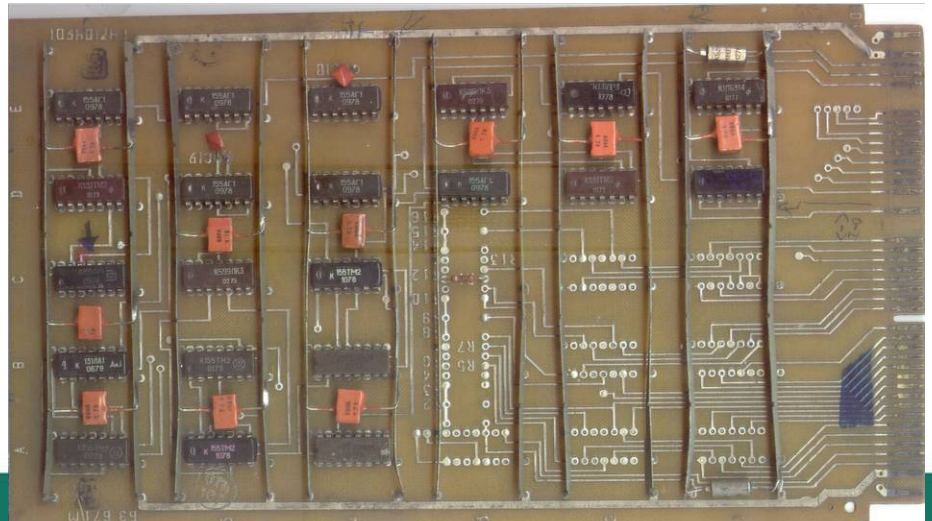
Implementation in electronics

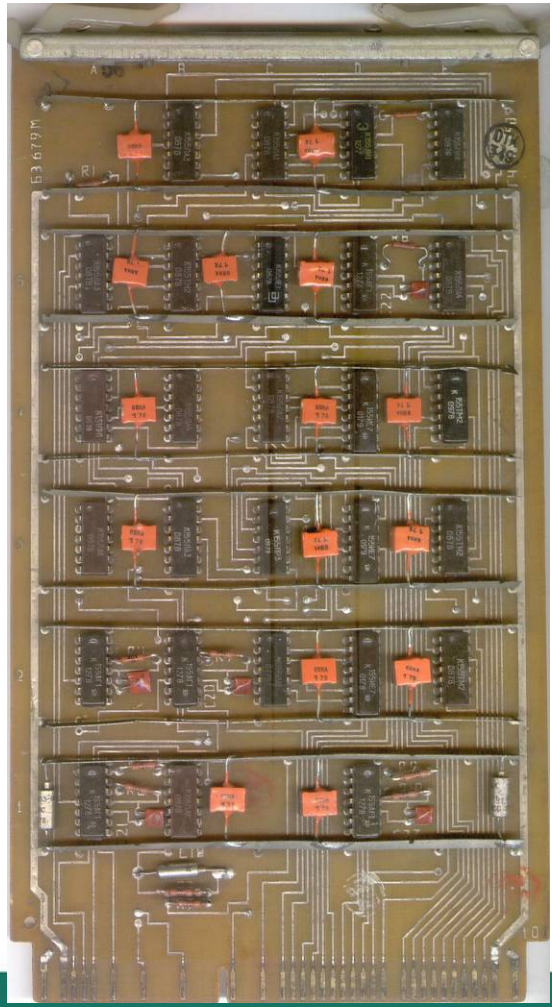
- Integrated circuits of 40xx or 7400 series contain logic gates with various number or inputs and various output characteristics, as well as triggers, counters, multiplexers etc.
  - Used even today, as a "glue logic".
- Modern CPUs – logic equations, written in HDL, are transferred to the CPU core design.
- Then, the design is equipped with I/O to outside world.
- This is designed as a processor.



A small bit of history

- A 1970s CPU contained a few tens of these functional boards made of 74xx TTL logic chips.
  - On the right – Subtract 11-bit numbers with "carry".
  - On the bottom, generate a FAULT signal when the number's even, but the parity bit states otherwise.
    (you needed two of these for a set)





AGH

PART 4: Now we can try to build an ALU

Let's build a simple ALU...

- We need the following features:
  - Add/subtract a 4-bit numbers
  - AND/OR the 4-bit words
  - Be controlled using 2 bits
  - Generate a CARRY, OVERFLOW and ZERO signals.

ALU components

- The adder/subtractor,
  - CARRY/overflow generator.
- The AND / OR module
- The (remaining) multiplexers
- The ZERO generating part,

How to add bits?

- Single bits

A	B	A + B	CARRY
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

INPUTS

OUTPUTS

This looks like XOR gate

This looks like AND gate

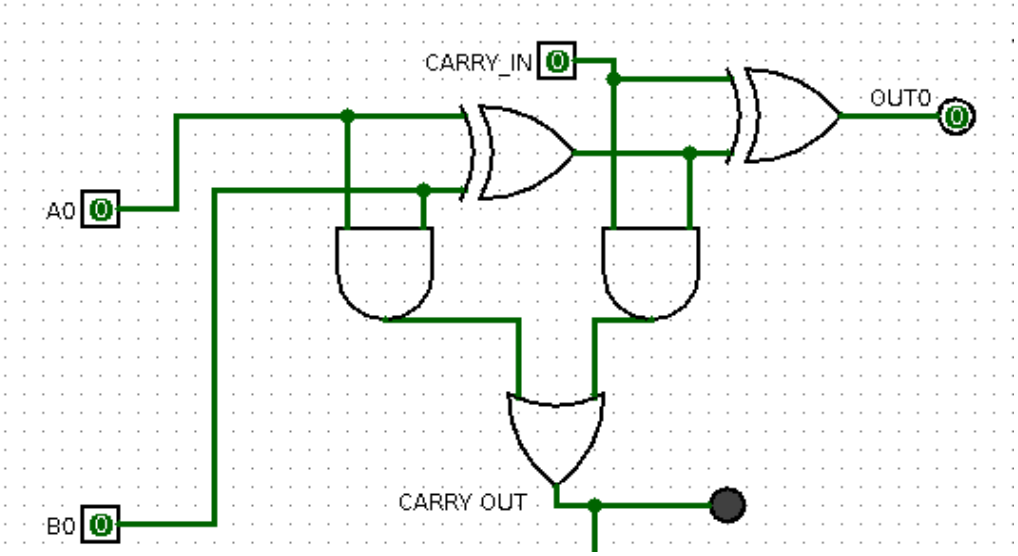
- With Carry in (C_{in})

A	B	C_{in}	A + B	CARRY
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

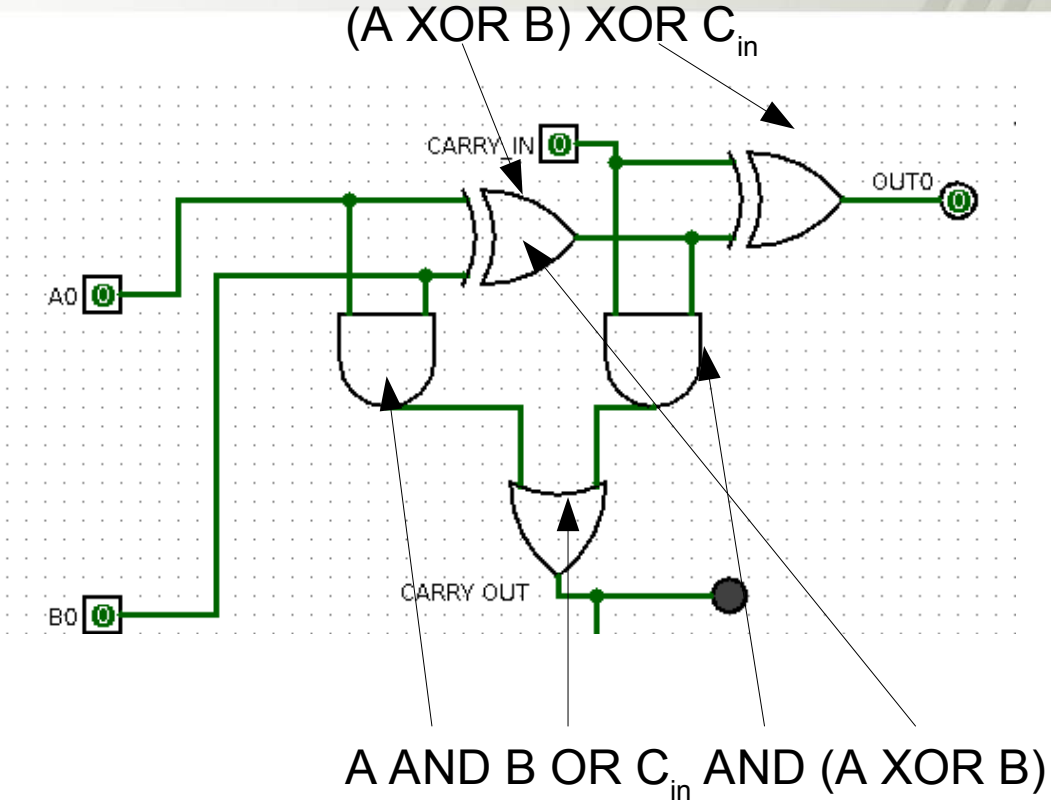
(A XOR B) XOR C_{in}

A AND B OR C_{in} AND (A XOR B)

1-bit adder...

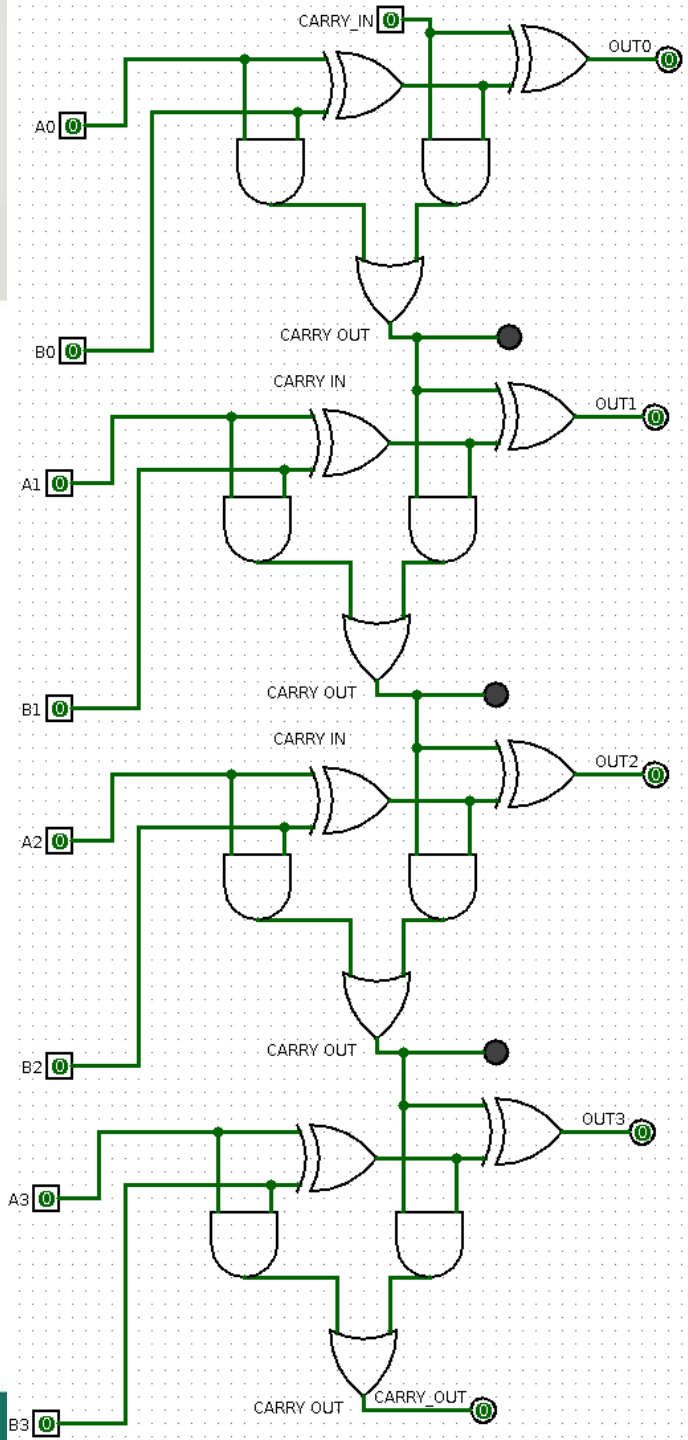


1-bit adder...


(A XOR B) XOR C_{in}
CARRY_IN
OUT0
A0
B0
CARRY OUT
A AND B OR C_{in} AND (A XOR B)

AGH

4-bit adder is just cascading the CARRY bits now!



Subtraction

- We know that A - B = A + (-B) .
- ...So if we **negate** one of the inputs, we will get subtraction instead of addition.
- In C2, we negate by **inverting a bit** and **adding one**.
- Finally:

$$A - B = A + (-B) = A + (!B) + 1$$

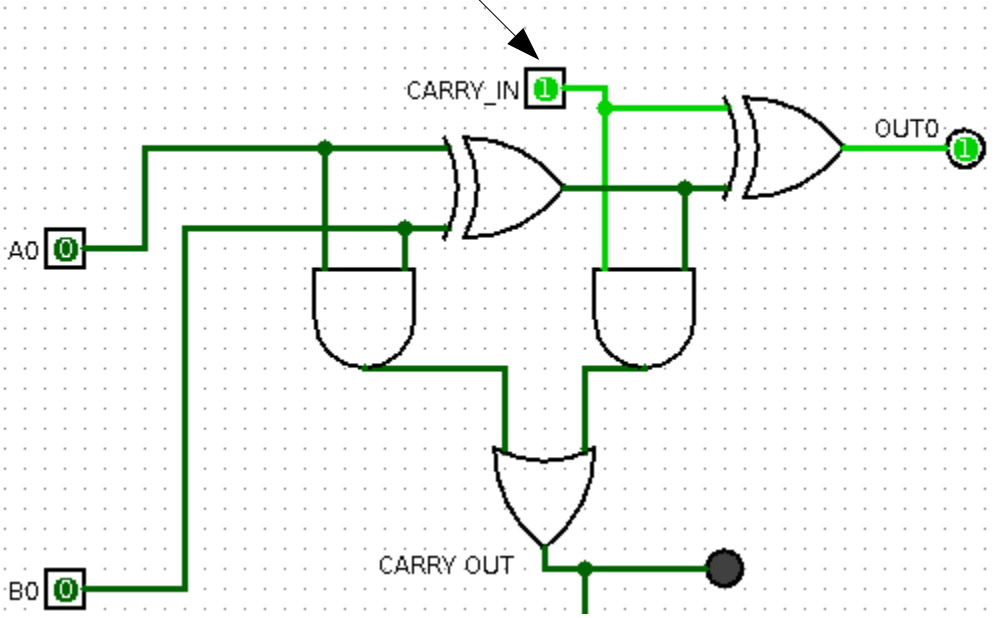
Logic negating,
in C++ notation.

Connecting the dots

- How to add 1 to our project?

If we force initial CARRY to 1, it will work like 1 has been added!

A	B	C_{in}	A + B	CARRY
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1



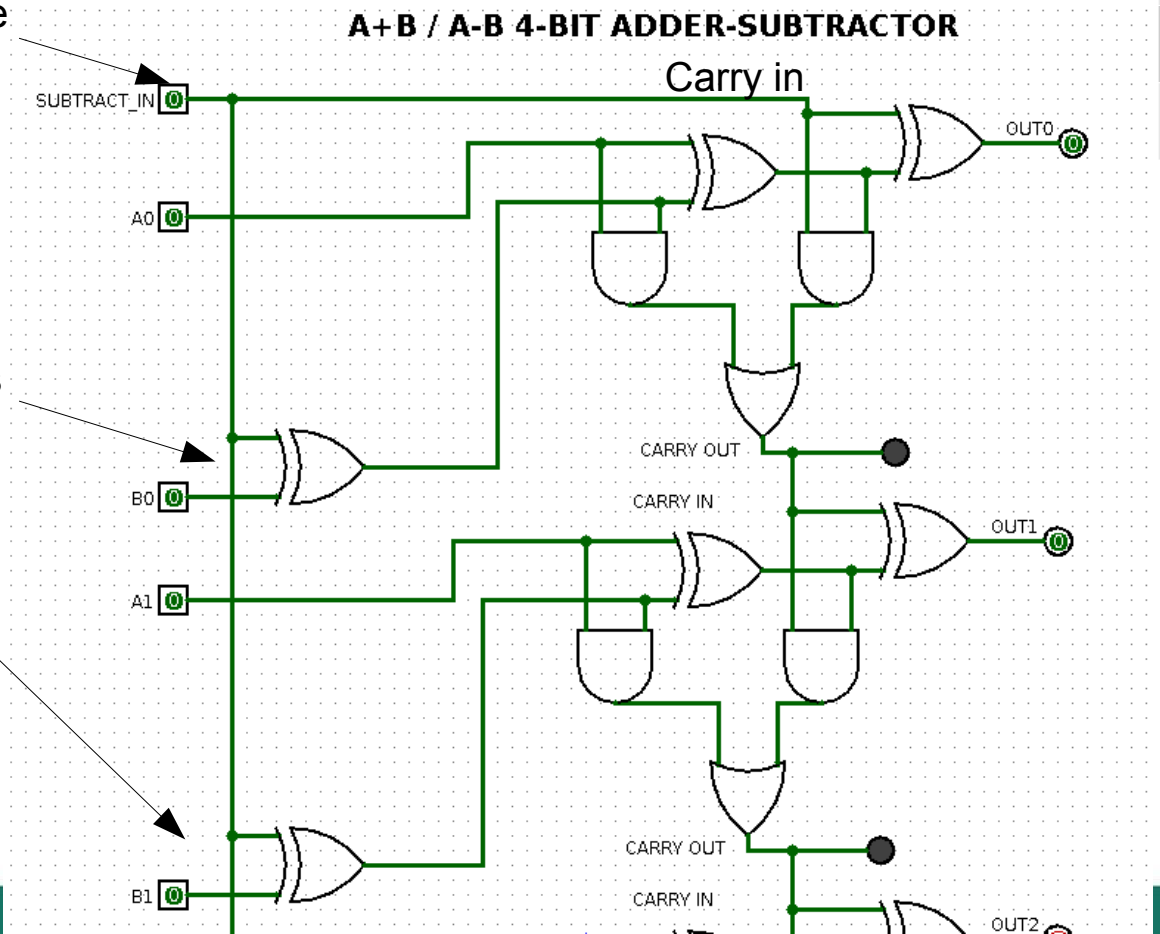
Connecting the dots

- Now we will conveniently invert one of the inputs using a XOR gate:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

If we put 1 here, we subtract. 0 will add

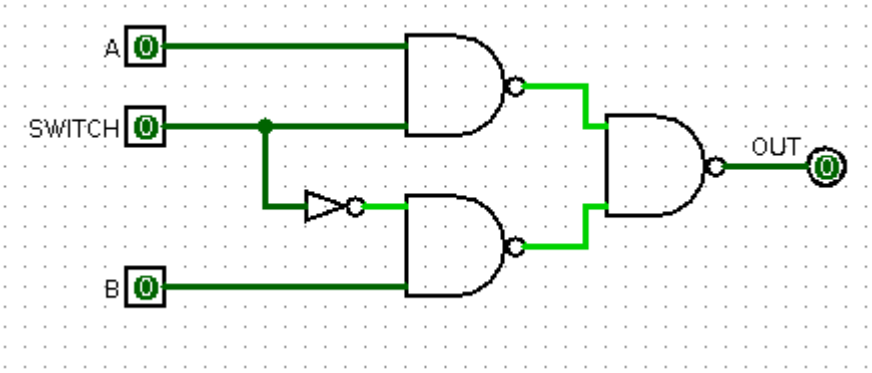
Added XOR gates



Switching the signals - multiplexer

- Notice that **depending on the SWITCH signal**, the circuit passes states from A or B inputs.
- This way we can introduce **control** to the ALU.
- ...and decide what command the ALU will do with input words.

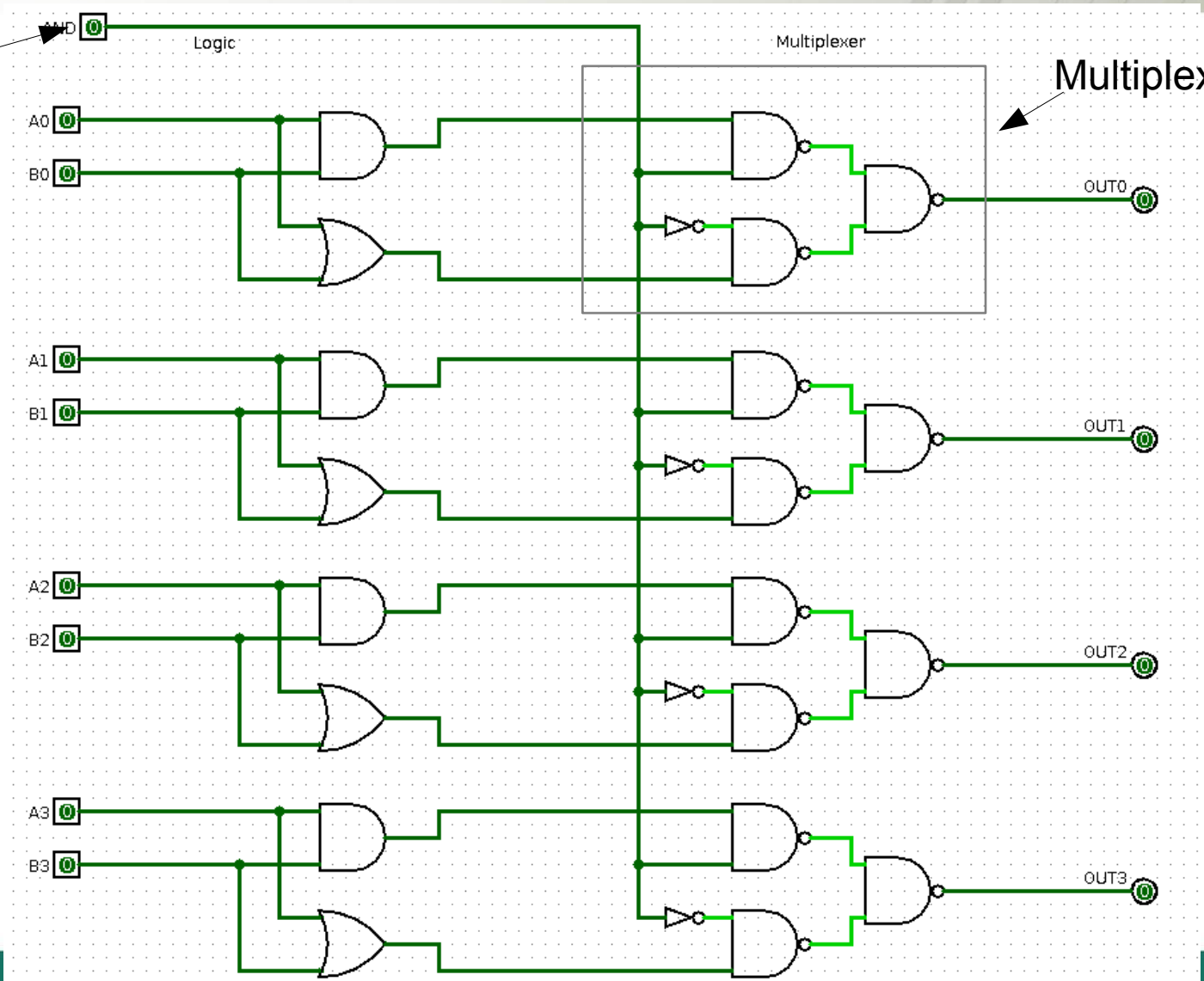
A	B	SWITCH	Q
0	0	0	0
0	1	0	1
1	0	0	0
1	1	0	1
0	0	1	0
0	1	1	0
1	0	1	1
1	1	1	1



A	B	SWITCH	Q
A	*B*	0	*B*
A	*B*	1	*A*

OR/AND?

Switch between OR/AND



Multiplexer block

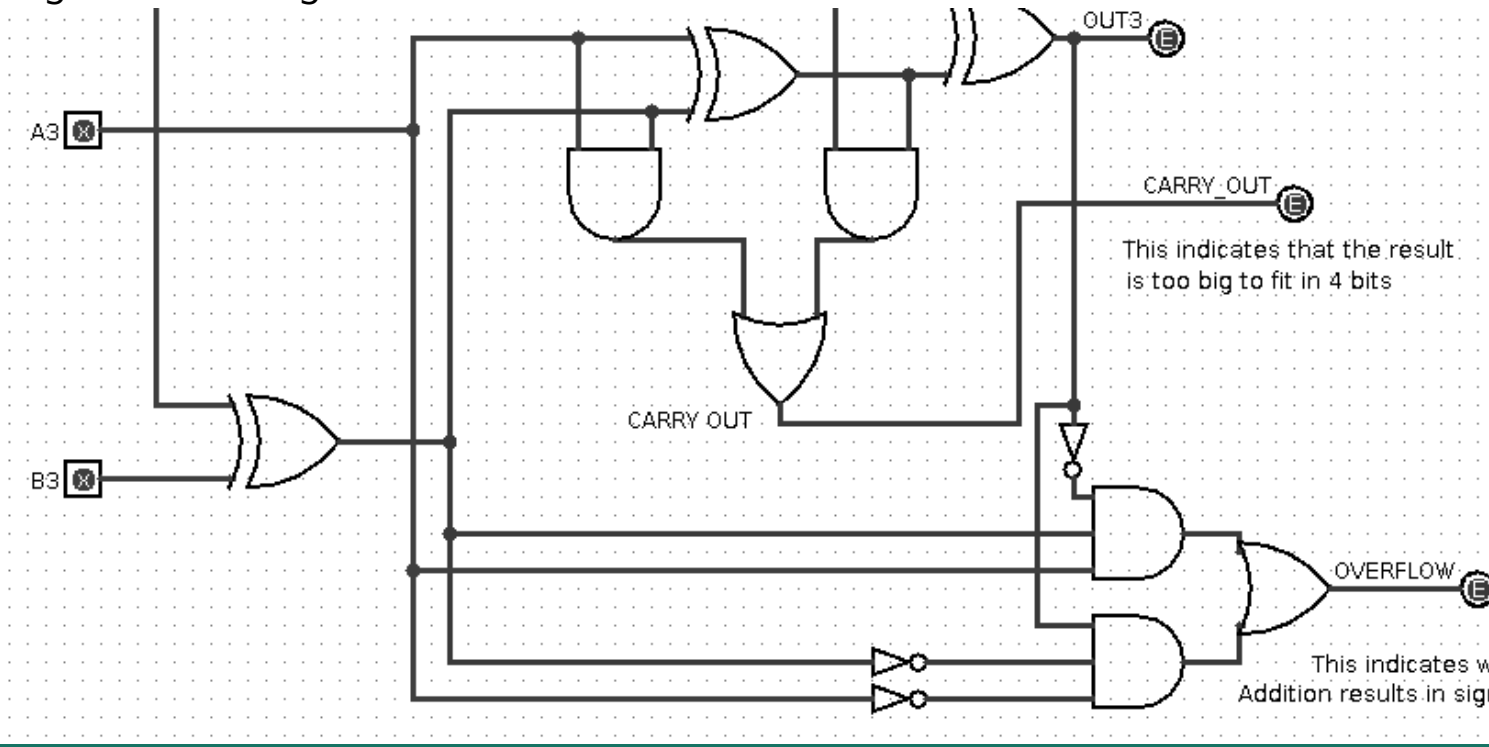
Building blocks

- The OR/AND block has the following pins:
  - A-input
  - B-input
  - Control input (1 – AND, 0 – OR)
  - Result output

- The Adder/subtractor has the following pins:
  - A-input
  - B-input
  - Control input (1 – subtract, 0 - add)
  - Result output
  - CARRY output

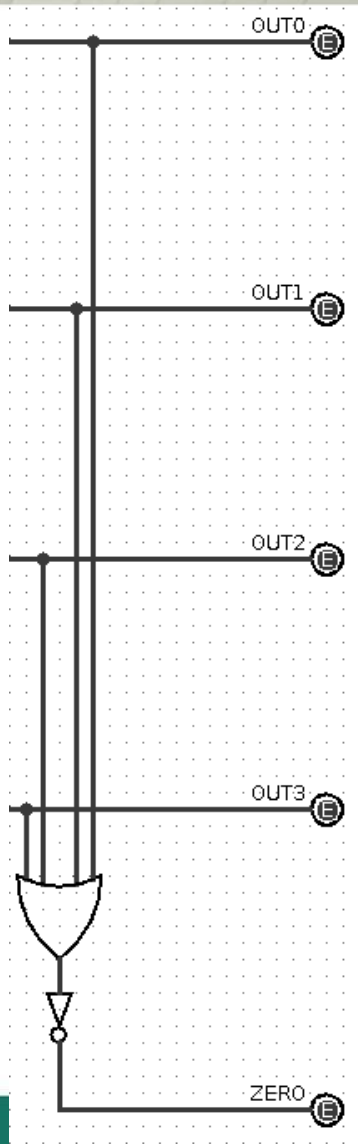
A few useful signals

- For ADDER, we can introduce OVERFLOW – when the operation caused the module to "turn again" like a mechanical counter. It happens when:
  - $!Q_3$ AND A_3 AND B_3
  - Q_3 AND $!A_3$ AND $!B_3$



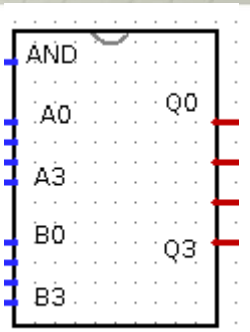
A few useful signals

- ZERO – which is lit when there is a zero as the result.
- If it was used in a computer, it can e.g. indicate an end of a loop which subtracts a pre-defined variable every iteration.
- ...or it can redundantly indicate that the addition of non-zero number resulted in a specific type of overflow.

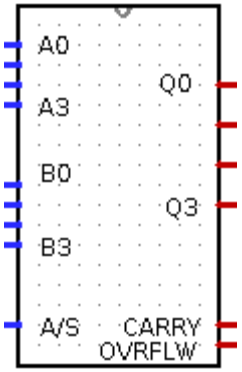


Let’s pack it up!

- This is our "AND/OR" chip. It has a complete AND/OR as shown in previous slides.



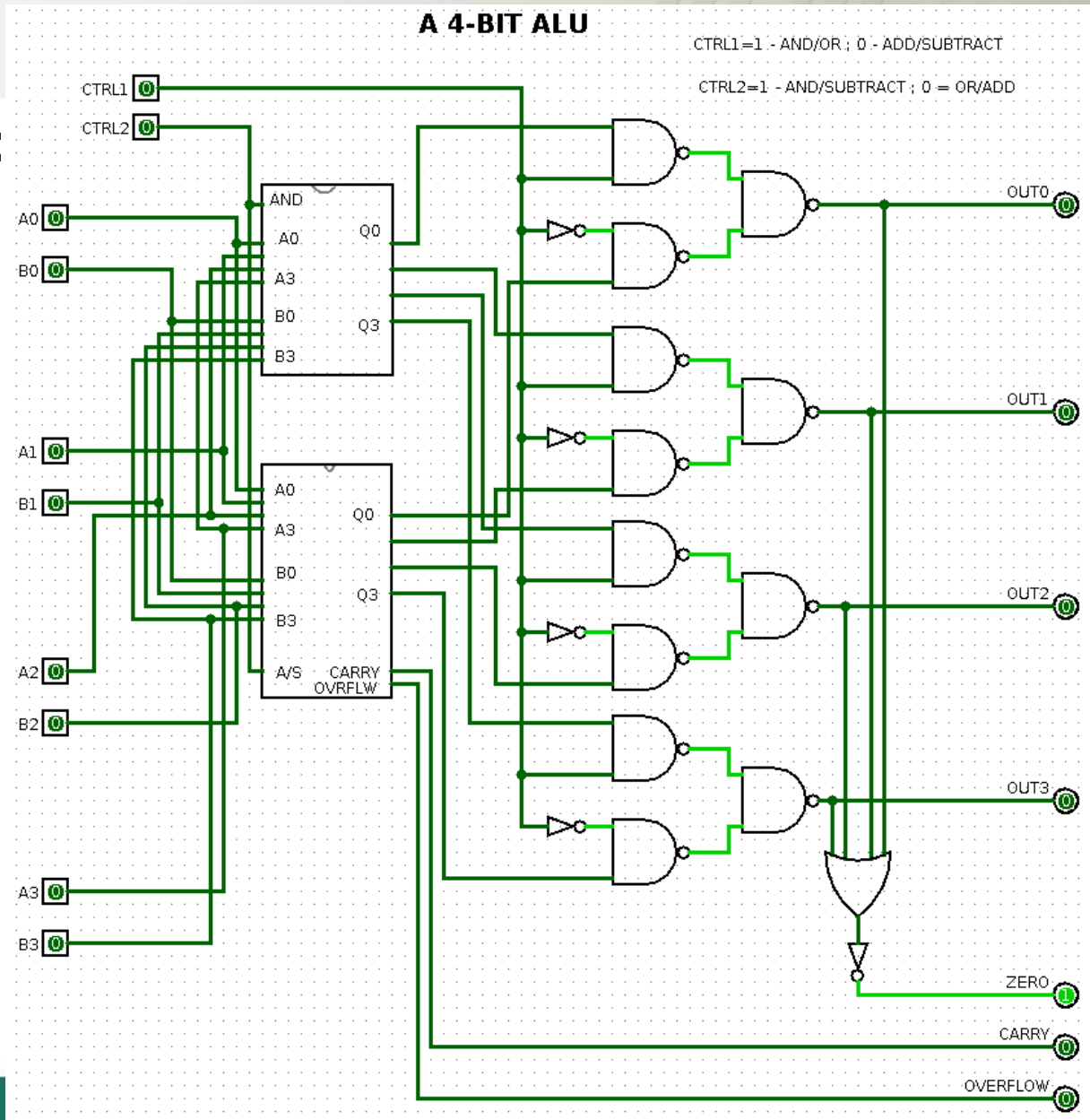
- This is our Adder-Subtractor chip. It has a complete Adder/Subtractor with Carry an Overflow outputs:



Let's pack it up!

- So this is a final ALU:
- The inputs are just parallel.
- The outputs are multiplexed.

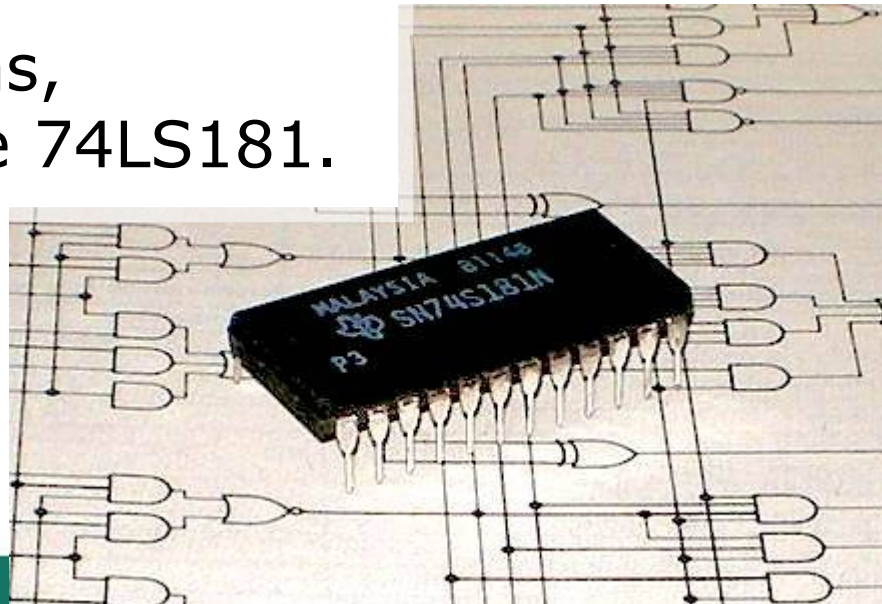
CTRL1	CTRL2	Function
0	0	ADD
0	1	SUB
1	0	OR
1	1	AND



How are ALUs made?

- In CPU, an ALU is the integral part of the chip. It is connected to the other parts in the silicon die.
- There is also an **FPU** – Floating-point unit – for calculating floating-point numbers more efficient.
  - In older systems (pre Intel 486) FPU was external to the CPU, as a separate chip.

- In pre-microprocessor designs, there is a 4-bit ALU chip: The 74LS181.



Real world CPUs

- Real world CPUs have a specific order for every instruction they perform.

- **Fetch** – The instruction is transferred from the memory to an intermediate storage in the CPU.
- **Decode** – the instructiond ecoder converts the instruction's bits to activating various components in the CPU. Connect proper registers to the ALU or address specific location in memory.
- **Execute –** run the command on register values.
- **Store** – Write the result back to the proper register. Or swap scratch registers content with actual ones.

Pipelining CPUs

- Notice that these 4 actions are performed by different parts of the CPU. That would take at least 4 clock cycles, not including the RAM access.
- Wouldn't be faster to run them simultaneously?
- If some of them are ran simultaneously, it's a **pipelining** CPU.
- Modern CPUs are pipelining and superscalar.

Basic five-stage pipeline

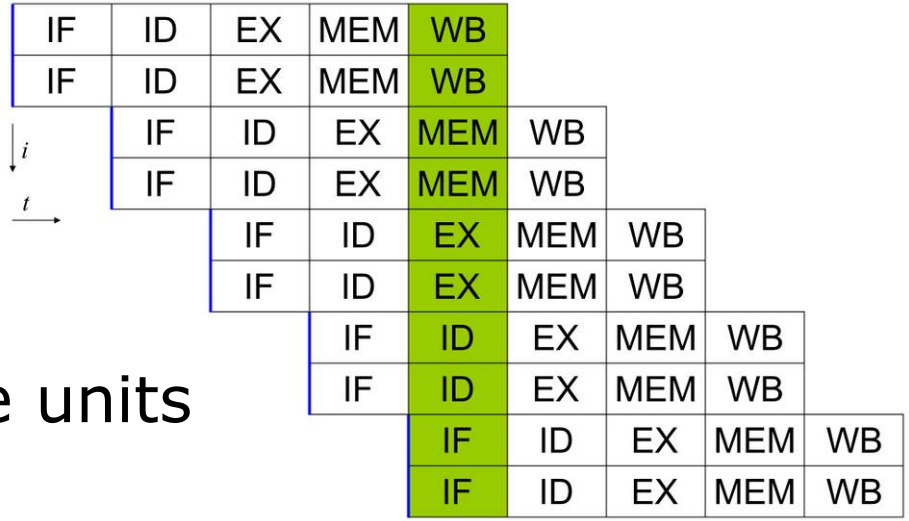
Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

Superscalar

- ...and wouldn't it be faster to execute a few instructions in parallel **if** some units are free?
- This means that resources for decoding will double the work, but it will make execution much faster.
- These CPUs are called **superscalar**.
  - In the beginning, it was operating like: If ALU's AND-OR part has nothing to do, let's multiplex it to another register pair!
- Modern CPUs have multiple units for this work.



RISC vs CISC

- Intel's x86/64-bit line are (externally) **CISC** – Complex Instruction Set Computing CPUs.
  - Instructions take some time, usually a few clock cycles.
  - On the other hand, these instructions handle much more complex data and have complex poerations, like vector operators, operating system's functions or context-aware operations.
  - Modern CISC processors decode the instruction to a set of RISC small, fast instructions and run it in the superscalar core.
  - The decoding is done using a **microcode** which can be even upgradeable.

RISC processors

- These are ARMs, first Apple Mac's CPUs, some IBM ones.
- Instructions are simplified so the amount of work per instruction is reduced.
  - There can be more instructions than in some CISC CPUs.
- Instructions are mostly divided to memory-related and ALU-related.
  - Notice memory access is much simpler now!
- At least one instruction per cycle.

- On the other hand, there is a lack of instructions corresponding to what high-level programming languages can do.

Branching

- If we have a command “Jump to …” - how to implement it?

- By the ALU!
- Jumping means changing position of an **instruction pointer**.
- And this is just a register we can alter with ALU operations.
- It only requires to “stand still” for the operation’s length (not to execute half of the jump’s address) and store in one cycle (to avoid superscalar part going somewhere else).

Other features of modern CPUs

- Speculative execution:
  - If the code is executed “in advance”, how about branching?
  - Will it branch or not?
  - Modern CPU’s unit responsible for predicting branches allows to execute more code “in advance”.
  - If branch was not predicted, but it happened, the intermediate results are discarded and execution resumes normal way.

- On the other hand, this is risky. Intermediate results may be cached, and then may be recovered revealing e.g. encryption keys.

Intel's Hyper threading, AMD's SMP

- Does it need for the superscalar pipeline to work on two instructions from the same program/context?
- What if we use one instruction from one program, and the second from another?
- We'll get a slight performance increase and it will look like one core is 2 cores.

- On the other hand we can analyze the missing time of one task to guess what another one is doing…
  - …to, for example, discover how it decrypts data!

Additional concepts in CPUs

- Why do we need clock?
  - To make sure CPU's components will execute operations in their time and synchronously.
- With or without clock?
  - Without clock – every part of CPU tells the next one that it's ready.
  - Then the CPU "rolls" into the next state of instruction execution.
  - After storage, the CPU has to go back to the initial state to fetch the next instruction.
  - Quite easy to design with pen and paper, yet difficult to upgrade – any further upgrade decreases efficiency.

Source: https://mera400.pl

Next lecture

- Netwide Assembler
- Simple programs
- Loops
- Memory operations.

- (if we'll have time) FPU and SIMD operations.

AGH

Thank you for attention