Computer Architecture, Networks & Operating Systems

Peyman Afshani and Jean Pichon-Pharabod

2024

Department of Computer Science – Aarhus University

About myself

Assistant Prof. Jean Pichon-Pharabod (jean.pichon@cs.au.dk)

Research interests:

- **Bridging practical engineering and mathematical rigour**
- Relaxed memory concurrency
- Hardware-software interface

Based on material by Diego F. Aranha

$\rightsquigarrow$ Security perspective

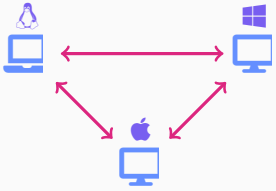
Course syllabus

Main goal

Give to the participants insight into basic architecture of computers, assembly language programming, the fundamentals of operating systems, and computer networks.

Structured in three parts:

1. Computer Architecture - 7 weeks (Jean)
2. Operating Systems - 4 weeks (Jean & Peyman)
3. Networks - 4 weeks (Peyman)



Course syllabus

Learning objectives

- Explain the architecture of computers as **multiple levels**
- Describe **each** level at some level of detail
- Describe the fundamental **functions** of modern operating systems
- Describe the fundamental **components** of computer networks
- Write **simple programs** that use assembly language, system calls, multithreading, sockets, and employ client-server interactions

Logistics: check Brightspace for list of TAs, schedule for exercise classes.

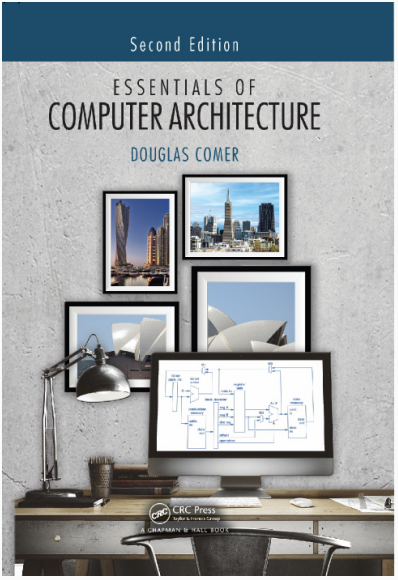
Exercises: can be found on the course webpage on Brightspace.

Groups: You should join a group on Brightspace.

Exam: Multiple choice. **A minimum grade is required to take the exam!**

Course syllabus

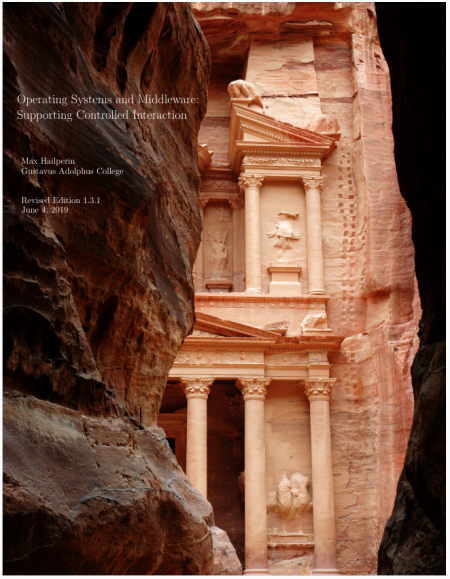
1. Douglas Comer
   Essentials of Computer Architecture (ECA) 2nd edition, Chapman and Hall/CRC, 2017



Course syllabus

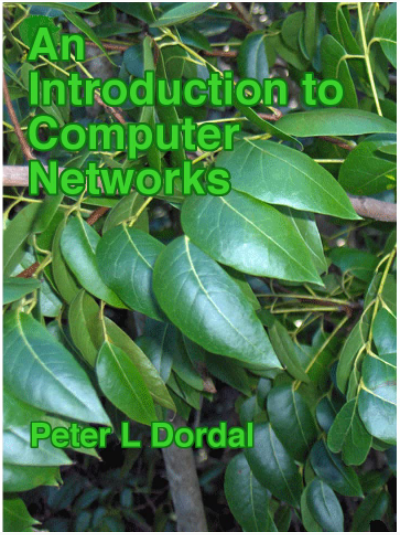
1. Douglas Comer
   Essentials of Computer Architecture (ECA) 2nd edition, Chapman and Hall/CRC, 2017

2. Operating systems and middleware, Edition 1.3 by Max Hailperin:
   `https://gustavus.edu/mcs/max/os-book/`



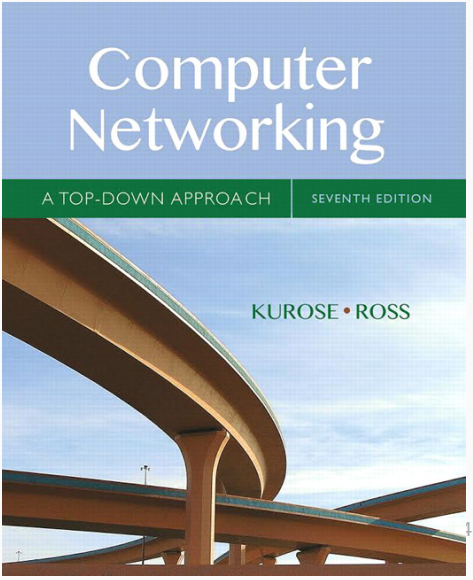
Course syllabus

1. Douglas Comer
   Essentials of Computer Architecture (ECA) 2nd edition, Chapman and Hall/CRC, 2017
2. Operating systems and middleware, Edition 1.3 by Max Hailperin:
   `https://gustavus.edu/mcs/max/os-book/`
3. Introduction to Computer Networks by Peter L Dordal. `https://intronetworks.cs.luc.edu/`



Course syllabus

1. Douglas Comer
   Essentials of Computer Architecture (ECA) 2nd edition, Chapman and Hall/CRC, 2017
2. Operating systems and middleware, Edition 1.3 by Max Hailperin:
   `https://gustavus.edu/mcs/max/os-book/`
3. Introduction to Computer Networks by Peter L Dordal. `https://intronetworks.cs.luc.edu/`
4. further: *Computer Networking, A Top-Down Approach*, 7th Edition, Kurose & Ross



Introduction to computer architecture

This week

1. Motivation and Context
2. Overall structure of computers, zooming in
3. Data representation
4. Later weeks: zoom out

What is a computer?

Q What is a computer?
What essential qualities does it need to have?

`menti.com` 1805 5354

What is a computer?

Not a definition

A *computer* is an automatic[1], programmable[2] information[3] treatment[4] system.
A program in *machine code* (in a *machine language*) instructs a computer how to treat the information.

[1] how automatic?
[2] how programmable?
[3] what counts as information?
[4] how expressive a treatment?

Exercise

Exercise For the rest of your life, every time you see something, ask yourself:

Is it a computer?

Does it contain a computer?

How much of a computer?

What is a computer?

What is a computer?

Narrower definitions

A *computer* is (often) a machine that can perform **calculations/operations** (*instructions*) in a methodological and step-by-step fashion.
A *program* in machine code is (often) a sequence of instructions.

There are multiple types of instructions:

- Arithmetic/logic operations
  - usual
  - specialised
- Flow control
- Memory access
- ...

Motivation

In this course, we will learn about **how** a computer works.

Q: But why is it **useful**? Why should you take this course **seriously**?

`menti.com` 1805 5354

Motivation

In this course, we will learn about **how** a computer works.

Why you should take this course **seriously**:

- It's on the exam! ... and employers will expect you to know about it!
- It's interesting!
- As a programmer, you will have to design abstractions $\rightsquigarrow$ learn by examples
- Write better and more optimised code making better usage of **resources**.
- Write more secure code by understanding how abstractions **leak**
  (security analysis is essentially about violating abstractions).
- Low-level system programming and domain-specific architectures
  Example: Google's TPU (Tensor Processing Unit) for Machine Learning
  - Up to 80 times CPU performance per watt!
  - Superhuman performance in some tasks
    (AlphaStar: `https://www.nature.com/articles/s41586-019-1724-z`)

Challenges

The *bad* news:

- Hardware is **complex**: many details, not always intuitive, legacy, ...
- The subject is **extensive**: many architectures, many components in each architecture, ...
- Terminology is sometimes confusing...

The *good* news:

- Abstractions make it possible to understand architecture without knowing **all** details
- Knowing the **essentials** will allow you to zoom in as needed
  - Characteristics of major components
  - Role in overall system
  - Consequences for software

Why are computers amazing?

Why are computers amazing?

Bootstrapping

If a computer is programmable enough,
then we can program it to program itself!

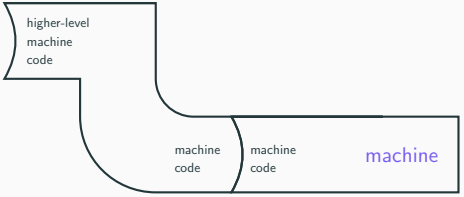
- use a nicer language $\leftsquigarrow$ interpreters, compilers
  - use natural language $\leftsquigarrow$ LLM-based chatbots
- pretend to be another computer $\rightsquigarrow$ virtual machines, emulation, ...
- pretend to be several computers $\rightsquigarrow$ operating systems
- ...

Used throughout modern computer systems,
including **within** computers! (e.g. x86)

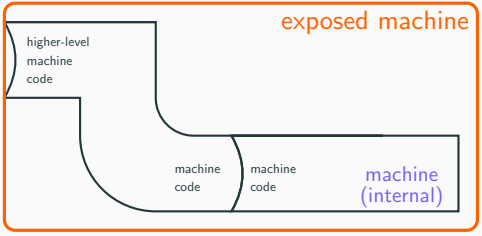
See Computability and Logic, Programming Languages, Compilers.

Example

Example

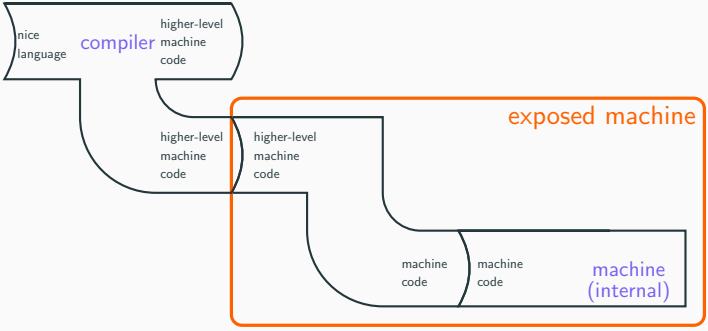

higher-level
machine
code
machine
code
machine
code
machine

Example



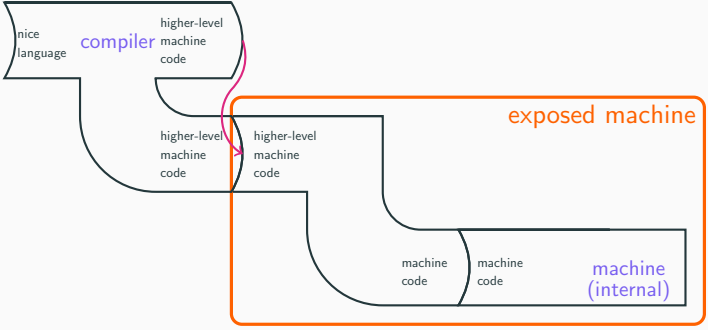
x86 machines are like that

Example



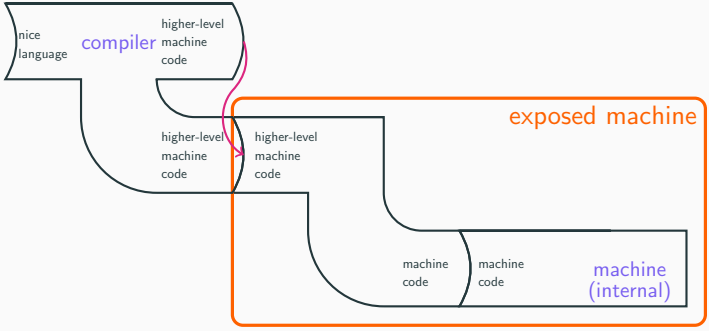
x86 machines are like that

Example



x86 machines are like that

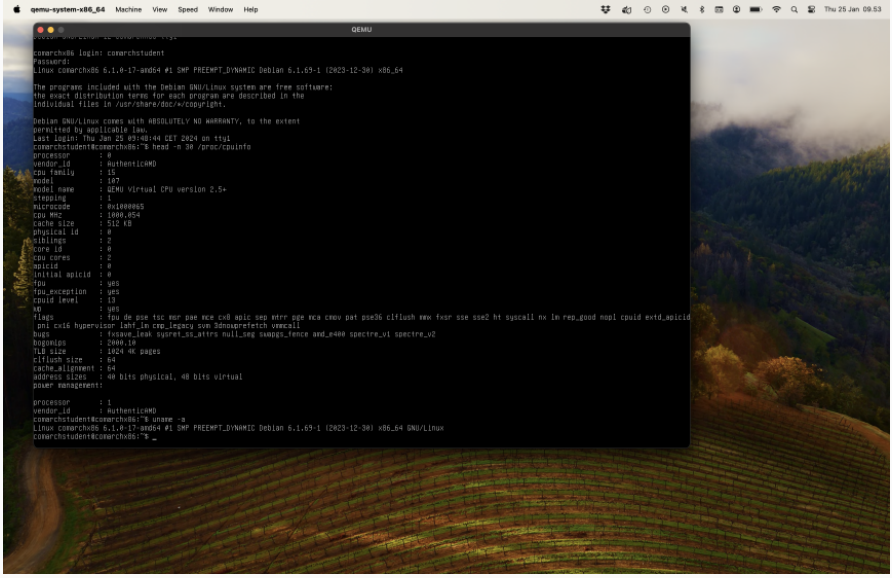
Example



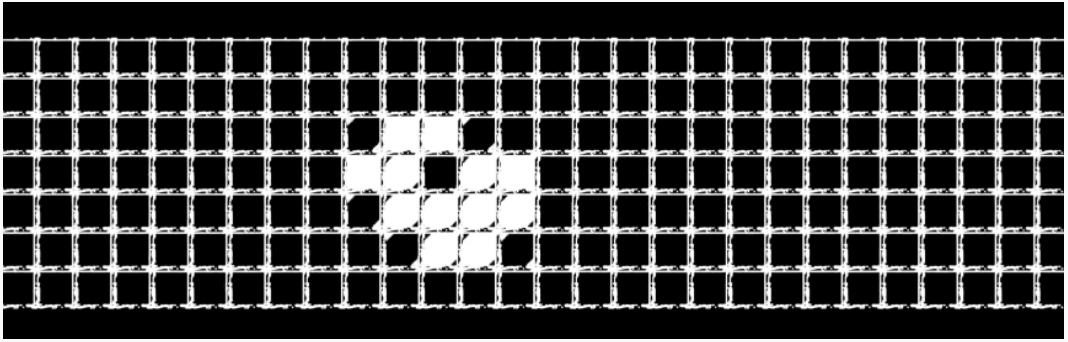
x86 machines are like that

J Diagrams: see “Pearl: Diagrams for Composing Compilers”, Wickerson & Brunet
`https://johnwickerson.wordpress.com/2020/05/21/diagrams-for-composing-compilers/`

Virtual machines: QEmu (and Docker!) — used in this course

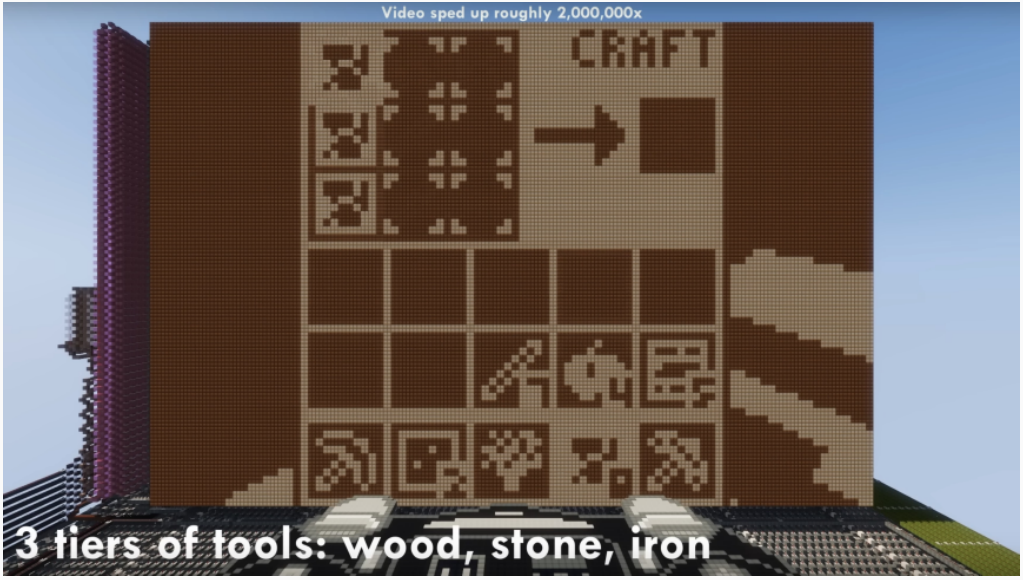


Game of Life in Game of Life



See also: `https://codegolf.stackexchange.com/questions/11880/build-a-working-game-of-tetris-in-conways-game-of-life`

Minecraft in Minecraft



Chatbots based on Large Language Models

Q: Can a modern chatbot pass this course?
A: Maybe?

Q: Should you use chatbots to learn ComArch?
A: Only carefully!

At the time of writing, even the best chatbots rely (largely) on syntax, not on semantics
$\rightsquigarrow$ no difference between "A and B", "A and not B", "A implies B", "A is implied by B"

Very brief history

Abstraction

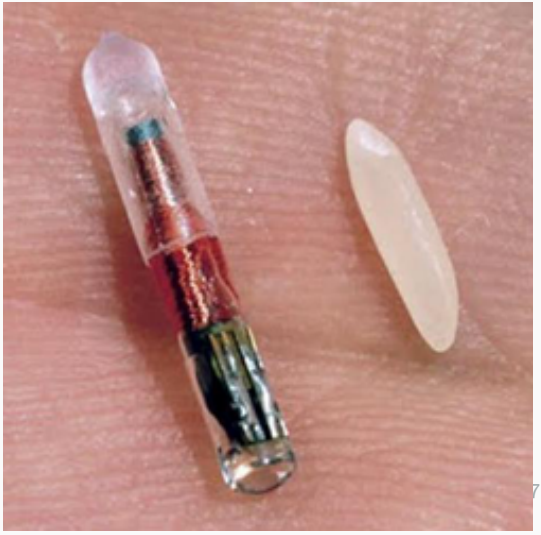
Computers started with just two levels: digital (hardware) and ISA (software)

With time, more levels were added to increase **abstraction**:

1. Invention of **microprogramming** (around 1950)
2. Invention of **operating system** (around 1960)
3. Migration of functionality to **microcode**
4. Reduction/elimination of microprogramming (RISC)
5. Capability to run assemblers
6. Capability to run compilers & interpreters for high-level languages

What are computers used for?

- **Disposable** computers: cheap sensors
- **Microcontroller**: e.g. simple IoT devices
- **Mobile** and gaming computers
- **Personal** computers: notebooks and desktops
- **Server**: high-performance clusters for scientific computing, cloud platforms
- **Mainframe**: throughput-heavy computation



What are computers used for?

- **Disposable** computers: cheap sensors
- **Microcontroller**: e.g. simple IoT devices
- **Mobile** and gaming computers
- **Personal** computers: notebooks and desktops
- **Server**: high-performance clusters for scientific computing, cloud platforms
- **Mainframe**: throughput-heavy computation

Intel P8051

What are computers used for?

- **Disposable** computers: cheap sensors
- **Microcontroller**: e.g. simple IoT devices
- **Mobile** and gaming computers
- **Personal** computers: notebooks and desktops
- **Server**: high-performance clusters for scientific computing, cloud platforms
- **Mainframe**: throughput-heavy computation

Playstation 5

What are computers used for?

- **Disposable** computers: cheap sensors
- **Microcontroller**: e.g. simple IoT devices
- **Mobile** and gaming computers
- **Personal** computers: notebooks and desktops
- **Server**: high-performance clusters for scientific computing, cloud platforms
- **Mainframe**: throughput-heavy computation

ThinkPad

What are computers used for?

- **Disposable** computers: cheap sensors
- **Microcontroller**: e.g. simple IoT devices
- **Mobile** and gaming computers
- **Personal** computers: notebooks and desktops
- **Server**: high-performance clusters for scientific computing, cloud platforms
- **Mainframe**: throughput-heavy computation

Intel Server System w/ 2x40 Xeon cores

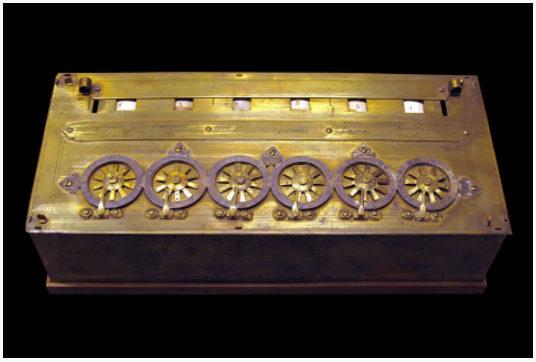
What are computers used for?

- **Disposable** computers: cheap sensors
- **Microcontroller**: e.g. simple IoT devices
- **Mobile** and gaming computers
- **Personal** computers: notebooks and desktops
- **Server**: high-performance clusters for scientific computing, cloud platforms
- **Mainframe**: throughput-heavy computation

IBM Mainframe

What are computers made of?

- **0th generation** (1642-1945): Mechanical machines
- **1st generation** (1945-1955): Vacuum tubes
  - electronic building blocks
  - initially, programmed through switches; later von Neumann machine (EDSAC)
- **2nd generation** (1955-1965): Transistors
  - invented in 1948, Bell Labs, Nobel prize in 1956
- **3rd generation** (1965-1980): integrated circuits
- **4th generation** (1980-?): VLSI
- **5th generation** (IoT?): Low-power, invisible, integrated

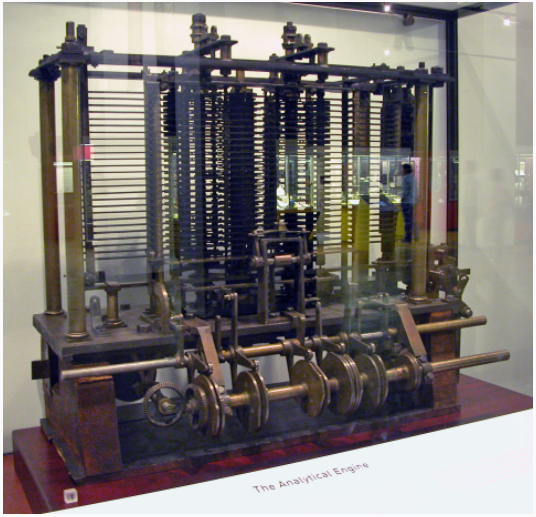


Pascal's calculator

https://forrestheller.com/Apollo-11-Computer-vs-USB-C-chargers.html

What are computers made of?

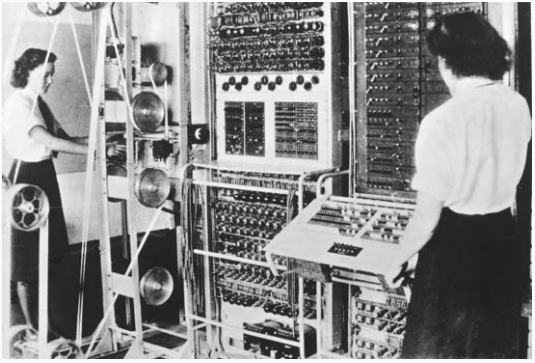
- **0th generation** (1642-1945): Mechanical machines
- **1st generation** (1945-1955): Vacuum tubes
  - electronic building blocks
  - initially, programmed through switches; later von Neumann machine (EDSAC)
- **2nd generation** (1955-1965): Transistors
  - invented in 1948, Bell Labs, Nobel prize in 1956
- **3rd generation** (1965-1980): integrated circuits
- **4th generation** (1980-?): VLSI
- **5th generation** (IoT?): Low-power, invisible, integrated



Babbage's Analytical Engine

What are computers made of?

- **0th generation** (1642-1945): Mechanical machines
- **1st generation** (1945-1955): Vacuum tubes
  - electronic building blocks
  - initially, programmed through switches; later von Neumann machine (EDSAC)
- **2nd generation** (1955-1965): Transistors
  - invented in 1948, Bell Labs, Nobel prize in 1956
- **3rd generation** (1965-1980): integrated circuits
- **4th generation** (1980-?): VLSI
- **5th generation** (IoT?): Low-power, invisible, integrated

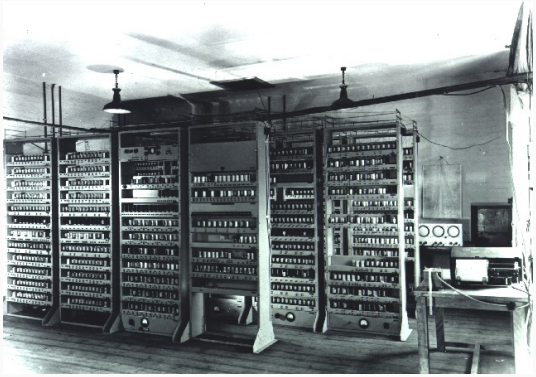


COLOSSUS

https://forrestheller.com/Apollo-11-Computer-vs-USB-C-chargers.html

What are computers made of?

- **0th generation** (1642-1945): Mechanical machines
- **1st generation** (1945-1955): Vacuum tubes
  - electronic building blocks
  - initially, programmed through switches; later von Neumann machine (EDSAC)
- **2nd generation** (1955-1965): Transistors
  - invented in 1948, Bell Labs, Nobel prize in 1956
- **3rd generation** (1965-1980): integrated circuits
- **4th generation** (1980-?): VLSI
- **5th generation** (IoT?): Low-power, invisible, integrated

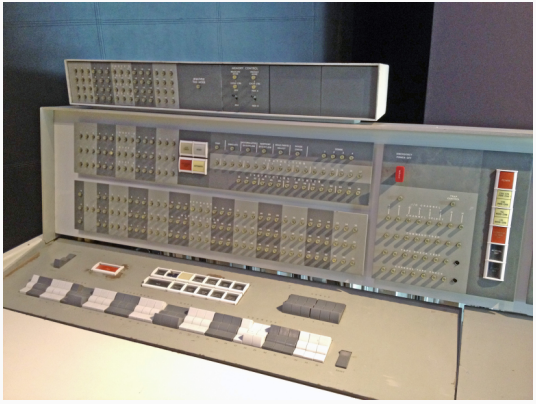


EDSAC

https://forrestheller.com/Apollo-11-Computer-vs-USB-C-chargers.html

What are computers made of?

- **0th generation** (1642-1945): Mechanical machines
- **1st generation** (1945-1955): Vacuum tubes
  - electronic building blocks
  - initially, programmed through switches; later von Neumann machine (EDSAC)
- **2nd generation** (1955-1965): Transistors
  - invented in 1948, Bell Labs, Nobel prize in 1956
- **3rd generation** (1965-1980): integrated circuits
- **4th generation** (1980-?): VLSI
- **5th generation** (IoT?): Low-power, invisible, integrated



IBM 7094

https://forrestheller.com/Apollo-11-Computer-vs-USB-C-chargers.html

What are computers made of?

- **0th generation** (1642-1945): Mechanical machines
- **1st generation** (1945-1955): Vacuum tubes
  - electronic building blocks
  - initially, programmed through switches; later von Neumann machine (EDSAC)
- **2nd generation** (1955-1965): Transistors
  - invented in 1948, Bell Labs, Nobel prize in 1956
- **3rd generation** (1965-1980): integrated circuits
- **4th generation** (1980-?): VLSI
- **5th generation** (IoT?): Low-power, invisible, integrated

IBM System/360 (first series)

https://forrestheller.com/Apollo-11-Computer-vs-USB-C-chargers.html

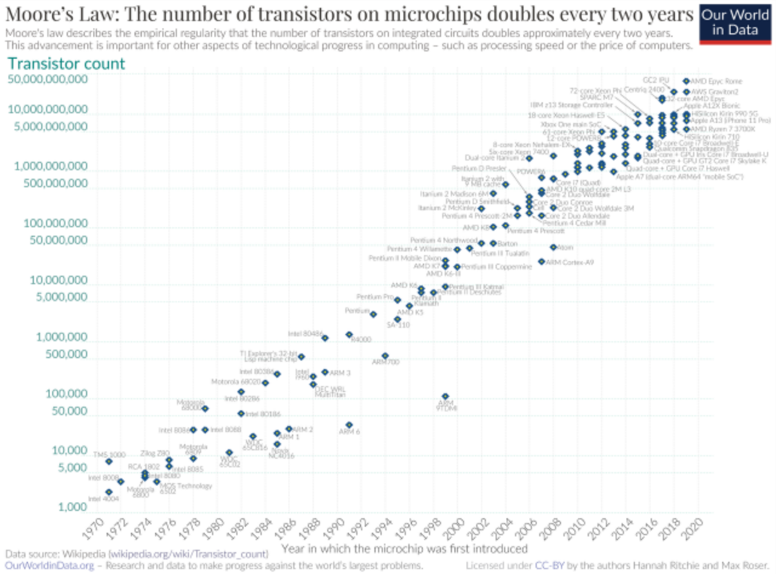
What are computers made of?

- **0th generation** (1642-1945): Mechanical machines
- **1st generation** (1945-1955): Vacuum tubes
  - electronic building blocks
  - initially, programmed through switches; later von Neumann machine (EDSAC)
- **2nd generation** (1955-1965): Transistors
  - invented in 1948, Bell Labs, Nobel prize in 1956
- **3rd generation** (1965-1980): integrated circuits
- **4th generation** (1980-?): VLSI
- **5th generation** (IoT?): Low-power, invisible, integrated

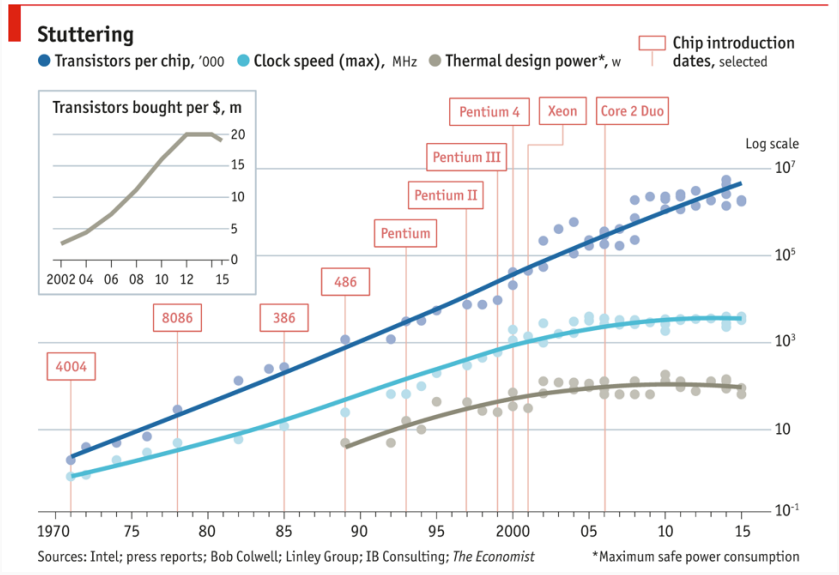
Intel 80386 25MHz

https://forrestheller.com/Apollo-11-Computer-vs-USB-C-chargers.html

Moore’s Law


Moore’s Law: The number of transistors on microchips doubles every two years
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 for other aspects of technological progress in computing – such as processing speed or the price of computers.
Transistor count
50,000,000,000
10,000,000,000
5,000,000,000
1,000,000,000
500,000,000
100,000,000
50,000,000
10,000,000
5,000,000
1,000,000
500,000
100,000
50,000
10,000
5,000
1,000
1970
1972
1974
1976
1978
1980
1982
1984
1986
1988
1990
1992
1994
1996
1998
2000
2002
2004
2006
2008
2010
2012
2014
2016
2018
2020
Year in which the microchip was first introduced
Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.
Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

Moore's Law: the end?



Terminology

Terminology: architecture vs. architecture

Computer Architecture:
the study of how computers are designed
$\rightsquigarrow$ peeking inside

Hardware Architecture:
specification of a type of computer (including ISA)
e.g. x86, x86-64, aarch64, RISC-V, ...
$\rightsquigarrow$ just the surface

Terminology

Computer systems are all about **abstraction**.

We use different terms to refer to different levels of abstraction:

1. **Hardware Architecture**:
   - Blueprint for the overall organisation of a system
   - Specifies functionality of major components and how they interconnect
   - Abstracts away details

Terminology

Computer systems are all about **abstraction**.

We use different terms to refer to different levels of abstraction:

1. **Hardware Architecture**:
   - Blueprint for the overall organisation of a system
   - Specifies functionality of major components and how they interconnect
   - Abstracts away details
2. **Micro-architectural Design**:
   - Translates architecture into components
   - Fills in details about how components are grouped and how power is distributed
   - Note: *Many* designs can satisfy a given architecture

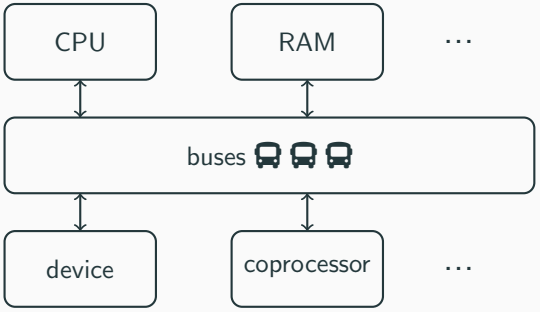
Terminology

Computer systems are all about **abstraction**.
We use different terms to refer to different levels of abstraction:

1. **Hardware Architecture**:
   - Blueprint for the overall organisation of a system
   - Specifies functionality of major components and how they interconnect
   - Abstracts away details
2. **Micro-architectural Design**:
   - Translates architecture into components
   - Fills in details about how components are grouped and how power is distributed
   - Note: *Many* designs can satisfy a given architecture
3. **Implementation**:
   - All details necessary to build a system:
     chassis, layout of components, power supply, wiring, part numbers

Organisation of a simple computer

Organisation of a simple computer: from a mile up



- Can be several of each type
- Buses are not always *n*-to-*n*

Processors

What is a processor?

A digital device that can perform a computation involving multiple steps.

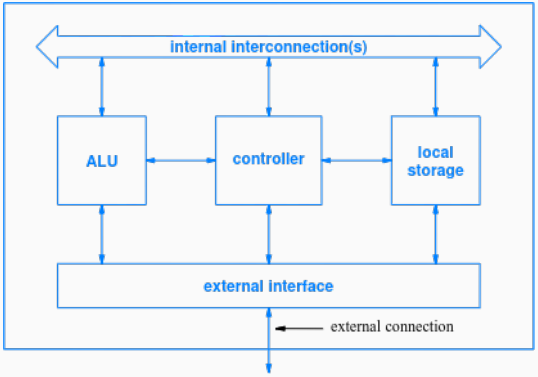
Can be as simple as a piece of dedicated hardware (circuit) implementing a *sine* function, or complex enough to be *programmable* and run any program.

Gradient:

1. *Coprocessor*: performs a single task at **high speed** (floating point, cryptography)
2. *Microcontroller*: programmed to **control** a physical system (car engine)
3. *Embedded Systems*: more complex, found in a wireless router or smart phone
4. *General-purpose*: Desktops and notebooks that can run **any program**

Organisation of a simple processor

- Controller
- Arithmetic Logic Unit (ALU)
- Local storage (registers)
- Internal interconnections (data path)
- External interface (I/O registers, bus)



Fetch-decode-execute cycle

A processor has a special register, the *program counter* (aka *instruction pointer*).

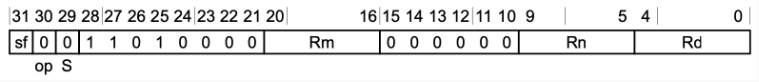
A fetch-decode-execute:

1. **Fetch**: get the instruction from memory at the address in the program counter.
2. **Decode**: determine the type of instruction just fetched, and its operands.
3. **Execute**: perform the instruction on its operands:
   - If the instruction operands refer to memory, perform *address calculation*.
   - Read from memory as needed.
   - Perform the computation.
     This might involve reading from and writing to registers, or communicating with coprocessors, etc.
   - Write to memory as needed.
   - Set the new program counter:
     - For usual instructions, increment the program counter to point to the **next** instruction.
     - For branches, set the program counter to its new destination.

The processor executes fetch-decode-execute in a loop (or in multiple loops at the same time).

Detailed example

- Read register pc. Let's say it contains 1000. Fetch from address 1000.
  Let's say we get value 1 0 0 11010000 00010 000000 00001 00011.
- Decode it: according to



  it's an add with carry of x1 and x2 into x3: `adc x3,x1,x2`
- Execute:
  Read from register x1. Let's say it contains 7.
  Read from register x2. Let's say it contains 6.
  Read from special register PSTATE.C. Let's say it contains 0.
  Ask the ALU to add 7, 6, and 0. The result is 13.
  Write 13 to register x3.
  Increment pc.

https://developer.arm.com/documentation/ddi0602/2023-12/Base-Instructions/ADC--Add-with-Carry-

Examples

Load register (immediate) `ldr x1,[x2,#2]`
$\rightsquigarrow$ load from memory at 2 past the address in x2, write the result to x1.

Store register (immediate) `str x1,[x2,#4]`
$\rightsquigarrow$ write to memory at 4 past the address stored in x2 the value in x1.

Branch if not equal `b.ne 1004`
$\rightsquigarrow$ if PSTATE.Z is 0, increment pc by 1004
(otherwise, increment as usual)

Floating-point division (scalar) `fdiv d3,d1,d2`
$\rightsquigarrow$ divide d1 by d2, write the result in d3

Exercise Read the ASL code for these instructions:
`https://developer.arm.com/documentation/ddi0602/2023-12/Base-Instructions`

Processor speed

Question: How **fast** does the fetch-execute cycle operate?

1. Clock rate (in Hz)
   - Rate at which the clock generator of a processor can generate pulses, used to **synchronize** its components
   - Provides a measure of the **underlying** hardware speed
2. Instruction rate (in instructions per second)
   - Measures the number of instructions a processor can execute per unit of time
   - Some instructions may take more clock cycles than other instructions
     Example: multiplication may take longer than addition

How does a processor start/stop?

Start When powering up the machine, the program counter has a designated address.

- *Small embedded systems*: the program resides in Read Only Memory (ROM)
- *Larger systems*: hardware runs a *bootloader* that starts from the designated address, which typically reads the operating system into memory.

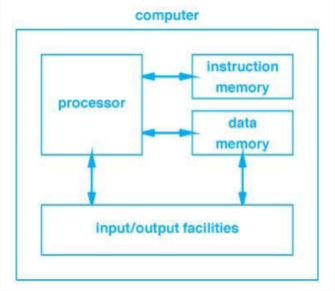
Stop The processor runs fetch-decode-execute indefinitely
$\rightsquigarrow$ "spin" waiting for something to change

There may be a way to cut power...
...or to sleep until something happens: timer, network, ...
There may also be a way to spin in a low-power mode.

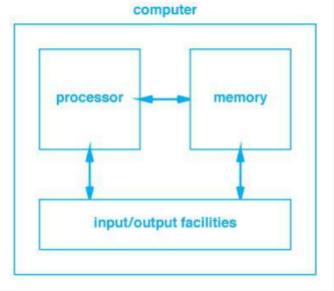
Relation between code and data

Where should code live? Two basic approaches:

Harvard Architecture:
two memories for data/program



Von Neumann Architecture:
stored program (**winner!***)



*Although modern computers still have separate caches for code and for data, and embedded systems have their code in ROM.

Complex computations

Complex computations

Instructions perform elementary computations on a fixed, small number of registers...

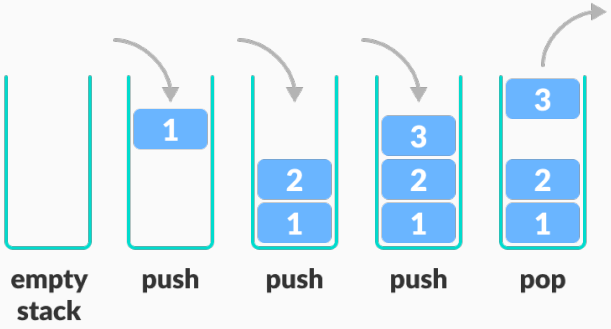
...but “real” programs work over more variables than there are registers

"We can solve any problem by introducing an extra level of indirection." — Butler Lampson

Stacks

Stack

A stack a data structure with two operations: **push** (insert), and **pop** (remove) an item, which implement a **LIFO** (*last-in-first-out*) access pattern.



Stacks can easily be implemented by just keeping track of the top

Implementing recursion using stacks

```
def fib(x):
    if x in [0,1]:
       return 1
    else:
        temp1 = fib(x - 1)
        temp2 = fib(x - 2)
        return temp1 + temp2
```

Implementing recursion using stacks

```
def fib(x):
    if x in [0,1]:
       return 1
    else:
        temp1 = fib(x - 1)
        temp2 = fib(x - 2)
        return temp1 + temp2
```

```
def fib2(x):
  tot = 0
  stack = [x]
  while stack:
     a = stack.pop()
     if a in [0,1]:
        tot += 1
     else:
         stack.append(a - 1)
         stack.append(a - 2)
  return tot
```

Detour

When describing mathematical formulae, it is common to use **infix** notation.
Infix means that the operator is in between the two operands:

$$(1 + 2) * (3 + 5)$$

Detour

When describing mathematical formulae, it is common to use **infix** notation.
Infix means that the operator is in between the two operands:

$$(1 + 2) * (3 + 5)$$

One of its downsides is that certain expressions require bracketing.

An alternate notation exists that does not have this issue: *Reverse Polish Notation* (RPN).

Detour: stack machines

The expression $(1 + 2) * (3 + 5)$ can be represented in RPN as

$$1\ 2\ +\ 3\ 5\ +\ *$$

Detour: stack machines

The expression $(1 + 2) * (3 + 5)$ can be represented in RPN as

$$1\ 2\ +\ 3\ 5\ +\ *$$

This generalises beyond arithmetic to programming: Idea Each symbol defines an **instruction**:

- A **number** just means push this number onto the stack.
- Symbol $+$ means pop two elements from the stack, add them, push the result back.
- Symbol $*$ does the same as $+$ but for multiplication.

The list above defines an **instruction set**. Implementing a **stack machine** for evaluating arithmetic expressions is now simple.

Implementing a stack machine — in software

```
def is_numeric(x):
  return ord('0') < ord(x) and ord(x) < ord('9')
def compute(exp):
    stack = []
    for op in exp:
        if is_numeric(op):
            stack.append(int(op))
        elif op == '+':
            stack.append(stack.pop() + stack.pop())
        elif op == '-':
            stack.append(stack.pop() - stack.pop())
        elif op == '*':
            stack.append(stack.pop() * stack.pop())
        # add more...
    return stack.pop()
print(compute(['1','2','+']))
```

Stack machines

We can do this in hardware!

Many *early* computers were **stack machines**.

A few programming languages implement **virtual stack machines**:
Java's JVM, WebAssembly, CPython, Ethereum's EVM, ...

`https://webassembly.github.io/spec/core/exec/instructions.html`

Many modern computers provide some support for a call stack: **stack pointer register**

`https://developer.arm.com/documentation/den0024/a/ARMv8-Registers/AArch64-special-registers`

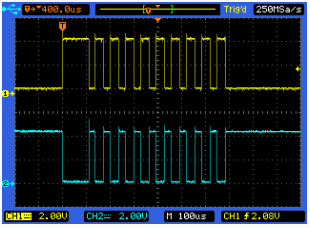
Data representation

Bits

A modern computer is built on a **two-valued** logic system. It can be progressively abstracted:

- Physics: electrons on wires
- Electronics: Close to zero or close to 3.3 volts
- Binary: High and low $\rightsquigarrow$ *binary digit*, abbreviated **bit**

  Can be interpreted as:
  - Numeric: 1 and 0
  - Logical: true and false



Important

A bit has no intrinsic meaning — all meaning is imposed by the interpretation!

Encoding

We can use lists of bits to represent more complex data:

- Numbers of different types: integers, floating point, ...
- Characters in different alphabets
- Addresses in memory
- Graphs, tax code, ...

Important

Bits have no intrinsic meaning — all meaning is imposed by the encoding!

Bytes

Current convention defines a **byte** to be *eight* (8) bits.
On most computers, the byte is the **smallest** addressable unit of storage

Question: How many values can be represented with 3 bits?

Bytes

Current convention defines a **byte** to be *eight* (8) bits.
On most computers, the byte is the **smallest** addressable unit of storage

Question: How many values can be represented with 3 bits?

1. **Positional** notation (list): 8 values:
   000, 001, 010, 011, 100, 101, 110, 111

   More generally, 2^k states for k bits.
2. **Non-positional** notation (set): 4 values:
   - 0: 000
   - 1: 001, 010, 100
   - 2: 110, 101, 011
   - 3: 111

   Q: How many for k bits?

Representing integers

Positional notation

We are used to decimal positional notation:
Decimal:

- $a_n a_{n-1} \dots a_2 a_1 a_0 = a_n \times 10^n + a_{n-1} \times 10^{n-1} + \dots + a_2 \times 10^2 + a_1 \times 10^1 + a_0 \times 10^0$
- $5234 = 5 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

Positional notation

We are used to decimal positional notation:
Decimal:

- $a_n a_{n-1} \dots a_2 a_1 a_0 = a_n \times 10^n + a_{n-1} \times 10^{n-1} + \dots + a_2 \times 10^2 + a_1 \times 10^1 + a_0 \times 10^0$
- $5234 = 5 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

We can do the same for binary:
Binary:

- $(a_n a_{n-1} \dots a_2 a_1 a_0)_2 = a_n \times 2^n + a_{n-1} \times 2^{n-1} + \dots + a_2 \times 2^2 + a_1 \times 2^1 + a_0 \times 2^0$
- $(1101)_2 = 0\text{b}1101 = 1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 13$

Important

From now on, we will allow ourselves to conflate a list of bits with a natural number.

Positional notation

This works for any base:
Arbitrary base $b \geq 2$:

- $(a_n a_{n-1} \dots a_2 a_1 a_0)_b = a_n \times b^n + a_{n-1} \times b^{n-1} + \dots + a_2 \times b^2 + a_1 \times b^1 + a_0 \times b^0$

Hexadecimal:
Digits: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A (10), B (11), C (12), D (13), E (14), F (15)

- $(\text{A3C9})_{16} = 0\text{xA3C9} = A \times 16^3 + 3 \times 16^2 + 12 \times 16 + 9 \times 16^0 = 41929$

Easy to convert to binary:

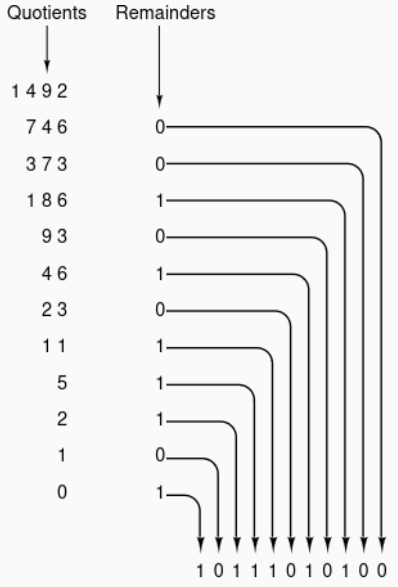
- $(\text{A3C9})_{16} = (1010\ 0011\ 1100\ 1001)_2$

Note: other common bases are 8 (octal), 64 (`base64`).
Python makes it easy to play with common bases, using notation 0bn, 0on, 0xn.

Basis conversion

Conversion from basis b_0 to b_1 can be done by performing consecutive **division** in b_0 and taking the remainders in reverse order in b_1.

Exercise Do it the other way around.



Fixed precisions

Computers often represent numbers in **fixed precision** because memory elements have fixed size (processor **word**), often 8, 16, 32, 64, or 128 bits.

- **Unsigned**
  With k bits, we can encode **unsigned integers** (**natural numbers**) 0, 1, 2, ..., $2^k - 1$:
  1. use positional notation: $117 = (1110101)_2$ — 7 bits
  2. pad left with 0s: $(01110101)_2$ in 8 bits

Fixed precisions

Computers often represent numbers in **fixed precision** because memory elements have fixed size (processor **word**), often 8, 16, 32, 64, or 128 bits.

- **Unsigned**
  With k bits, we can encode **unsigned integers** (**natural numbers**) 0, 1, 2, ..., $2^k - 1$:
  1. use positional notation: $117 = (1110101)_2$ — 7 bits
  2. pad left with 0s: $(01110101)_2$ in 8 bits

- **Signed**
  We can also encode **signed** (i.e. potentially negative) integers.
  Idea 1: use a bit for the sign.
  Idea 2: three main methods.

Encoding signed numbers in fixed precision

1. Historical: **Signed magnitude**:
   Use 1 bit for sign s, and $k-1$ bits for an unsigned integer factor $n \in [0, 2^{k-1}-1]$:

encodes $(-1)^s \times n$

   Counts $0, 1, ..., 2^{k-1}-1, -0, -1, ..., -(2^{k-1}-1)$

   - Encodes from $-(2^{k-1}-1)$ to $(2^{k-1}-1)$
   - Familiar to humans
   - Quirk: has a negative 0.

Encoding signed numbers in fixed precision

2. Historical: **1's complement**:
   Use 1 bit for sign, and $k-1$ bits for:
   - For positives: an unsigned integer factor

encodes n

   - For negatives: the bitwise negation of the encoding of the unsigned integer factor

encodes $-(\overline{n})$

   Counts $0, 1, ..., 2^{k-1}-1, -(2^{k-1}-1), -(2^{k-1}-1)+1, \ldots, -0$

   - Encodes from $-(2^{k-1}-1)$ to $(2^{k-1}-1)$
   - Computing $-n$ is easy: flip all the bits
   - Quirk: two representations for 0.

Encoding signed numbers in fixed precision

3. **2's complement**: 1 bit for 'sign factor', $k-1$ bits for an offset:

encodes $s \times (-2^{k-1}) + n$

Counts $0, 1, ..., 2^{k-1} - 1, -2^{k-1}, -2^{k-1} + 1, \ldots, -1$

- Encodes from -2^{k-1} to $2^{k-1} - 1$
  $\rightsquigarrow$ Quirk: has one more negative value than positive values
- ...but a unique representation of 0
- Computing $-n$ is relatively easy:
  invert all bits of positive value and add 1
- Almost exclusively used now

Representing 4-bit numbers

Binary String	Unsigned (positional) Interpretation	Sign Magnitude Interpretation	One's Complement Interpretation	Two's Complement Interpretation
0000	0	0	0	0
0001	1	1	1	1
0010	2	2	2	2
0011	3	3	3	3
0100	4	4	4	4
0101	5	5	5	5
0110	6	6	6	6
0111	7	7	7	7
1000	8	−0	−7	−8
1001	9	−1	−6	−7
1010	10	−2	−5	−6
1011	11	−3	−4	−5
1100	12	−4	−3	−4
1101	13	−5	−2	−3
1110	14	−6	−1	−2
1111	15	−7	−0	−1

Encoding signed numbers

Modern computers use **two's complement** due to several advantages:

- Only one encoding of zero
- Negative is given by **inverting bits + 1**
- Hence, $(A - B) = A + (\neg B + 1)$.

Wow: We can use addition circuit for **subtraction**!

Fixed precision arithmetic

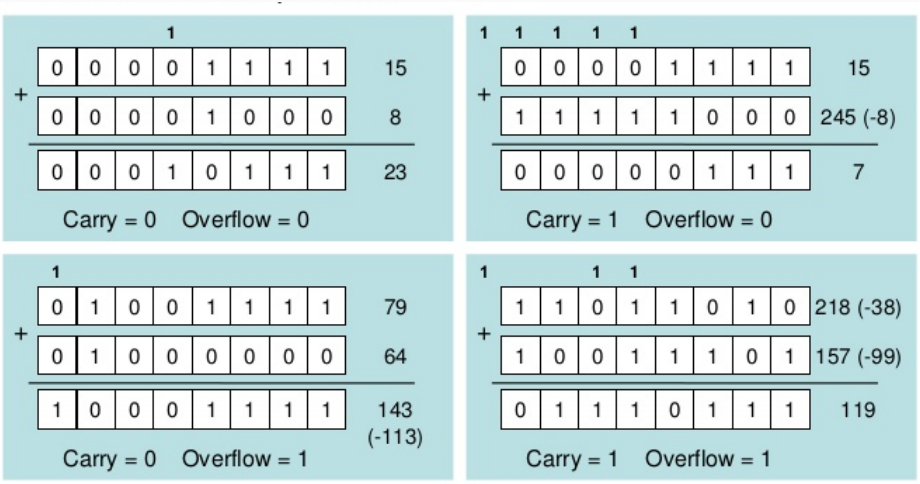
Fixed precision arithmetic can produce *overflow* or *underflow*, or have a *carry* when results cannot be represented in k bits.

Example: $1010_2 + 0111_2$ "should" be $1\ 0001_2$ — but that does not fit in 4 bits
$\rightsquigarrow$ it's "only" 0001_2 in 4 bits fixed precision...

Bits thrown out are made available in **processor flags** (special registers)
e.g. PSTATE.C and PSTATE.V in aarch64.

`https://developer.arm.com/documentation/100933/0100/Processor-state-in-exception-handling`

Binary arithmetic

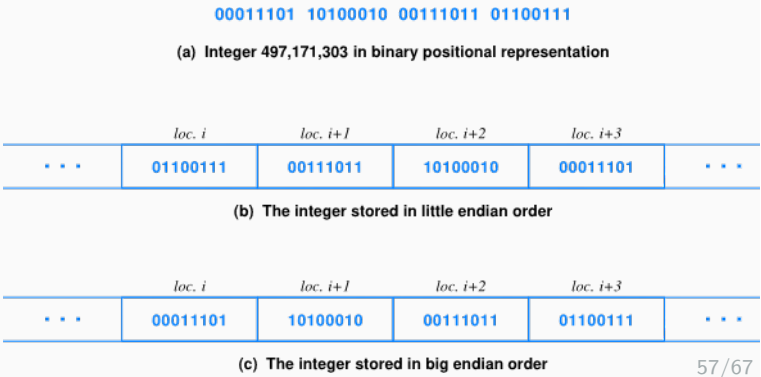


In short: *carry* happens when result does not **fit**, *overflow* when sign is **wrong**.

Representing numbers – byte order

When numbers take > 1 byte, we need to **choose** the order to store and transmit them. While *bit order* is usually handled by the hardware and hidden to the programmer, *byte order* is typically **visible** to the programmers:

1. *Little Endian*
   least significant byte in lowest memory location
2. *Big Endian*
   most significant byte in lowest memory location



Converting integers: sign extension

Computers represent integers of different sizes
e.g. X and W registers on aarch64

`https://developer.arm.com/documentation/den0024/a/ARMv8-Registers`

$\rightsquigarrow$ sometimes we need to convert numeric values to **larger** representations.

Idea: Perform **sign extension** of the high-order bit.
Example:

- 124 represented in two's complement in 8 bits is 0111 1100b.
- The 16-bit version is 0000 0000 0111 1100b.
- -4 represented in two's complement in 8 bits is 1111 1100b.
- The 16-bit version is 1111 1111 1111 1100b.

This works with two's complement signed integers

Sign extension and unsigned integers

For unsigned integers, this does not work:

Example:

```
#include <stdio.h>
#include <stdint.h>
int main(void) {
  int8_t c = 0xf0; // 240
  int16_t x, y;
  x = c; // -16
  y = c & 0xff; // 240
  printf("x = %x %d, y = %x %d\n", x, x, y, y);
}
```

Question: What happens when shifting to the right?

Representing characters

Character sets

Two programs that communicate in text with each other need to pick a **character set** that associates a unique bit pattern to each symbol.
A typical character set contains:

- uppercase and lowercase letters, digits
- punctuation marks.
- unprintable characters for **control**

Various character sets have been used in commercial computers:

- EBCDIC: popular in the 60's and IBM mainframes
- ASCII: previous de facto standard
- **Unicode**: extends ASCII to accommodate many languages, 1-4 bytes long

The full ASCII character set

00	nul	01	soh	02	stx	03	etx	04	eot	05	enq	06	ack	07	bel
08	bs	09	ht	0A	lf	0B	vt	0C	np	0D	cr	0E	so	0F	si
10	dle	11	dc1	12	dc2	13	dc3	14	dc4	15	nak	16	syn	17	etb
18	can	19	em	1A	sub	1B	esc	1C	fs	1D	gs	1e	rs	1F	us
20	sp	21	!	22	"	23	#	24	$	25	%	26	&	27	'
28	(	29	)	2A	*	2B	+	2C	,	2D	–	2E	.	2F	/
30	0	31	1	32	2	33	3	34	4	35	5	36	6	37	7
38	8	39	9	3A	:	3B	;	3C	<	3D	=	3E	>	3F	?
40	@	41	A	42	B	43	C	44	D	45	E	46	F	47	G
48	H	49	I	4A	J	4B	K	4C	L	4D	M	4E	N	4F	O
50	P	51	Q	52	R	53	S	54	T	55	U	56	V	57	W
58	X	59	Y	5A	Z	5B	[	5C	\	5D	]	5E	^	5F	_
60	‘	61	a	62	b	63	c	64	d	65	e	66	f	67	g
68	h	69	i	6A	j	6B	k	6C	l	6D	m	6E	n	6F	o
70	p	71	q	72	r	73	s	74	t	75	u	76	v	77	w
78	x	79	y	7A	z	7B	{	7C	|	7D	}	7E	~	7F	del

Representing floats

Why floats?

Fixed precision integers work quite well:
Observation: In practice, integers largely stay within bounds
(and corner cases can be handled easily).

Rationals can be represented by a pair of integers...
...but then the observation above does not hold anymore.
(Intuition: the multiplications in $\frac{a}{b} + \frac{c}{d} = \frac{a \times c + b \times d}{c \times d}$ compound)
$\rightsquigarrow$ does not work in fixed width

So what to do?

A very poor compromise: floats.

Floating-point representation

Fundamental idea: follow standard scientific representation that specifies a few significant digits and an order of magnitude

Example: Avogadro's number

$$6.022 \times 10^{23}$$

Tweak: (most) hardware uses base 2 instead of base 10.

Floats are *weird*: sometimes, $x + 1.0\text{f} = x$

Sometimes, $x + (y + z) \neq (x + y) + z$

$\rightsquigarrow$ Warning! Floats are a datatype of **last resort**!

What Every Computer Scientist Should Know About Floating-Point Arithmetic

`https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html`

Floating-point representation (IEEE 754 standard)

To implement this idea, allocate fixed-size bit strings for:

1. **Sign** s (1 bit)
   - Because of the sign, $+0 \neq -0$ (!).
2. **Mantissa** m:
   - Not the 'actual' mantissa, but only its decimals
   - $\rightsquigarrow$ zero is a special case (*all bits zero*)
3. **Exponent** e:
   - To be able to represent numbers of magnitude < 1, e is then 'offset' by e_0

$(-1)^s \times (1.m) \times 2^{e-e_0}$

Plus some tricks to represent $-\infty$, $+\infty$, NaN

Floating-point representation (IEEE 754 standard)

$$1 =$$

$$2 =$$

$$-9 =$$

$$0.5 =$$

$$-2.25 =$$

Floating-point representation (IEEE 754 standard)

$$1 = (-1)^0 \times (1.0)_2 \times 2^{e_0 - e_0}$$

$$2 =$$

$$-9 =$$

$$0.5 =$$

$$-2.25 =$$

Floating-point representation (IEEE 754 standard)

$$
\begin{aligned}
1 &= (-1)^0 \times (1.0)_2 \times 2^{e_0 - e_0} \\
2 &= (-1)^0 \times (1.0)_2 \times 2^{(e_0+1) - e_0} \\
-9 &= \\
0.5 &= \\
-2.25 &=
\end{aligned}
$$

Floating-point representation (IEEE 754 standard)

$$
\begin{aligned}
1 &= (-1)^0 \times (1.0)_2 \times 2^{e_0 - e_0} \\
2 &= (-1)^0 \times (1.0)_2 \times 2^{(e_0+1) - e_0} \\
-9 &= (-1)^1 \times (1.001)_2 \times 2^{(e_0+3) - e_0} \\
0.5 &= \\
-2.25 &=
\end{aligned}
$$

Floating-point representation (IEEE 754 standard)

$$
\begin{aligned}
1 &= (-1)^0 \times (1.0)_2 \times 2^{e_0 - e_0} \\
2 &= (-1)^0 \times (1.0)_2 \times 2^{(e_0+1) - e_0} \\
-9 &= (-1)^1 \times (1.001)_2 \times 2^{(e_0+3) - e_0} \\
0.5 &= (-1)^0 \times (1.0)_2 \times 2^{(e_0-1) - e_0} \\
-2.25 &=
\end{aligned}
$$

Floating-point representation (IEEE 754 standard)

$$
\begin{aligned}
1 &= (-1)^0 \times (1.0)_2 \times 2^{e_0 - e_0} \\
2 &= (-1)^0 \times (1.0)_2 \times 2^{(e_0+1) - e_0} \\
-9 &= (-1)^1 \times (1.001)_2 \times 2^{(e_0+3) - e_0} \\
0.5 &= (-1)^0 \times (1.0)_2 \times 2^{(e_0-1) - e_0} \\
-2.25 &= (-1)^1 \times (1.01)_2 \times 2^{(e_0+1) - e_0}
\end{aligned}
$$

Floating-point representation (IEEE 754 standard)

Specifies **single**- and **double**-precision formats:

1. Single-precision: magnitudes between 2^{-126} to 2^{127}

Floating-point representation (IEEE 754 standard)

Specifies **single**- and **double**-precision formats:

1. Single-precision: magnitudes between 2^{-126} to 2^{127}

2. Double-precision: magnitudes between 2^{-1022} to 2^{1023}

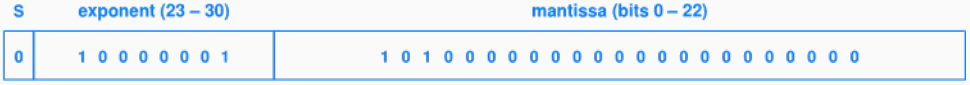
Floating-point representation (IEEE 754 standard)

Specifies **single**- and **double**-precision formats:

1. Single-precision: magnitudes between 2^{-126} to 2^{127}

2. Double-precision: magnitudes between 2^{-1022} to 2^{1023}

Example: Represent $6.5 = 110.1 = 1.101 \times 2^2$ (exponent is 129 by adding 127).



Summary

- A computer is a programmable machine
- Modern computers are made of digital circuits
- Major components: CPU, memory, bus
- Fetch-decode-execute cycle
- Fundamental value in digital logic is a bit, grouped into lists to represent integers, characters, floating-point values, etc.
- Integers can be represented in multiple forms...
- ...two's complement w/ little endian being the most common

Next: Basics of digital **circuits** and logic!