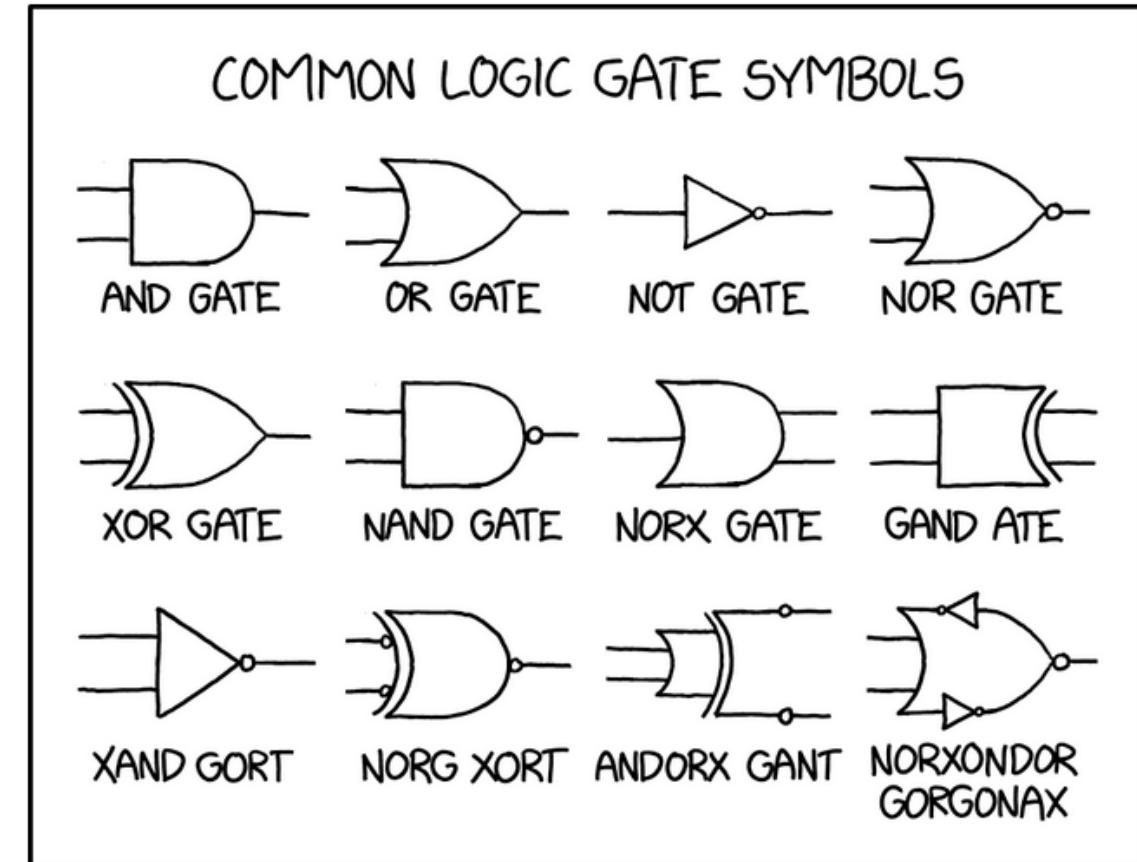
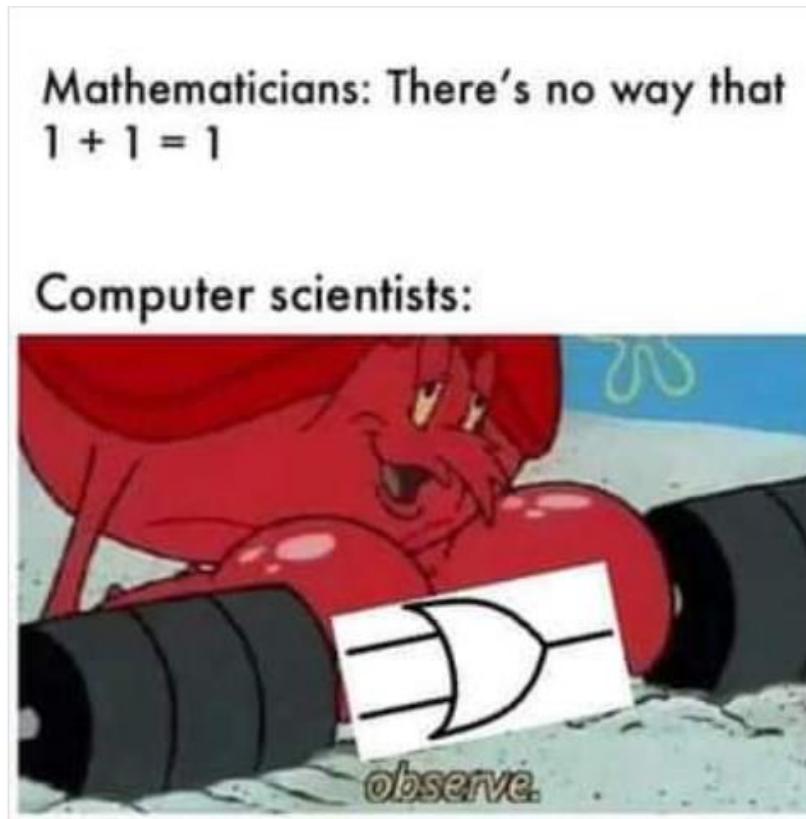
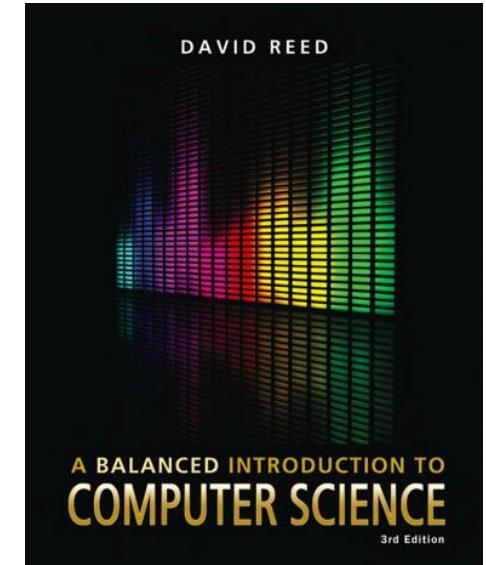


# 1. Computer science, Boolean algebra and logic gates



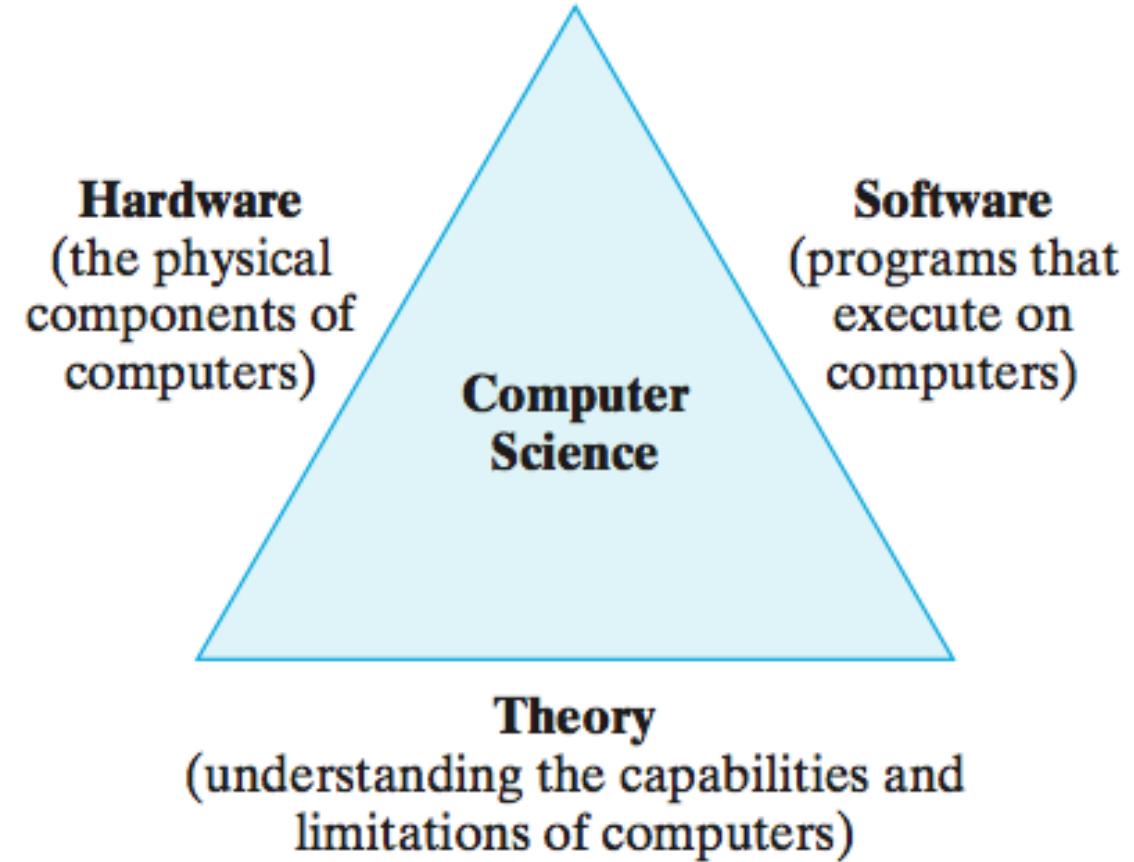
# What is “computer science”?

- Computer science is very much different from all natural sciences
  - Some authors use the term “artificial science” for distinction
- It is the study of computation, which deals with problem solving:
  - Design and analysis of algorithms
  - Formalization of algorithms to programs
  - Development of computational devices for executing programs
  - Theoretical study of the power and limitations of computing
- Every other science aims to understand how the world works, and then model and predict its behavior
- Computer scientists, on the other hand, work in a universe that they have created by themselves; they try to create abstractions of real-world problems that can be understood and solved by computers



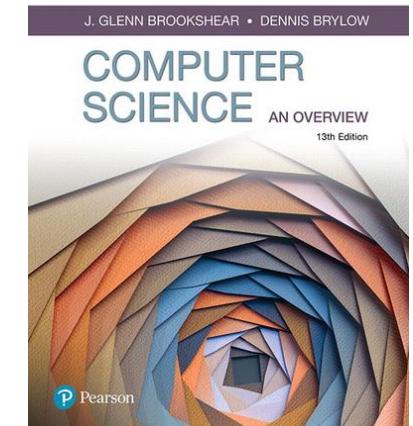
# Computer science themes

- Computer science has three themes that define the discipline:
  - Hardware
  - Software
  - Theory
- Analogy: an athlete performing his/her sport
  - Hardware = sport-specific equipment (which enables the task)
  - Software = the athlete (who is performing the task)
  - Theory = trainers (who constantly think of ways to improve the result of the task)



# The “Seven Big Ideas of Computer Science” (Brookshear-Brylow)

- Algorithms
  - Step-by-step directions on how to perform a task; the origin of Computer Science
- Abstraction
  - Different levels of abstraction allow us to handle complex processes via abstract tools
- Creativity
  - A human is creative, a machine is not – it's just performing the task it's been given
- Data
  - How computers store known data, approximate unknown data and handle data errors?
- Programming
  - Translating human intentions into executable computer algorithms
- Internet
  - Connecting computers and electronic devices; privacy and security issues
- Impact
  - Influence of CS on the society (societal, ethical and legal)



# Subfields of CS

- Computer science can be divided to several subfields
  - This division according to P. Denning already in 2000
  - Nowadays maybe even more?
- Many of these subfields are follow-up courses on their own
- This course tries to act as a primer for further studies, so these subfields are not considered very thoroughly

Subfields of Computer Science <sup>1</sup>	
<b>Algorithms and Data Structures</b>	The study of methods for solving problems, designing and analyzing algorithms, and effectively using data structures in software systems.
<b>Architecture</b>	The design and implementation of computing technology, including the integration of effective hardware systems and the development of new manufacturing methods.
<b>Operating Systems and Networks</b>	The design and development of software and hardware systems for managing the components of a computer or network of communicating computers.
<b>Software Engineering</b>	The development and application of methodologies for designing, implementing, testing, and maintaining software systems.
<b>Artificial Intelligence and Robotics</b>	The study and development of software and hardware systems that solve complex problems through seemingly "intelligent" behavior.
<b>Bioinformatics</b>	The application of computing methodologies and information structures to biological research, such as the characterization and analysis of the human genome.
<b>Programming Languages</b>	The design and implementation of languages that allow programmers to express algorithms so that they are executable on computers.
<b>Databases and Information Retrieval</b>	The organization and efficient management of large collections of data, including the development of methods for searching and recognizing patterns in the data.
<b>Graphics</b>	The design of software and hardware systems for representing physical and conceptual objects visually, such as with images, video, or three-dimensional holograms.
<b>Human–Computer Interaction</b>	The design, implementation, and testing of interfaces that allow users to interact more effectively with computing technology.
<b>Computational Science</b>	Explorations in science and engineering that utilize high-performance computing, such as modeling complex systems or simulating experimental conditions.
<b>Organizational Informatics</b>	The development and study of management processes and information systems that support technology workers and organizations.

# Boolean algebra

- Boolean algebra is an algebra where variables can only have two possible values: TRUE or FALSE – or 1 and 0, respectively
- Also, the number of possible operations that we can perform is limited to three:
  - Conjunction (“AND”, symbol  $\wedge$ )
  - Disjunction (“OR”, symbol  $\vee$ )
  - Negation (“NOT”, symbol  $\neg$  or overbar)
- Boolean expressions can be formed just like propositions in propositional logic
  - Propositional logic should be familiar from FoIP course
  - Propositions can be simplified to Boolean expressions by replacing “arrows” (conditionals, biconditionals, etc.) with appropriate Booleans

# Boolean laws

- Boolean algebra follows its own laws (which quite well match the simplification rules of propositional logic, presented earlier in FoIP course)

- Commutativity:  $x \wedge y = y \wedge x$      $x \vee y = y \vee x$

- Distributivity:  $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$      $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$

- Complementation:  $x \wedge \bar{x} = 0$      $x \vee \bar{x} = 1$

- Identity:  $x \wedge 1 = x$      $x \vee 0 = x$

- Idempotence:  $x \wedge x = x$      $x \vee x = x$

- Associativity:  $(x \wedge y) \wedge z = x \wedge (y \wedge z)$      $(x \vee y) \vee z = x \vee (y \vee z)$

- Absorption:  $x \wedge (x \vee y) = x$      $x \vee (x \wedge y) = x$

- De Morgan laws:  $\overline{x \wedge y} = \bar{x} \vee \bar{y}$      $\overline{x \vee y} = \bar{x} \wedge \bar{y}$

- Annihilation and double negation:  $x \wedge 0 = 0$      $x \vee 1 = 1$      $\bar{\bar{x}} = x$

# Boolean functions

- Every Boolean expression represents some Boolean function
  - When variables (or arguments) are given values, the Boolean function returns a TRUE/FALSE value
- Boolean functions can be displayed as truth tables:
  - Rows correspond to all possible combinations of truth values for the variables
  - Columns for each variable (here  $p$ ,  $q$  and  $r$ ) and a column for the output value of the function (here denoted as  $f$ )
- Size of the truth table depends on the number of variables
  - If a Boolean function has  $k$  variables, the truth table has  $2^k$  rows
  - Therefore, the truth table contains all possible variable combinations
  - Tip: Write truth tables in binary order (easier to check)

$p$	$q$	$r$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

# Boolean functions

- While truth tables are a nicely “complete” way to represent Boolean functions, their exponentially growing size forces us to find other means to evaluate them
  - 10 variables  $\rightarrow 2^{10} = 1024$  rows
- For evaluation purposes, it is enough to find a Boolean expression that returns exactly the same output as the original Boolean function
- For each Boolean function, there exists an infinite amount of such expressions
  - How could we choose the simplest one?
  - Simplification using Boolean laws, Karnaugh maps, ...
- Finding out the simplest possible expression is, in general, a quite difficult task
  - Luckily, we don't usually need THE simplest one (if such even exists)
- Good baseline for simplification AND also for design of logical circuits (which perform Boolean operations in reality) is to present the expression in normal form

# Literals and elementary disjunctions

- Expressions which are either single propositional variables or their negations are called *literals*
- A Boolean expression is said to be an *elementary disjunction* if it has the form

$$A_1 \vee A_2 \vee \cdots \vee A_n$$

where

- Every  $A_i$  is a literal
- Every variable appears only in one  $A_i$
- Literals have (some) order

Example:

$p \vee q \vee \bar{r}$  is an elementary disjunction

$p \vee \bar{p} \vee s$  is not ( $p$  present in two literals!)

# Conjunctive normal form

- A Boolean expression is in *Conjunctive normal form* (CNF, also known as “Product of Sums” = POS), if it has the form

$$B_1 \wedge B_2 \wedge \cdots \wedge B_n$$

where

- Every  $B_i$  is an elementary disjunction
- None of the  $B_i$ 's are the same
- Elementary disjunctions have (some) order

Example:

$(p \vee q \vee \bar{r}) \wedge (\bar{q} \vee s)$  is a CNF

$(p \wedge (q \vee \bar{r})) \wedge (p \vee r)$  is not ( $B_1$  is not an elementary disjunction!)

# Disjunctive normal form

- A Boolean expression is in *Disjunctive normal form* (DNF; also known as “Sum of Products” = SOP), if it has the form

$$B_1 \vee B_2 \vee \cdots \vee B_n$$

where

- Every  $B_i$  is an *elementary conjunction* ( $A_1 \wedge A_2 \wedge \cdots \wedge A_n$ )
  - None of the  $B_i$ 's are the same
  - Elementary conjunctions have (some) order
- 
- Notice: the difference between CNF and DNF
    - CNF = operators between brackets are conjunctions
    - DNF = operators between brackets are disjunctions

Example of a DNF:

$$(p \wedge q) \vee (\bar{p} \wedge \bar{q} \wedge s)$$

$$(p \vee q) \wedge (q \vee s) \quad \text{CNF}$$

$$(p \wedge q) \vee (q \wedge s) \quad \text{DNF}$$

# From truth table to normal form

- Normal forms can be formed via truth tables
- From truth table to DNF:
  - Find all rows where the Boolean function is 1
  - Write these rows as elementary conjunctions
  - Formulate the disjunctive expression
- From truth table to CNF:
  - Find all rows where the Boolean function is 0
  - Change the truth value of all variables on these rows
  - Write these rows as elementary disjunctions
  - Formulate the conjunctive expression

**EXAMPLE:**

$p$	$q$	$r$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

# From truth table to normal form

- Normal forms can be formed via truth tables
- From truth table to DNF:
  - Find all rows where the Boolean function is 1
  - Write these rows as elementary conjunctions
  - Formulate the disjunctive expression
- From truth table to CNF:
  - Find all rows where the Boolean function is 0
  - Change the truth value of all variables on these rows
  - Write these rows as elementary disjunctions
  - Formulate the conjunctive expression

$$DNF: (\bar{p} \wedge \bar{q} \wedge r) \vee (p \wedge q \wedge \bar{r}) \vee (p \wedge q \wedge r)$$

**EXAMPLE:**

$p$	$q$	$r$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

# From truth table to normal form

- Normal forms can be formed via truth tables
- From truth table to DNF:
  - Find all rows where the Boolean function is 1
  - Write these rows as elementary conjunctions
  - Formulate the disjunctive expression
- From truth table to CNF:
  - Find all rows where the Boolean function is 0
  - Change the truth value of all variables on these rows
  - Write these rows as elementary disjunctions
  - Formulate the conjunctive expression

CNF:  $(p \vee q \vee r) \wedge (p \vee \bar{q} \vee r) \wedge (p \vee \bar{q} \vee \bar{r}) \wedge (\bar{p} \vee q \vee r) \wedge (\bar{p} \vee q \vee \bar{r})$

## EXAMPLE:

$p$	$q$	$r$	$f$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

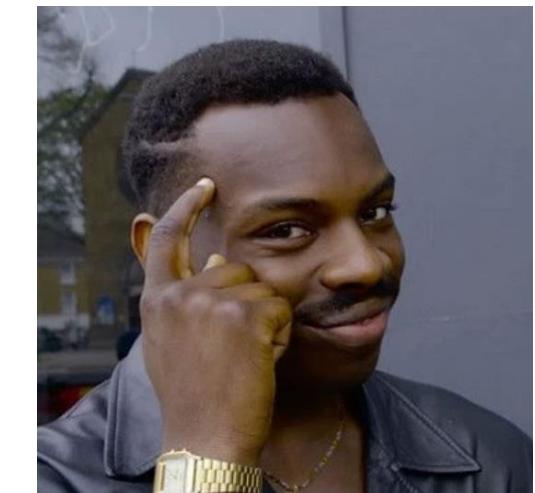
# Shorthand notation

- As you probably noticed, the mathematical notation containing conjunctions and disjunctions is not very reader-friendly
- In applications it is common to use shorthand notation:
  - Conjunctions are represented as multiplication (with multiplication symbol omitted)
  - Disjunctions are represented as summation (+)
  - Negations are represented by overbars OR an apostrophe (')
- This results in expressions which don't require a math editor!
- Example: DNF and CNF expressions from previous slides

$$DNF: (\bar{p} \wedge \bar{q} \wedge r) \vee (p \wedge q \wedge \bar{r}) \vee (p \wedge q \wedge r) \rightarrow p'q'r + pqr' + pqr$$

$$CNF: (p \vee q \vee r) \wedge (p \vee \bar{q} \vee r) \wedge (p \vee \bar{q} \vee \bar{r}) \wedge (\bar{p} \vee q \vee r) \wedge (\bar{p} \vee q \vee \bar{r})$$

$$\rightarrow (p+q+r)(p+q'+r)(p+q'+r')(p'+q+r)(p'+q+r')$$

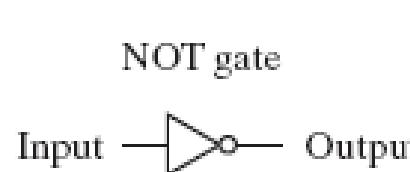


# Semiconductors

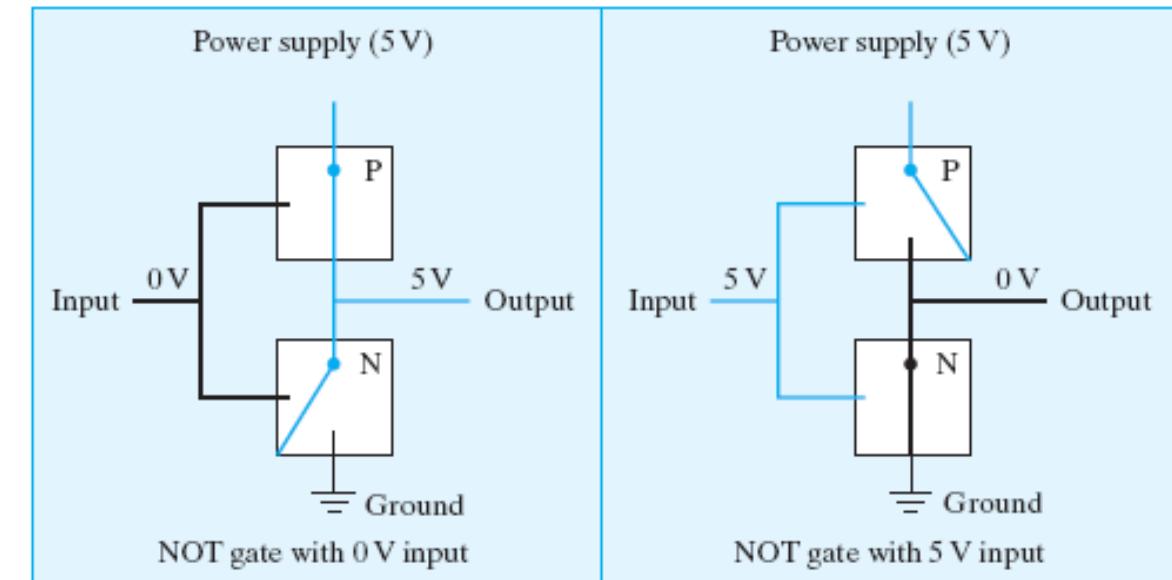
- Semiconductors are metals that can be manipulated to be either good or bad conductors of electricity
  - Most common semiconductor material is silicon
- This manipulation is done by adding impurities; the process is called “doping”
  - Negative doping = addition of atoms which have more valence electrons than base material
  - Positive doping = addition of atoms which have less valence electrons than base material
- Positively doped semiconductors are called p-type and negatively doped ones are called n-type
- Simple electrical components consist of these:
  - Diode = p-n-junction, that allows electricity to flow in only one direction
  - Transistor = two consecutive n-p-junctions, where electricity flow can be controlled via base voltage; can be used as a small and fast switch

# Logic gates

- Propositional logic and Boolean algebra can be used to design digital circuits
  - Digital = two voltage levels; “low” and “high” – corresponding to 0 and 1 in Boolean
- Diodes and transistors can be used to build logic gates that execute the Boolean functions in real life
- For example a NOT gate (negation)
  - Notice the drawing symbol (small circle)!

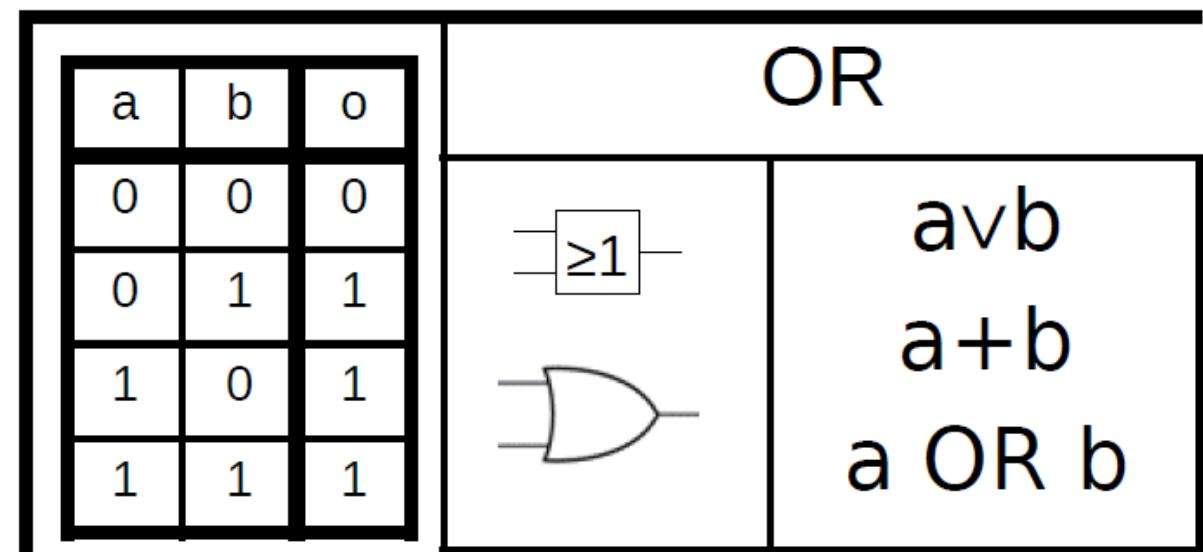
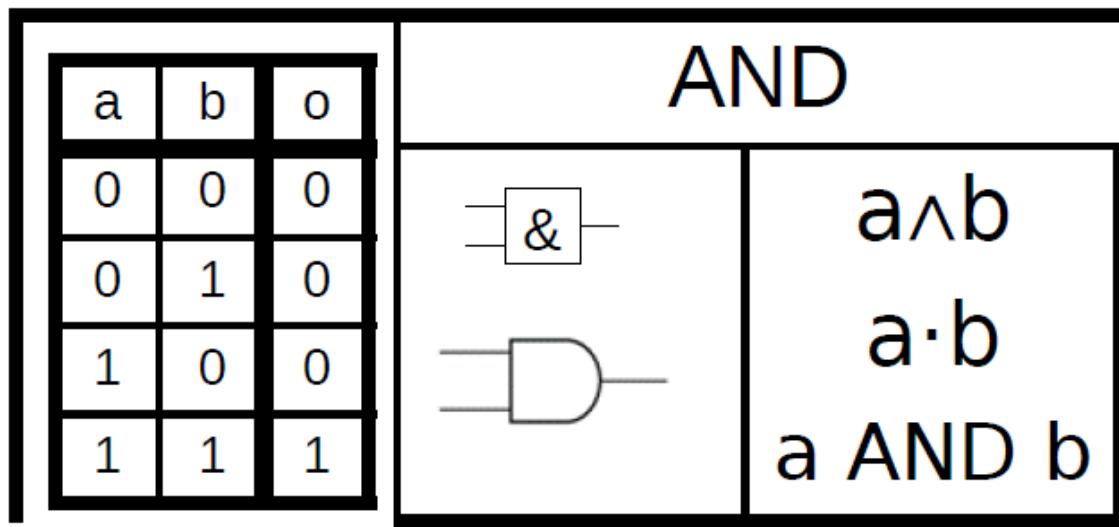


input	NOT output
0	1
1	0



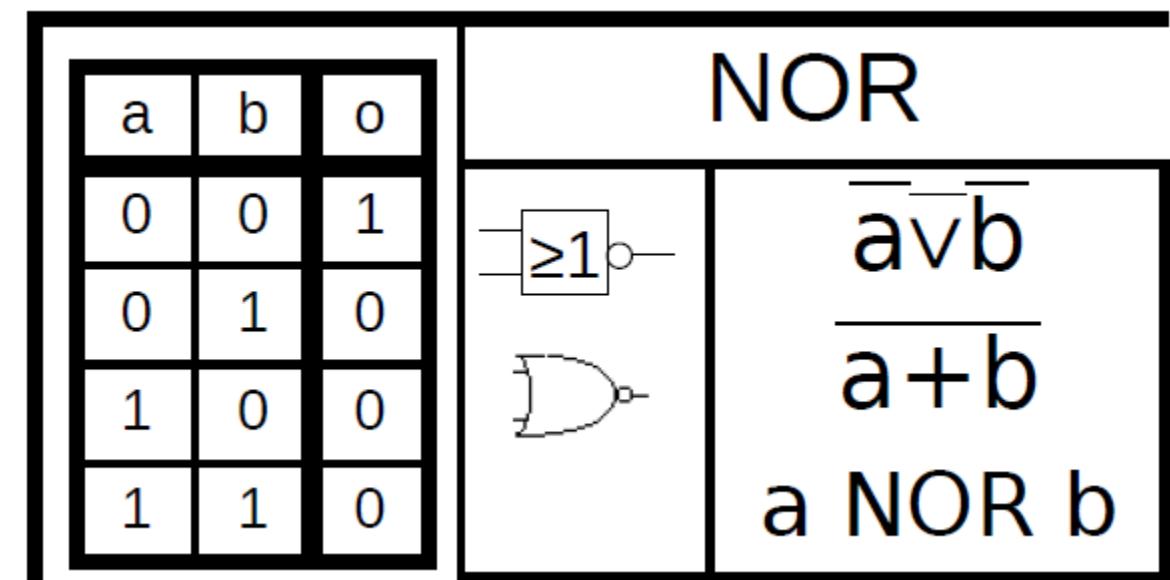
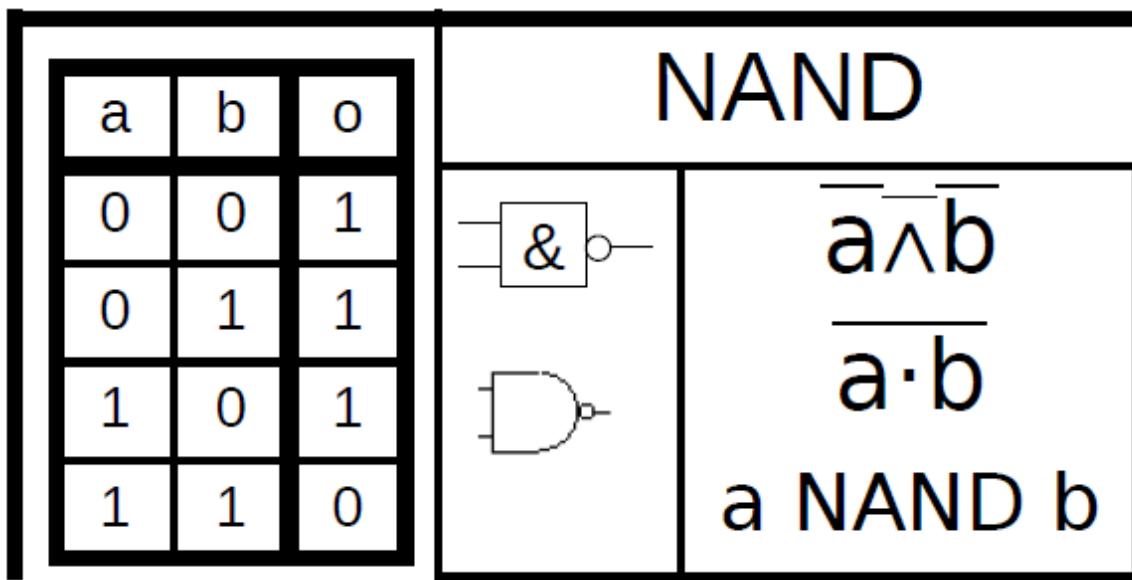
# Logic gates

- In similar fashion, AND & OR gates:
  - Two possible drawing symbols (the lower ones are more commonly used)
  - Inputs (left) are  $a$  and  $b$ ,  $o$  is the output (right)



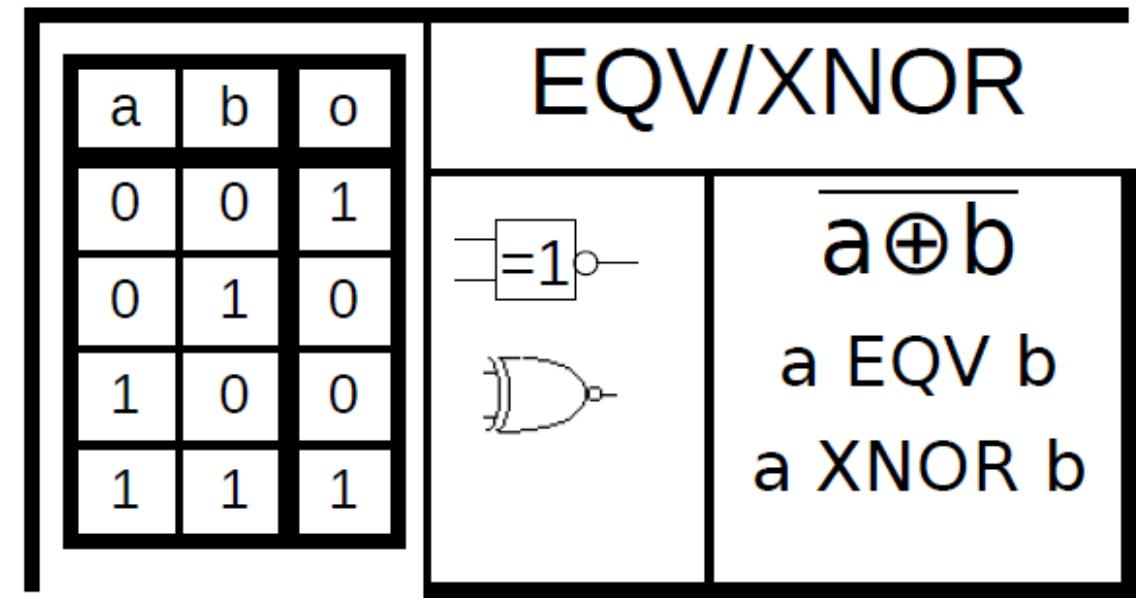
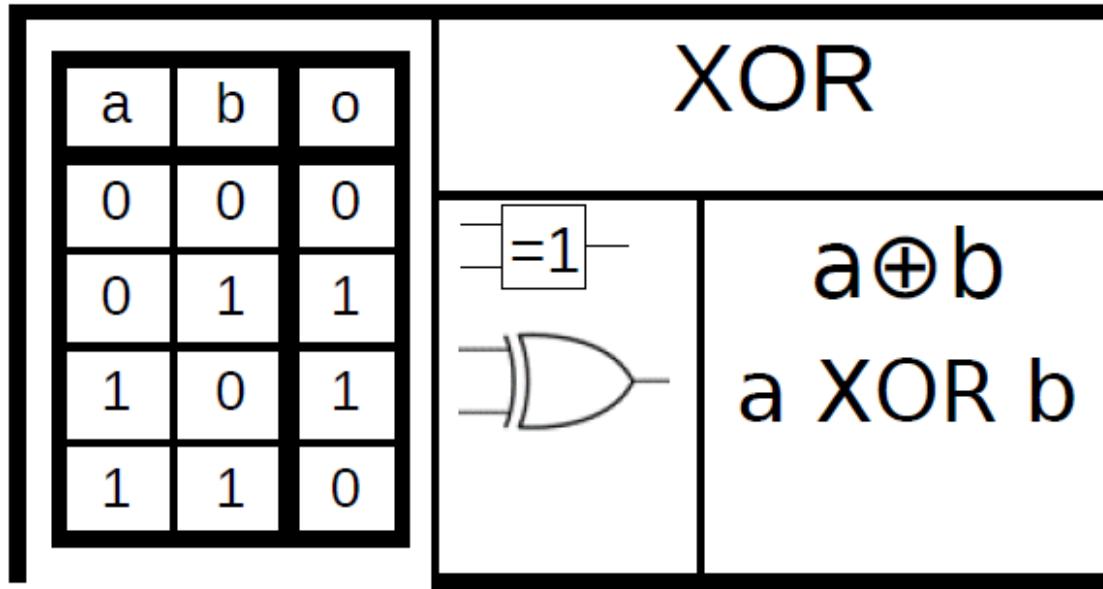
# Logic gates

- In addition to the three previous ones (which are basic Boolean gates), there are also additional gates which come in handy:
  - NAND = “Not AND”; returns 0 if both inputs are true and 1 otherwise
  - NOR = “Not OR”; returns 1 if both inputs are false and 0 otherwise



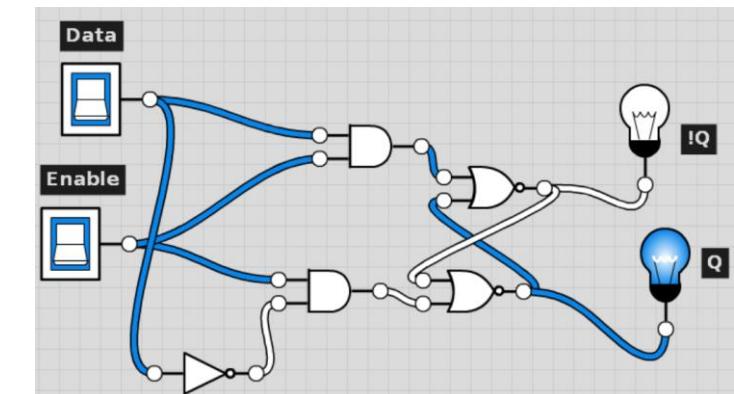
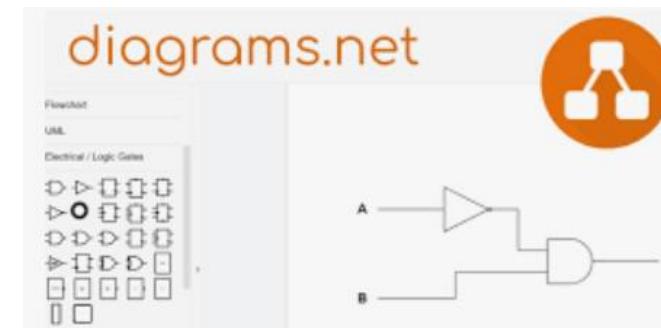
# Logic gates

- Two more:
  - XOR = “Exclusive OR”; returns 0 if inputs are the same and 1 if they differ
  - EQV or XNOR = “Equivalent”; returns 1 if inputs are the same and 0 if they differ

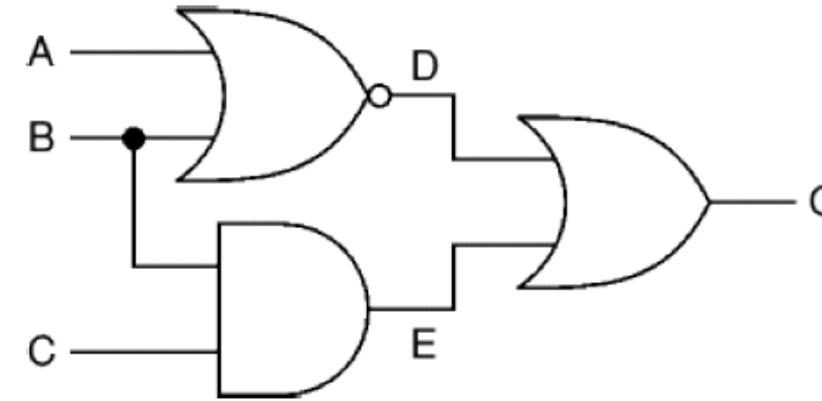


# Logic circuits

- More advanced logic circuits can be constructed by combining several gates
- The circuit can be presented by
  - Drawing a logic diagram using previous symbols
  - Giving a Boolean function
- The purpose and action of the circuit can be shown by a truth table
  - Note! A truth table doesn't uniquely specify the circuit; same truth table can be achieved by various circuits!
- Tip: There are several online tools which can be used to sketch a logic circuit and test how it works
  - logic.ly
  - draw.io etc...

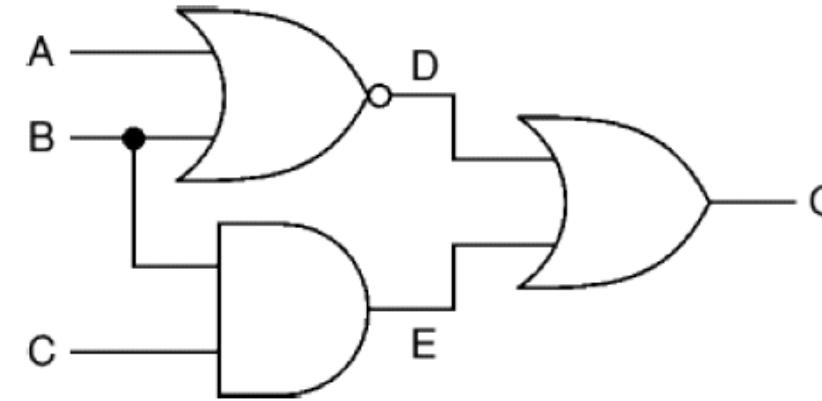


# From logic diagram to truth table



# From logic diagram to truth table

- 1) Decipher the gate symbols.



$$D = \text{NOT } (A \text{ OR } B)$$

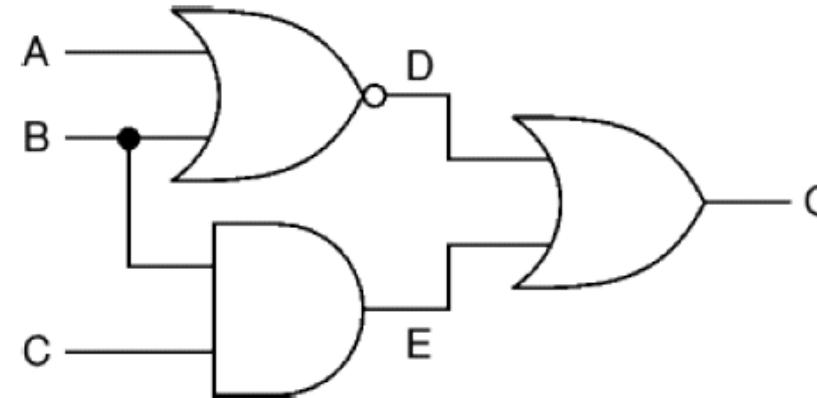
$$E = B \text{ AND } C$$

$$Q = D \text{ OR } E$$

$$= (\text{NOT } (A \text{ OR } B)) \text{ OR } (B \text{ AND } C)$$

# From logic diagram to truth table

- 1) Decipher the gate symbols.
- 2) Initialize truth table by writing columns for inputs.

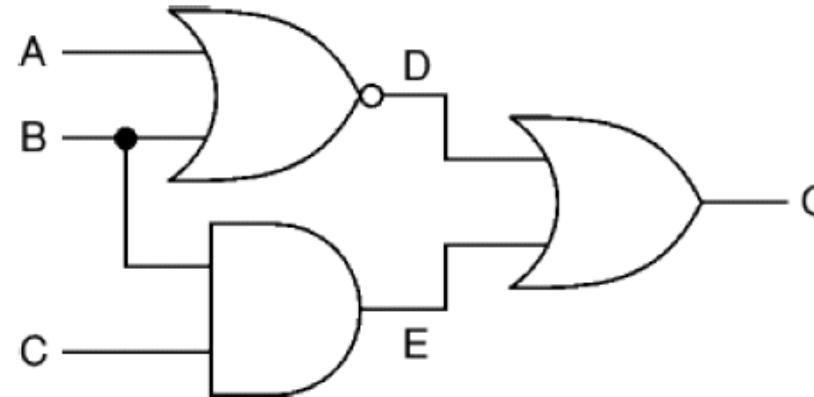


$$\begin{aligned}
 D &= \text{NOT } (A \text{ OR } B) \\
 E &= B \text{ AND } C \\
 Q &= D \text{ OR } E \\
 &= (\text{NOT } (A \text{ OR } B)) \text{ OR } (B \text{ AND } C)
 \end{aligned}$$

Inputs			Outputs		
A	B	C	D	E	Q
0	0	0	1	0	0
0	0	1	1	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	0	1	1
1	1	1	0	1	1

# From logic diagram to truth table

- 1) Decipher the gate symbols.
- 2) Initialize truth table by writing columns for inputs.
- 3) Find out the output values layer by layer.

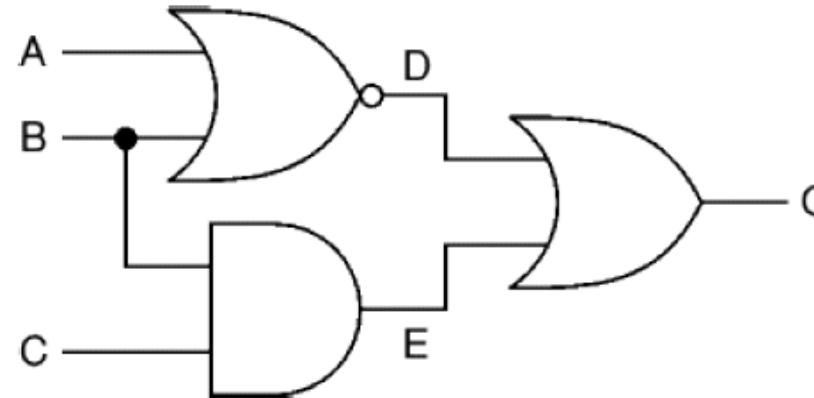


$$\begin{aligned}
 D &= \text{NOT}(A \text{ OR } B) \\
 E &= B \text{ AND } C \\
 Q &= D \text{ OR } E \\
 &= (\text{NOT}(A \text{ OR } B)) \text{ OR } (B \text{ AND } C)
 \end{aligned}$$

Inputs			Outputs		
A	B	C	D	E	Q
0	0	0	1	0	0
0	0	1	1	0	0
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

# From logic diagram to truth table

- 1) Decipher the gate symbols.
- 2) Initialize truth table by writing columns for inputs.
- 3) Find out the output values layer by layer.



$$\begin{aligned}
 D &= \text{NOT } (A \text{ OR } B) \\
 E &= B \text{ AND } C \\
 Q &= D \text{ OR } E \\
 &= (\text{NOT } (A \text{ OR } B)) \text{ OR } (B \text{ AND } C)
 \end{aligned}$$

Inputs			Outputs		
A	B	C	D	E	Q
0	0	0	1	0	1
0	0	1	1	0	1
0	1	0	0	0	0
0	1	1	0	1	1
1	0	0	0	0	0
1	0	1	0	0	0
1	1	0	0	0	0
1	1	1	0	1	1

# From truth table to circuit

A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

# From truth table to circuit

- 1) Formulate a Boolean function that matches the output of the truth table in either CNF or DNF

A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

**DNF:**

$$O = A'BC + AB'C + ABC' + ABC$$

# From truth table to circuit

- 1) Formulate a Boolean function that matches the output of the truth table in either CNF or DNF
- 2) Simplify the expression using Boolean laws (if possible)

A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

**DNF:**

$$O = A'BC + AB'C + ABC' + ABC$$

Use distributivity law: C of the first 2 terms, AB of the latter 2 terms

$$= (A'B + AB')C + AB(C' + C)$$

# From truth table to circuit

- 1) Formulate a Boolean function that matches the output of the truth table in either CNF or DNF
- 2) Simplify the expression using Boolean laws (if possible)

A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

**DNF:**

$$O = A'BC + AB'C + ABC' + ABC$$

Use distributivity law: C of the first 2 terms, AB of the latter 2 terms

$$= (A'B + AB')C + AB(C' + C)$$

Use complementation law on the latter:

$$= (A'B + AB')C + AB$$

# From truth table to circuit

- 1) Formulate a Boolean function that matches the output of the truth table in either CNF or DNF
- 2) Simplify the expression using Boolean laws (if possible)
- 3) Draw the circuit using standard drawing symbols

A	B	C	O
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

**DNF:**

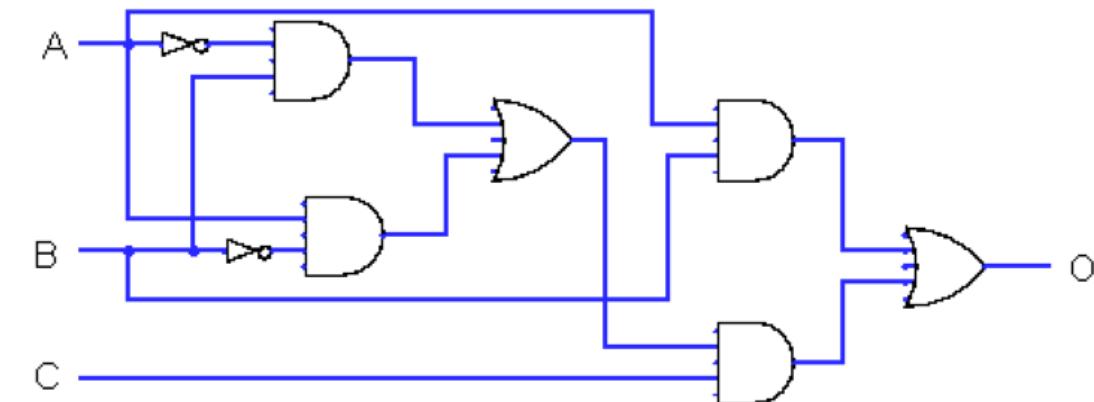
$$O = A'BC + AB'C + ABC' + ABC$$

Use distributivity law: C of the first 2 terms, AB of the latter 2 terms

$$= (A'B + AB')C + AB(C' + C)$$

Use complementation law on the latter:

$$= (A'B + AB')C + AB$$

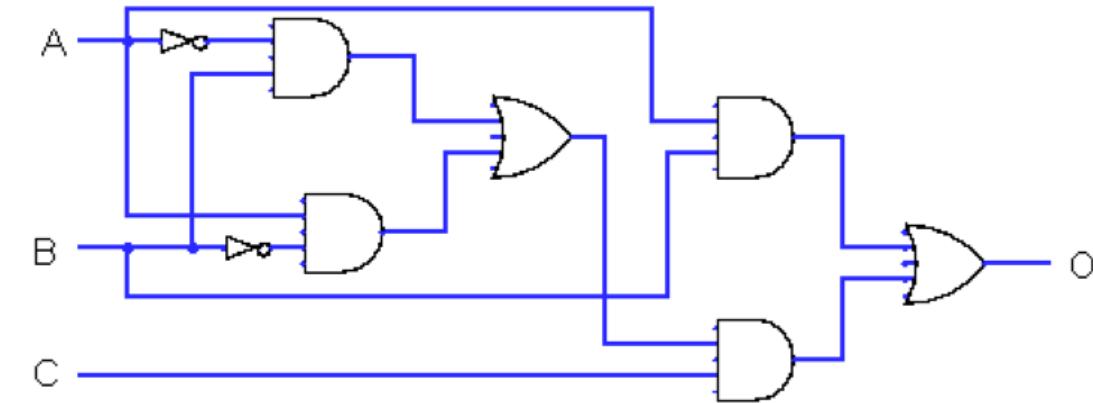


# From truth table to circuit

- Tip: you can check your circuit by counting the operations

$$(A'B + AB')C + AB$$

- 4 multiplications, 2 summations, 2 negations
- These must match the number of gates in your circuit!
  - 4 AND gates, 2 OR gates, 2 NOT gates
- If you want to use more advanced gates (NAND, NOR, XOR, EQV,...), this naturally doesn't work (because they're not Boolean operators)



Now it should be easy to notice why simplification is important: The original Boolean function in DNF was

$$A'BC + AB'C + ABC' + ABC$$

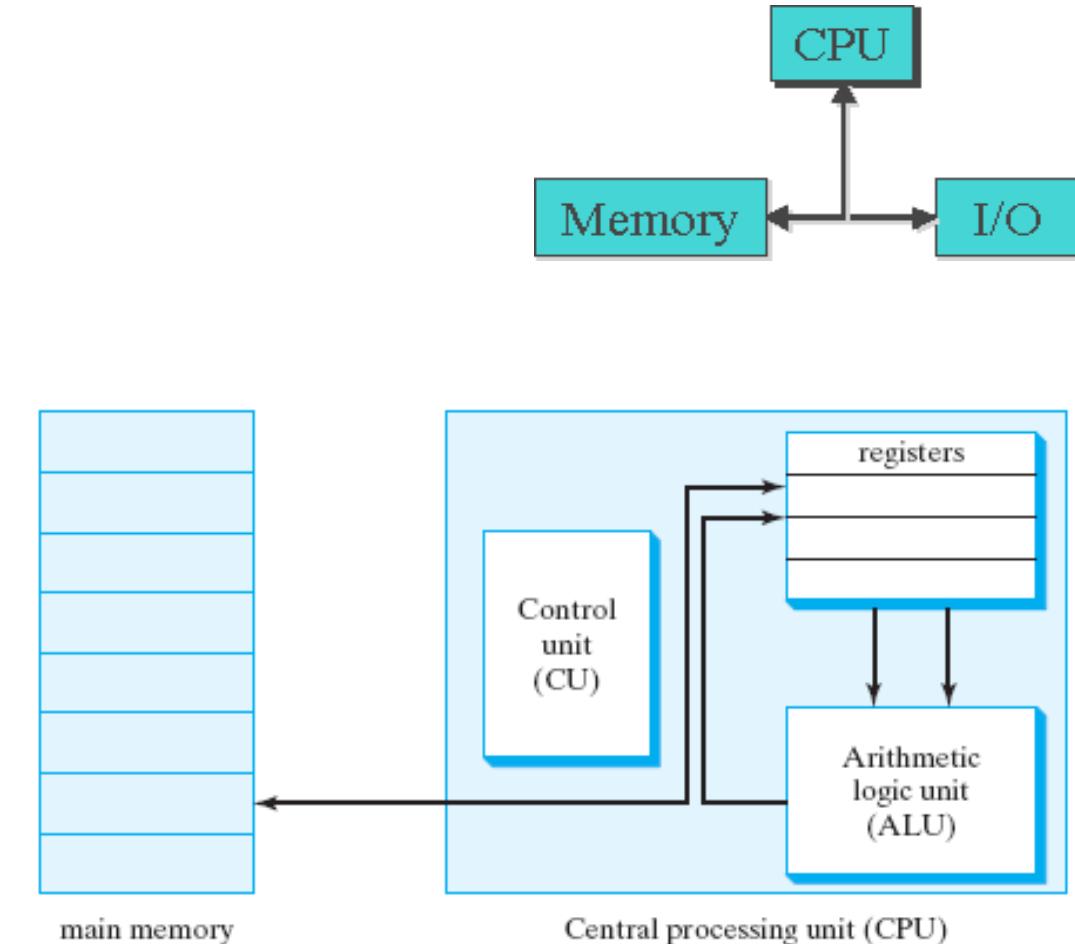
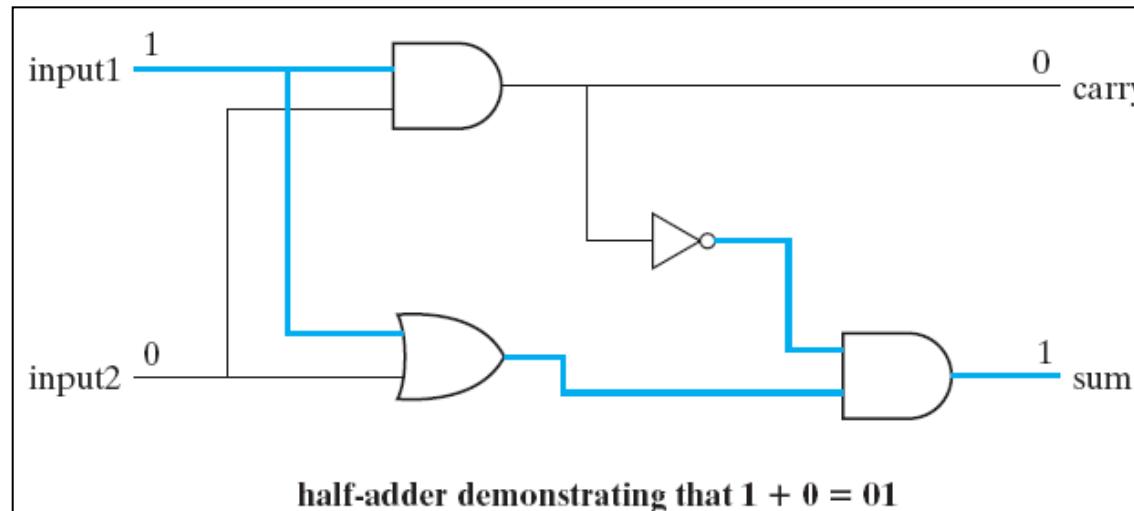
...this contains 8 multiplications, 3 summations, 3 negations -> in total 14 gates!

The simplified version contains only 8 gates.

# Thank you for listening!

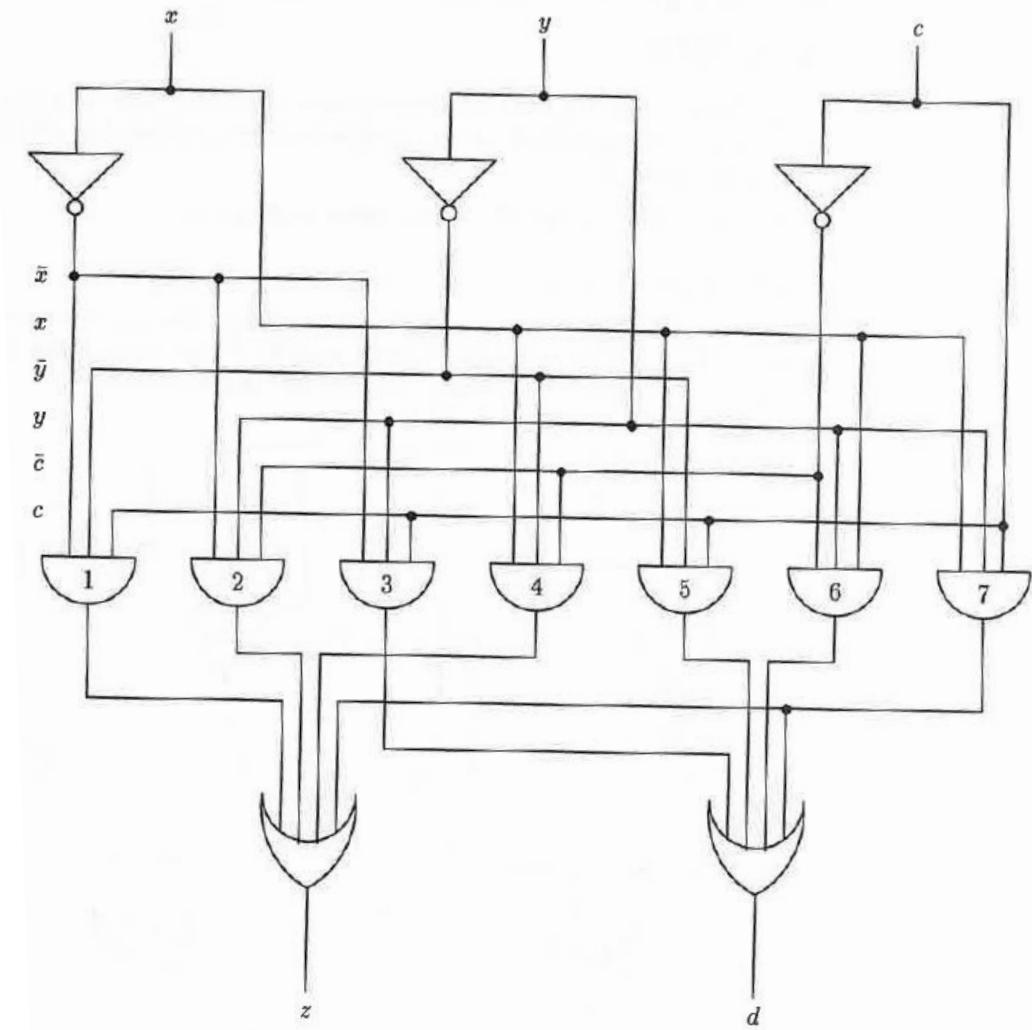


## 2. From logic gates to computer components



# Connections between logic gates

- When drawing more complex logic circuits, it is often needed to cross the lines that connect gates to each other
  - Is there a junction or not?
- Good way to improve notation:
  - If there's a black dot ("node") marked at the point of intersection, it's a junction
  - If there's no black dot, then the signals don't get mixed up
- Usually this notation is not actually needed, but it makes life easier
  - Especially in case feedback loops occur



# Half adder

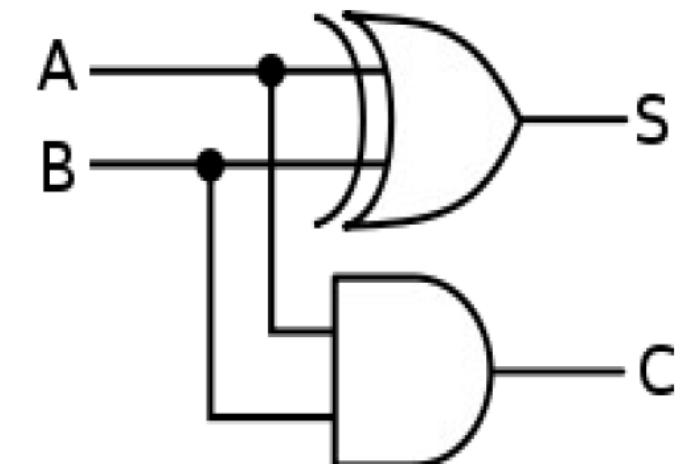
- A logic circuit that performs the addition of two 1-bit numbers is called a *half adder*
- Inputs: A and B
- Outputs: sum bit S and carry bit C
- Can be constructed in multiple ways, but the easiest way would be to use a XOR gate for the sum bit and an AND gate for the carry bit
- Disjunctive normal forms for output bits would be

$$S = AB' + A'B$$

$$C = AB$$

Why are there  
two outputs?

Input		Output	
A	B	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



# Ripple-carry addition algorithm

- Let's revise how we have learned to perform addition of regular base 10 numbers in elementary school:

$$\begin{array}{r} & 1 \\ & 4 \ 5 \ 6 \\ 8 \ 2 \ 9 & \hline 5 \end{array} \quad \begin{array}{r} 0 \\ 4 \ 5 \ 6 \\ 8 \ 2 \ 9 \\ \hline 8 \ 5 \end{array} \quad \begin{array}{r} 4 \ 5 \ 6 \\ 8 \ 2 \ 9 \\ \hline 1 \ 2 \ 8 \ 5 \end{array}$$

- Here we've used the *Ripple-carry addition algorithm*:

- If the sum of numbers in a column goes to double digits, we carry the first number to the next column (in this example,  $6+9 = 15$ , so we write 5 below and carry the 1 to the next column)

- The same logic applies for addition of binary numbers

- So, the “carry bit” C is literally a carry bit!
- Binary addition rules:

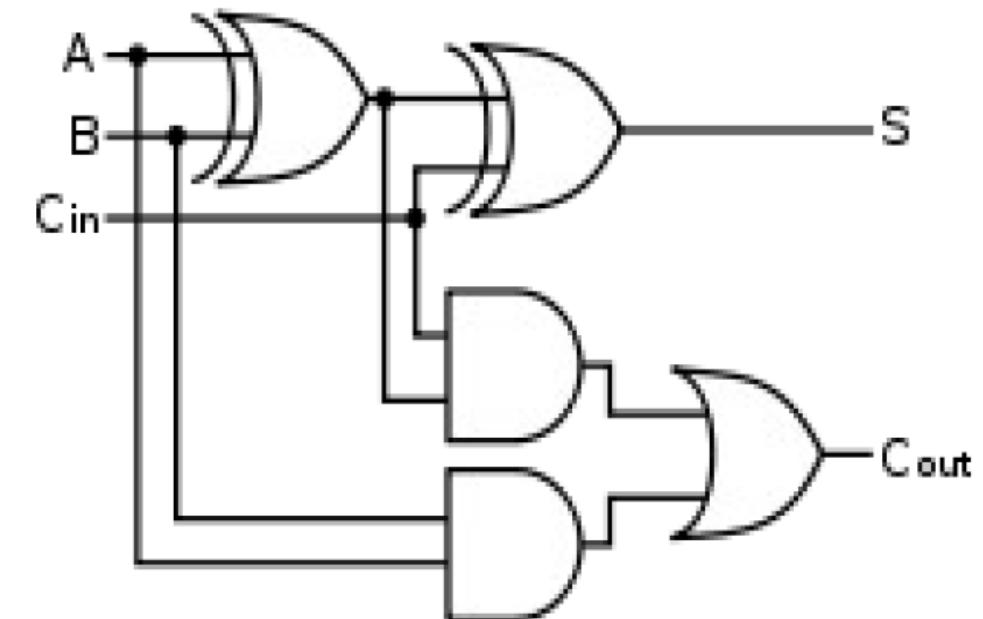
$$\begin{array}{r} 0 \\ + 0 \\ \hline 0 \end{array} \quad \begin{array}{r} 1 \\ + 0 \\ \hline 1 \end{array} \quad \begin{array}{r} 0 \\ + 1 \\ \hline 1 \end{array} \quad \begin{array}{r} 1 \\ + 1 \\ \hline 10 \end{array}$$

# Full adder

- Calculation of numbers which are more than 1 bit long requires taking this carry bit into account in further columns
- This can be done by constructing a *full adder* from two half adders and an OR gate

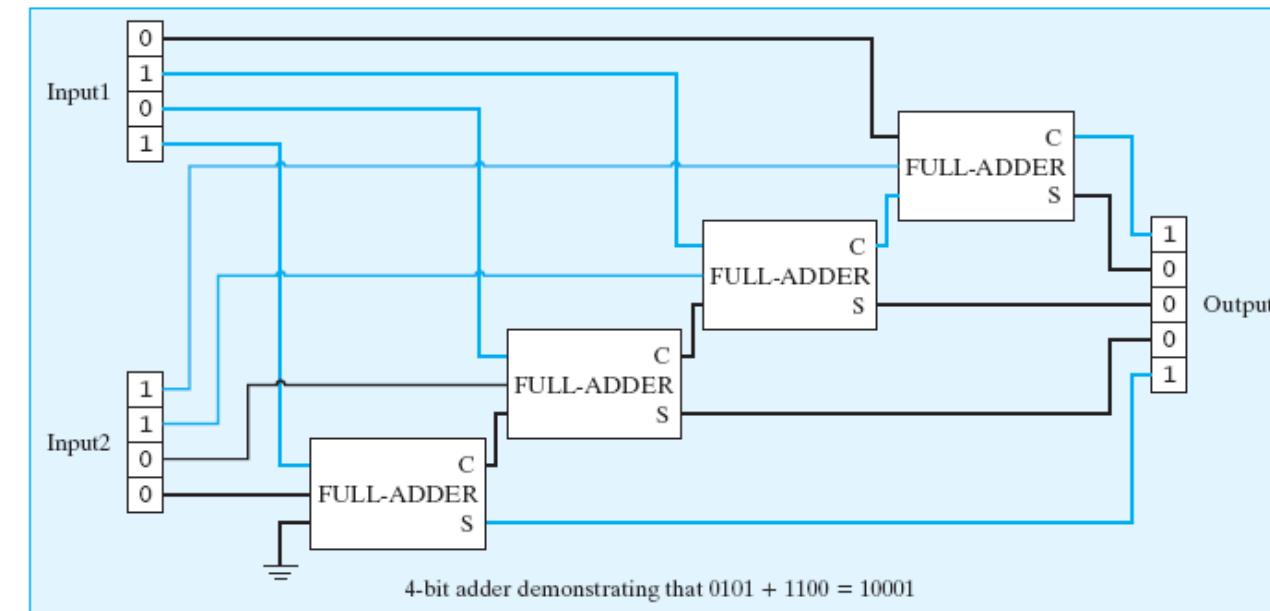
**Truth Table**

A	B	Carry-in	Sum	Carry-out
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



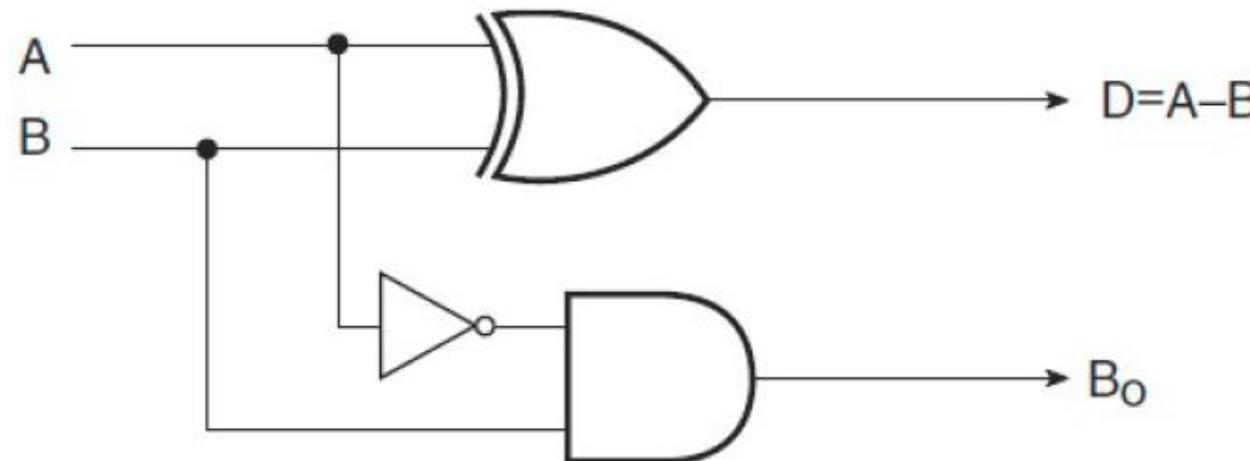
# Adder circuits

- By combining adders, we can construct *adder circuits* that are capable of performing addition calculations for much larger numbers
- Generally speaking: a circuit that performs the addition of two n-bit numbers consists of n pcs of full adder modules
  - Example: 4-bit full adder
  - The last carry bit can be taken into account (if the end result is  $n+1$  bits long), or if the output is also 4-bit, it can be neglected (overflow)



# Half subtractor

- Likewise, a subtraction of two 1-bit numbers is possible to do using logic gates
- A circuit that does this is called a *half subtractor*
  - Identical to half adder with the exception of a NOT gate before the AND gate
- n-bit full subtractors can be constructed, too



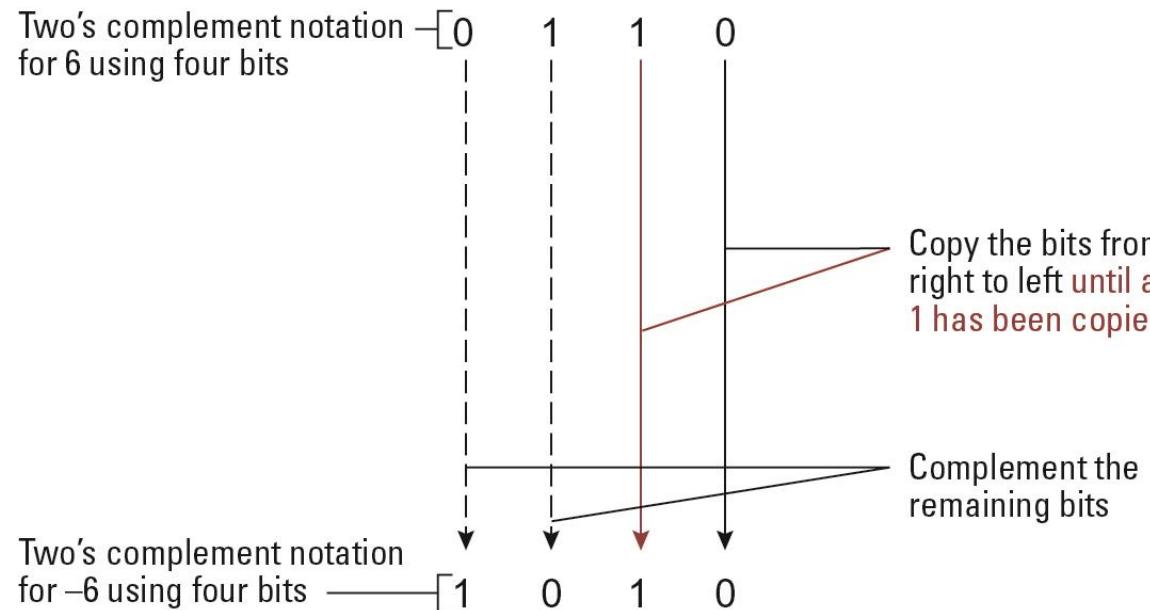
A	B	$B_o$	$D_i$
0	0	0	0
0	1	1	1
1	0	0	1
1	1	0	0

$$D_i = A'B + AB'$$

$$B_o = A'B$$

# Two's complement

- Instead of half/full subtractors it is possible to use *two's complement*:
  - Transform the number that's to be subtracted to its complement (remember from FoIP?)
  - Then perform the addition as before



a. Using patterns of length three

Bit pattern	Value represented
011	3
010	2
001	1
000	0
111	-1
110	-2
101	-3
100	-4

b. Using patterns of length four

Bit pattern	Value represented
0111	7
0110	6
0101	5
0100	4
0011	3
0010	2
0001	1
0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

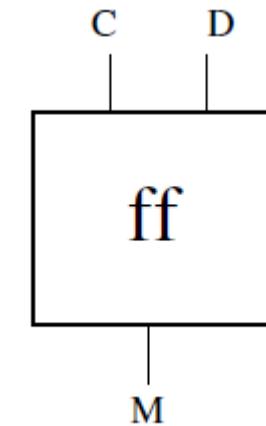
# Two's complement, calculation examples

- Some examples calculated both in base 10 and in base 2 using two's complement
- (This is revision from FoIP course)

Problem in base 10	Problem in two's complement	Answer in base 10
$\begin{array}{r} 3 \\ + 2 \\ \hline \end{array}$	$\begin{array}{r} 0011 \\ + 0010 \\ \hline 0101 \end{array}$	5
$\begin{array}{r} -3 \\ + -2 \\ \hline \end{array}$	$\begin{array}{r} 1101 \\ + 1110 \\ \hline 1011 \end{array}$	-5
$\begin{array}{r} 7 \\ + -5 \\ \hline \end{array}$	$\begin{array}{r} 0111 \\ + 1011 \\ \hline 0010 \end{array}$	2

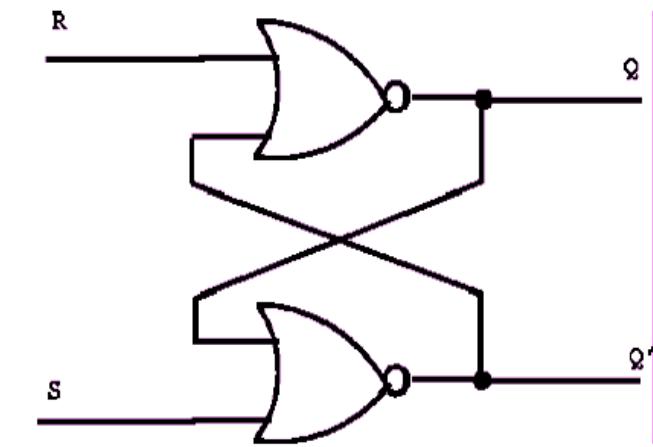
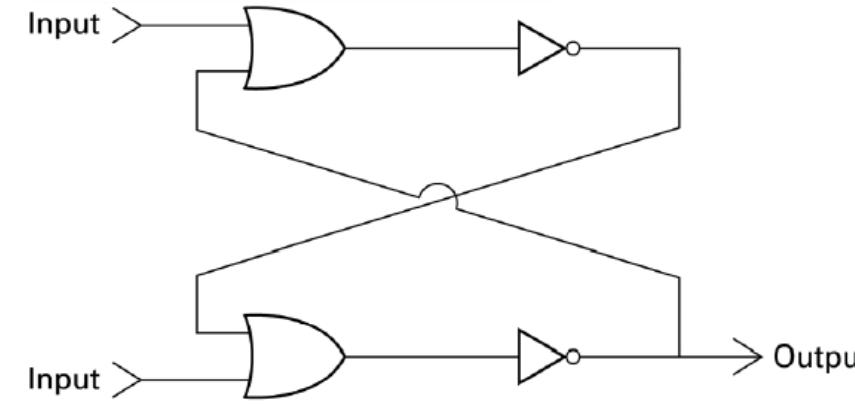
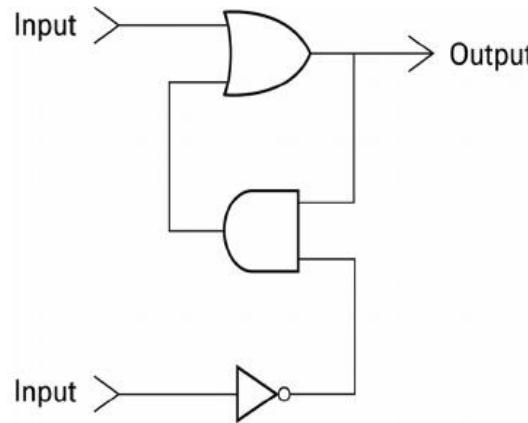
# Flip-flops

- A *flip-flop* is a logic circuit that has memory: its action depends not only on the inputs, but also on what's happened before
- In other words, it's capable of storing a bit value
- Flip-flops can be asynchronous or synchronous
  - Asynchronous flip-flops are called *latches*, and they change their state right away when the input signal changes
  - Synchronous flip-flops are controlled via clock signal; they change their state “on their turn”
- A flip-flop has two inputs:
  - Control bit (C)
  - Data bit (D)
- The output bit depends on C, D and previous value of M



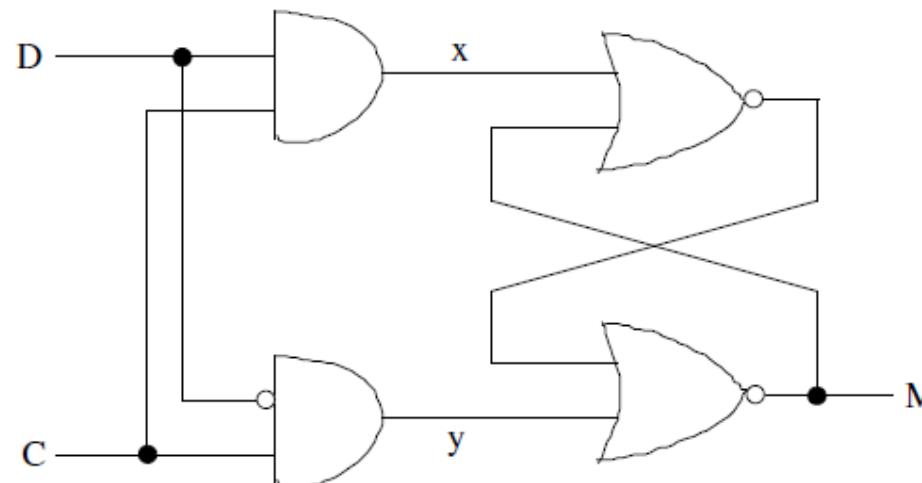
# Flip-flops

- A flip-flop can be constructed in several ways using different gates – for example
  - OR, AND & NOT gates
  - 2 OR gates + 2 NOT gates (or, simply, 2 NOR gates)
- Common feature in all of them is a *feedback loop* (or two)
- Even a case of two outputs (result and its complement) is possible



# Flip-flops

- Because flip-flop remembers history, the truth table for a flip-flop must be constructed in such a way that the 3<sup>rd</sup> input is  $M_t$  and the output is  $M_{t+1}$
- Some flip-flops may be equipped with a control part in order to prevent unwanted input combinations
  - This might be because they would cause the circuit to be unstable



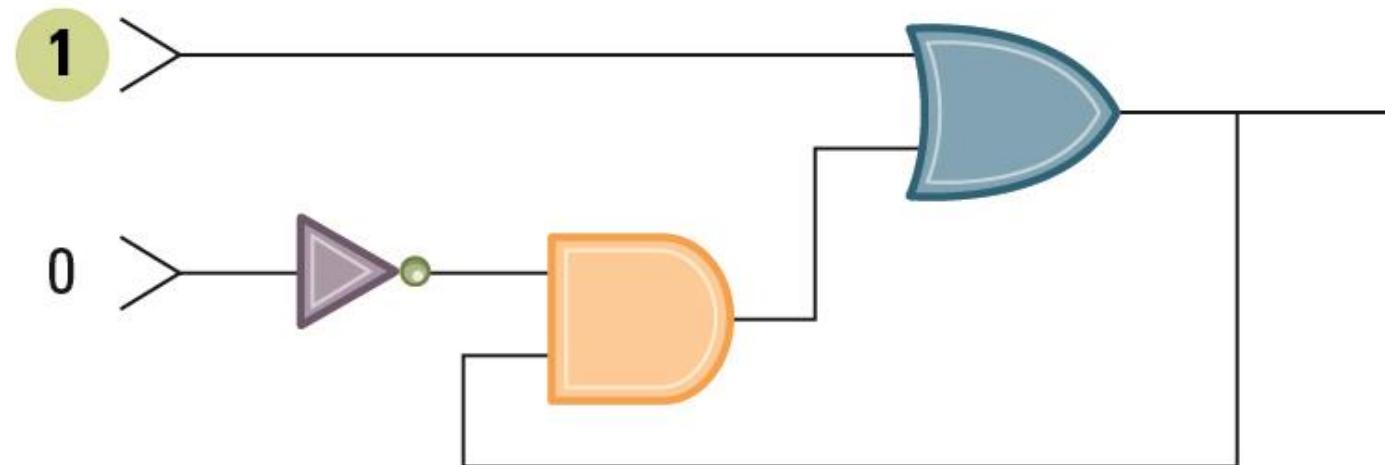
x	y	$M_t$	$M_{t+1}$
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1

Here the control part (two AND gates) causes that x and y can never be 1 at the same time.

# Simple flip-flop operation

- Let's examine how the following simple flip-flop operates:

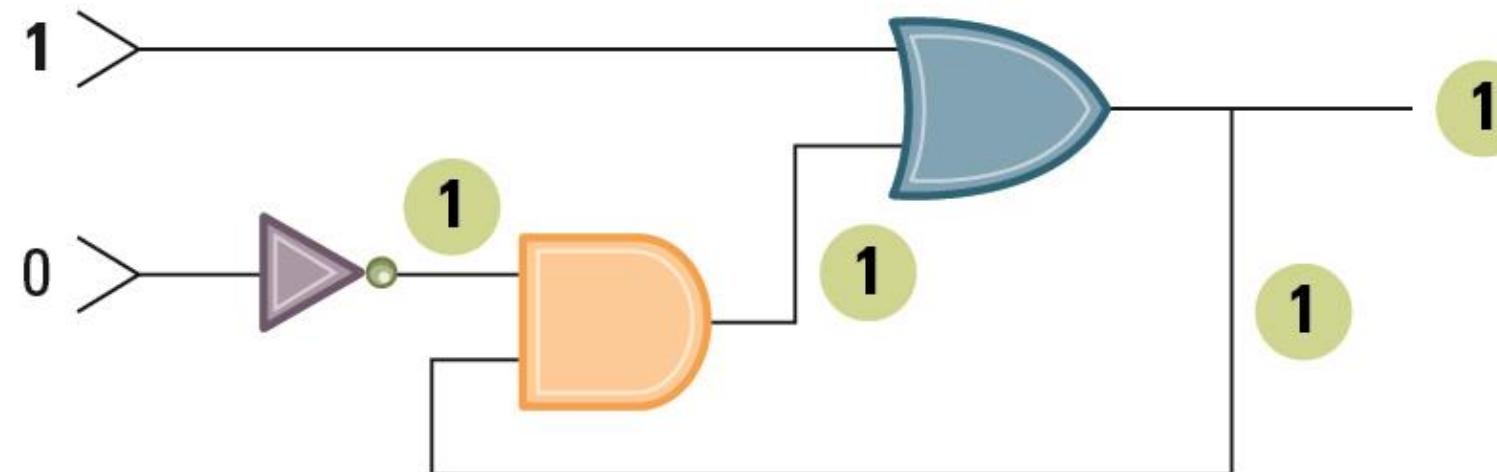
a. First, a 1 is placed on the upper input.



# Simple flip-flop operation

- Mark in the diagram what the values will be
- We notice, that the flip-flop is stable in this state

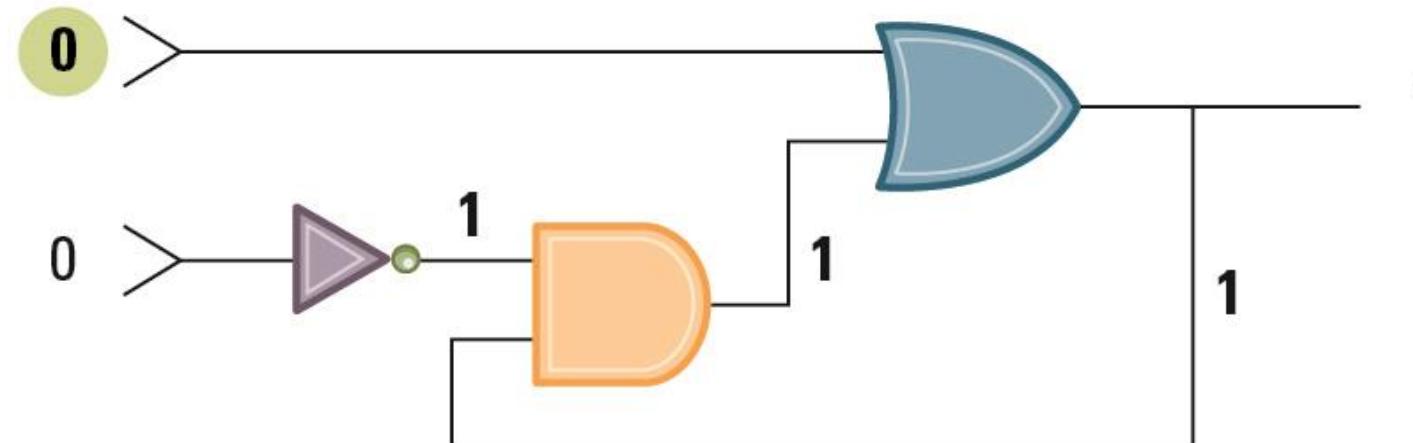
b. This causes the output of the OR gate to be 1 and, in turn, the output of the AND gate to be 1.



# Simple flip-flop operation

- Notice then, that changing the value of the upper input causes no changes
- The state of the flip-flop can be changed only by changing the lower input to 1
  - When the lower input is 1, the flip-flop has no memory (output is decided by upper input)

c. Finally, the 1 from the AND gate keeps the OR gate from changing after the upper input returns to 0.



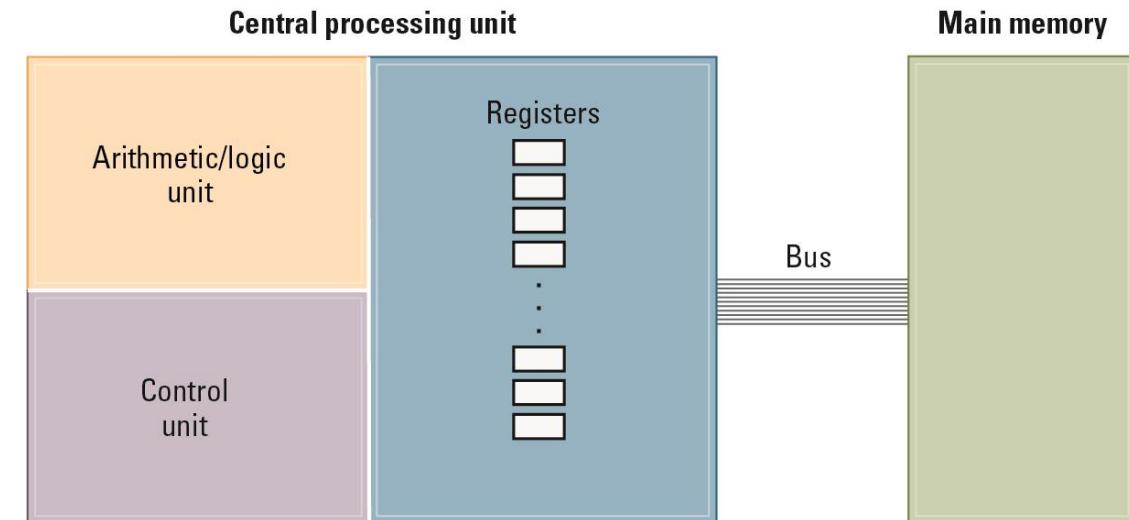
# Core components of a computer

- Computer is basically a logic circuit that consists of **very many** gates
- The structure is very modular, so it consists of components
- At least the following *logic components* (not parts!) are needed:
  - Arithmetic/logic unit (ALU)
  - Register unit
    - General-purpose registers
    - Special-purpose registers (instruction register, program counter)
  - Main memory
  - Buses
  - Clock
  - Input/Output (I/O)



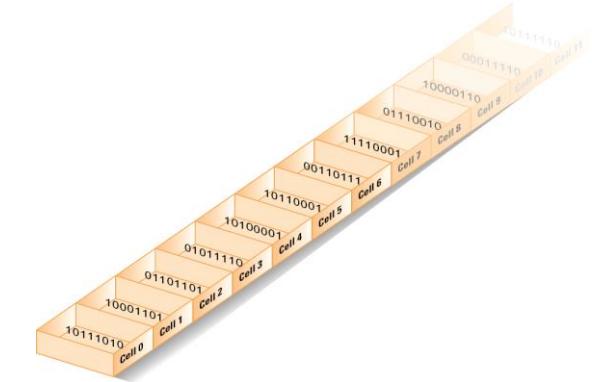
# Central Processing Unit (CPU)

- *Central Processing Unit* (known as “processor” or CPU) controls all manipulation of data; it contains three main parts:
  - Arithmetic/logic unit (ALU), which performs calculations and logic comparisons
  - Control unit, which is a “bookkeeper” of how the execution of the program is going – it coordinates the machine’s activities
  - Registers, which are built of flip-flops; the data needed by the program is temporarily stored here for rapid access
- CPU is connected to the main memory
- Connects via a bus



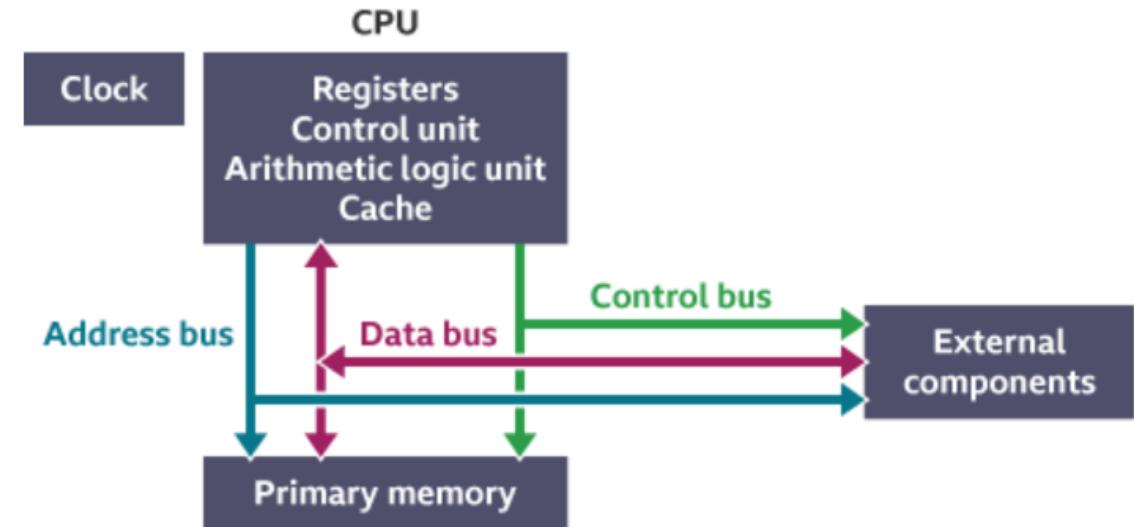
# Types of memory

- Generally, there are three types of memory in a basic computer
- Registers, which are inside the CPU
  - Rapid access, can be read and written; data needed immediately
- Main memory
  - Random Access Memory (RAM); data which is needed in near future
  - Quick, but not as quick as registers (because data has to be fetched via a bus)
  - Individual memory cells can be accessed in any order
  - Often Dynamic (DRAM), which is volatile: memory is cleared when the power is turned off
- Mass storage
  - Read-Only Memory (ROM); data which might be needed “at some point”
  - Relatively cheap and non-volatile (information is permanently stored)
  - Quite slow (due to memory type & slower bus connection)



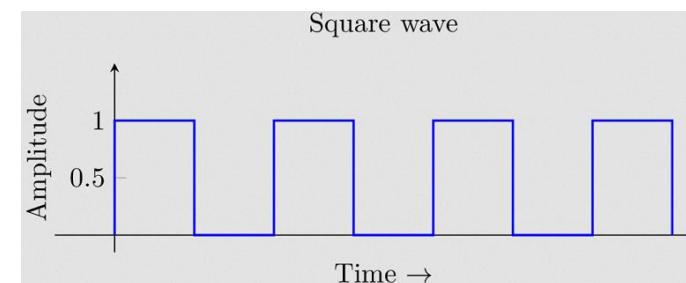
# Buses

- Different components in a computer are connected by *buses*, which transfer (or rather copy) information between components
- Bus bandwidth = the total amount of bits that can be transferred in a unit of time; usually this matches the word size of the computer (8bit, 16bit, 32bit, 64bit...)
- Buses can be divided in three groups:
  - Address bus: referral to desired memory address
  - Data bus: the data itself
  - Control bus: keeps the components informed



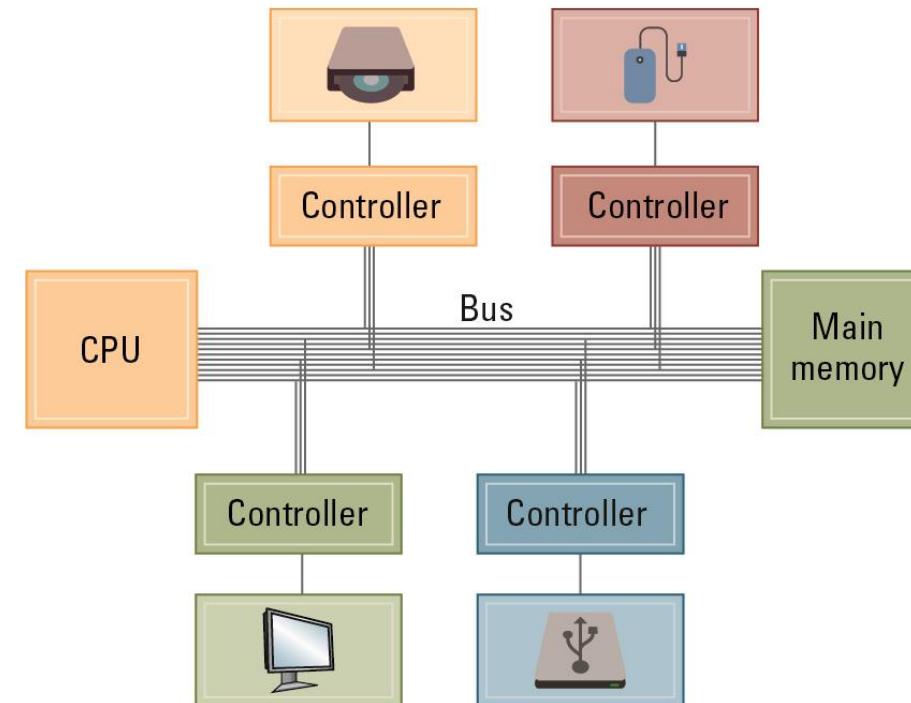
# Clock

- Components of the computer need a control signal in order to perform in a synchronized fashion
- Control signal, for example, commands when a flip-flop can save its input and orders when different components have the permission to transfer information to certain buses
- Control signal is created by the control logic and the clock circuit which vibrates
  - The clock generates a square wave (that consists of alternating 1 and 0 signals) on a certain frequency
  - During one signal the computer executes one elementary event
  - Therefore, the performance of the computer is partially dependent on the clock frequency



# Input/Output (I/O)

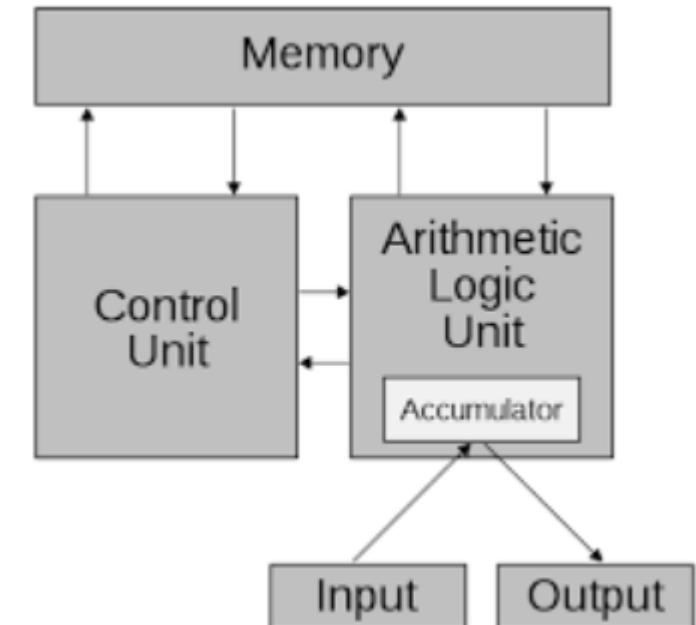
- In order to transfer information to and from the computer, different controllers (and also converters and communication protocols) are needed
- These devices are not essential, but often improve the possibilities of the computer
  - as well as user experience
- Connect to bus
- For example:
  - Monitor
  - Keyboard
  - Mouse
  - Printer
  - External drives
  - Internet connection



# Von Neumann architecture

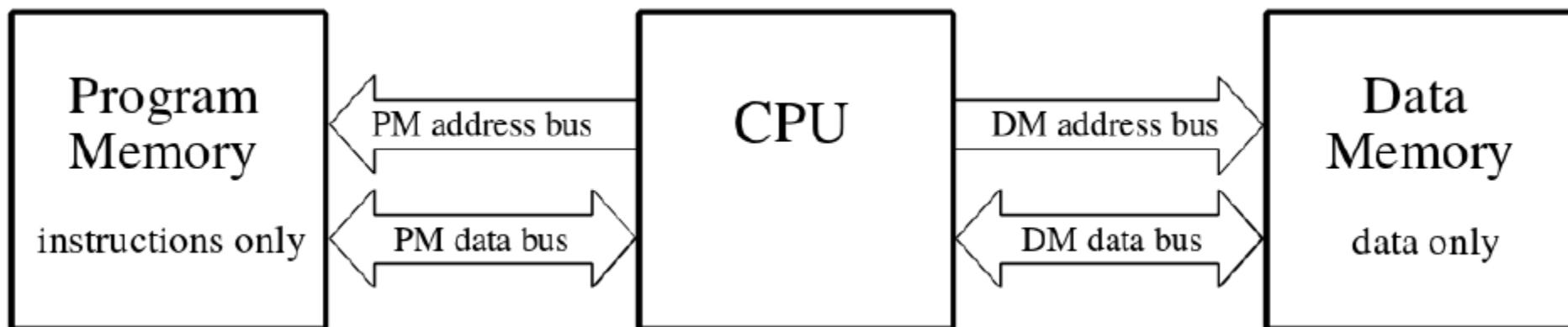
- Early computers were not flexible; they were made to perform one program, and if the program needed to be changed, it required actual re-wiring of the CPU
- John von Neumann\* came up with a solution and published it in 1945
  - *Stored-Program Concept*: a program can be encoded and stored in main memory just like data
  - Control unit extracts the program from memory, decodes the instructions and executes
  - Therefore, if we want to change the program, we only need to change the contents of the memory – no need for hardware changes!

\*Actually, the main development was made by a research team at University of Pennsylvania, led by J. P. Eckert.



# Harvard architecture

- One major disadvantage in von Neumann architecture is the von Neumann bottleneck: because only one bus can be accessed at a time, the CPU spends quite a lot of time just sitting idle and waiting for data to arrive
- Harvard architecture improved this by separating data memory and program memory – thus allowing the use of 2 buses simultaneously, which results in less CPU idle time and better performance
  - Downside: this architecture is expensive to manufacture

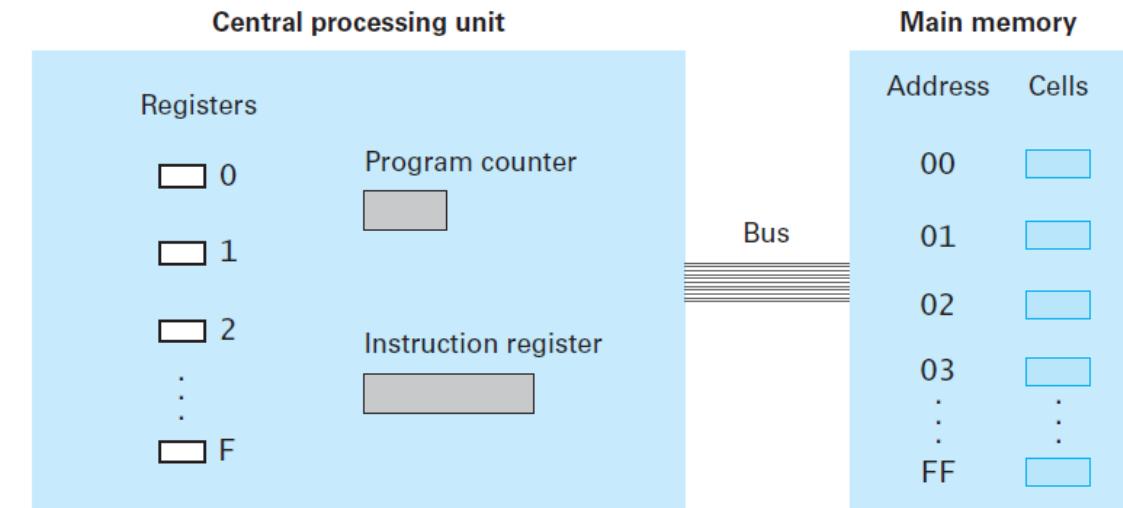


# Machine language

- In order to apply the stored-program concept, CPUs are designed to recognize instructions encoded as bit patterns
- This collection of instructions plus its encoding system is called machine language
  - Using this we can write machine-level instructions
- How many instructions should a CPU be able to decode and execute?
  - The less, the better – if we want efficiency in calculation
  - The more, the better – if we want ease of programming
- Two types of CPU philosophies:
  - RISC (Reduced Instruction Set Computer); few simple and efficient instructions
    - Used in mobile devices & consumer electronics, f.ex. ARM processors (Advanced RISC Machine)
  - CISC (Complex Instruction Set Computer); large set of convenient and powerful instructions
    - Used in desktop and laptop computers, f.ex. Intel & AMD processors
- Due to differences, RISC & CISC CPUs can't be easily compared in performance

# Machine cycle example

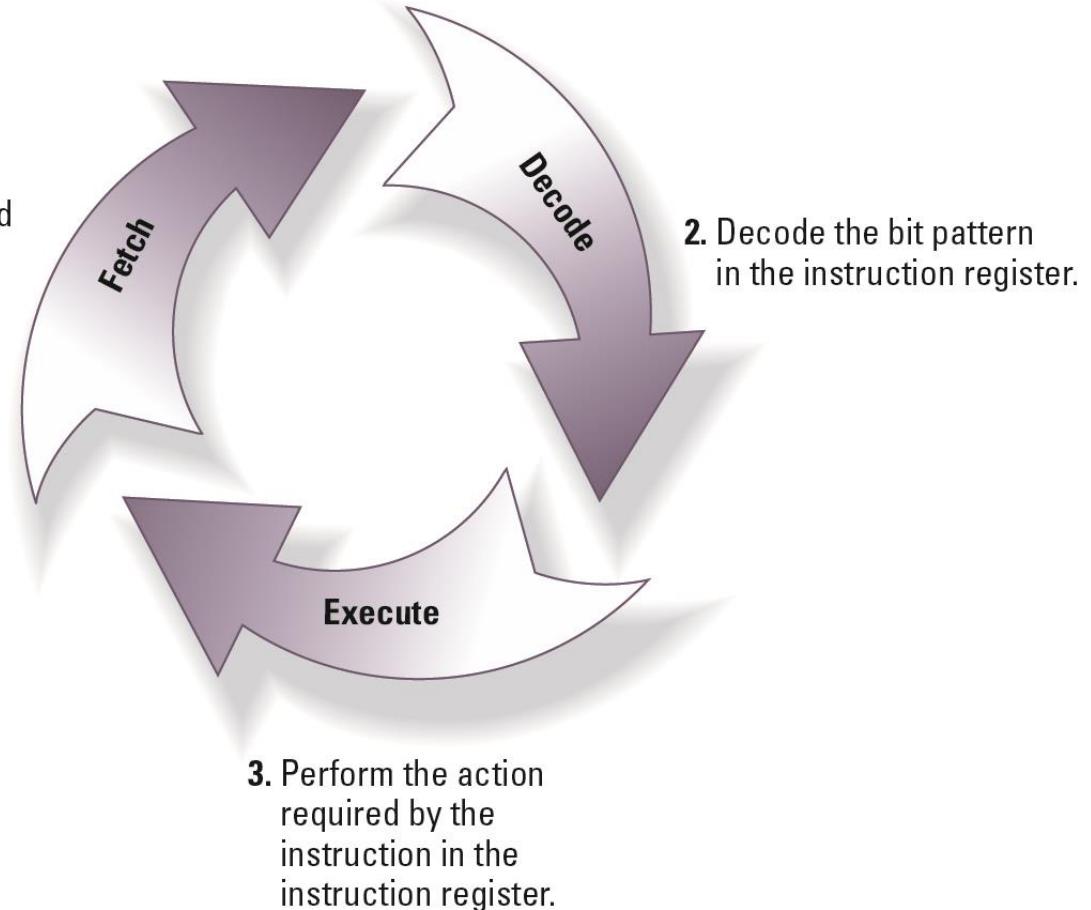
- Let's consider how a typical computer encodes instructions
- Our example computer has
  - 16 general-purpose registers (labeled 0...15)
  - 16-bit instruction register
  - 256 main memory cells, each with a capacity of 8 bits (labeled 0...255)
- Binary representation of these would be inconvenient, so we refer to registers and memory cells by hexadecimal notation:
  - Registers labeled 0...F
  - Memory cells labeled 00...FF



# Machine cycle

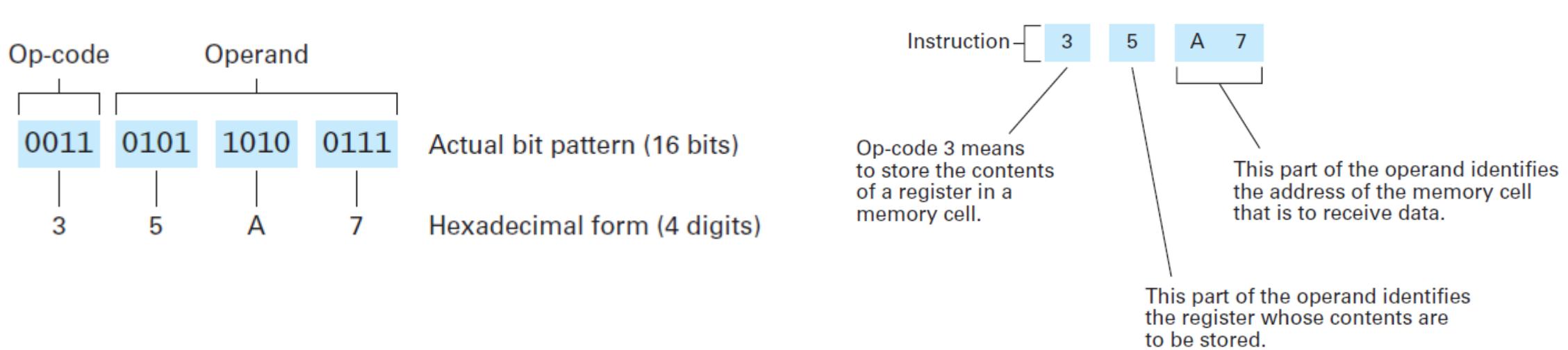
- CPU executes programs by repeating a three-phase process known as the machine cycle
- Phases are
  - Retrieve
  - Decode
  - Execute
- This process is repeated until the program halts

1. Retrieve the next instruction from memory (as indicated by the program counter) and then increment the program counter.



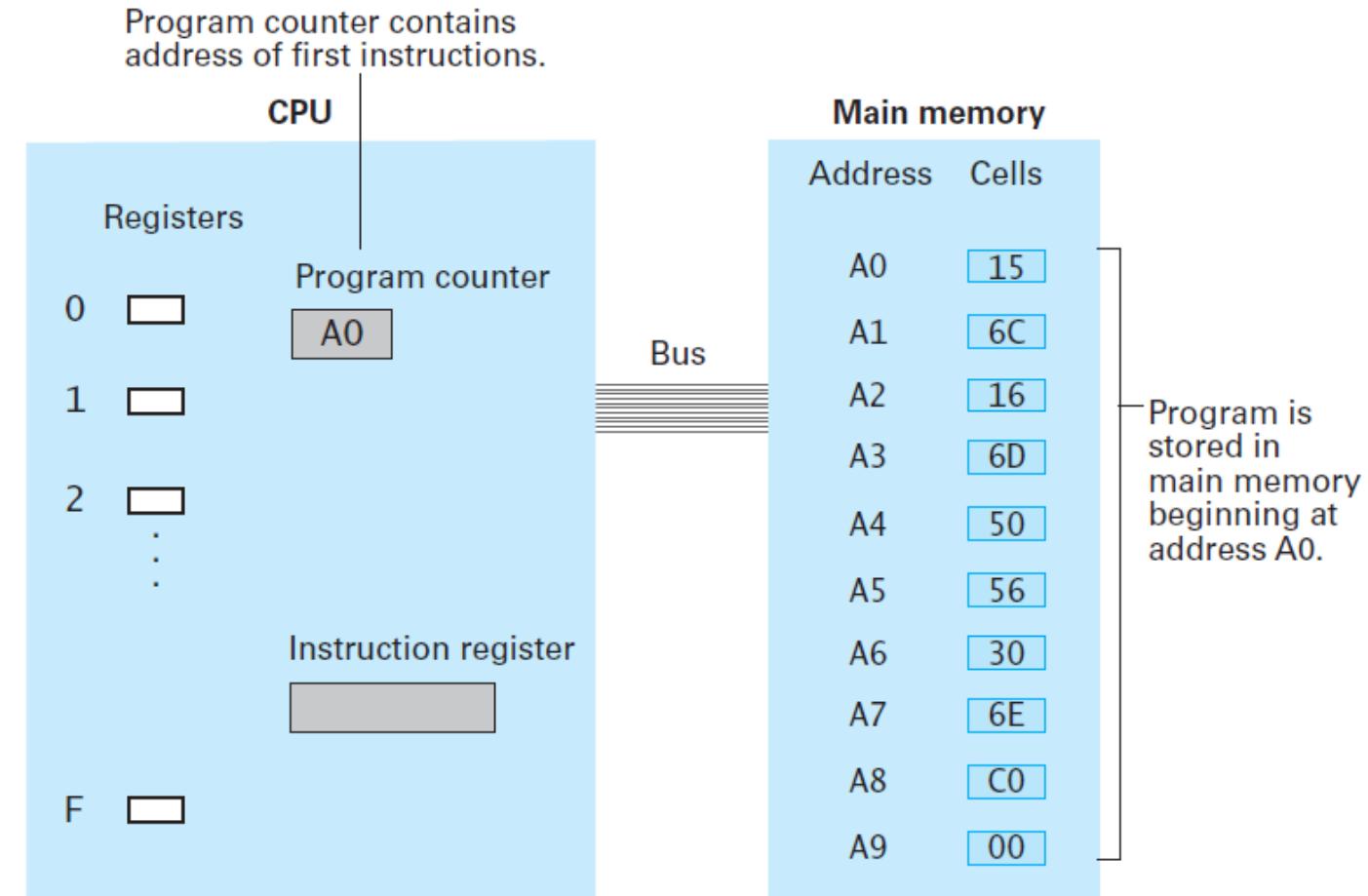
# Machine cycle example

- Length of all instructions is 16 bits, which consists of:
  - Op-code (first 4 bits), which specifies the operation (LOAD, STORE, JUMP,...)
  - Operand (following 12 bits), which clarifies the input of the operation and where do we put the output of the operation



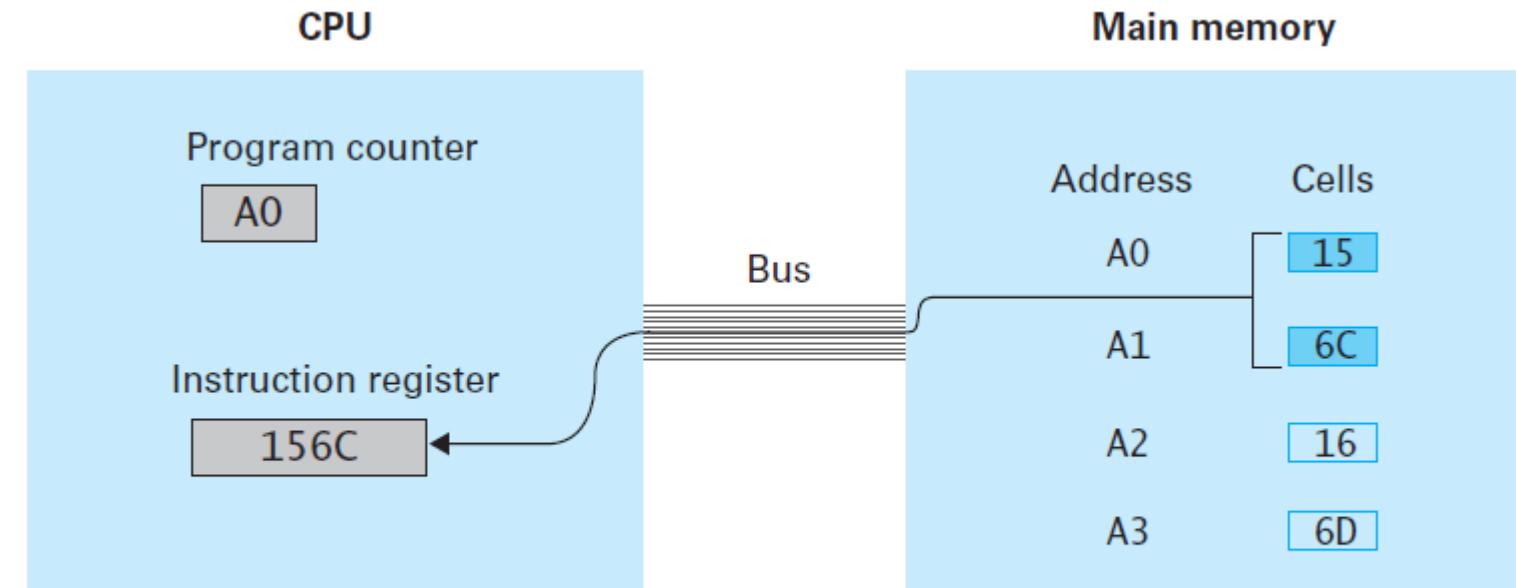
# Machine cycle example

- Initial state: program is stored in main memory, starting at A0.
- Program is started by setting the program counter to A0



# Machine cycle example

- At the beginning of the fetch step, instructions beginning from A0 are fetched
- Because the memory cells are 8bit and the instruction register is 16bit, the program fetches the contents of two consecutive memory cells (A0 and A1)

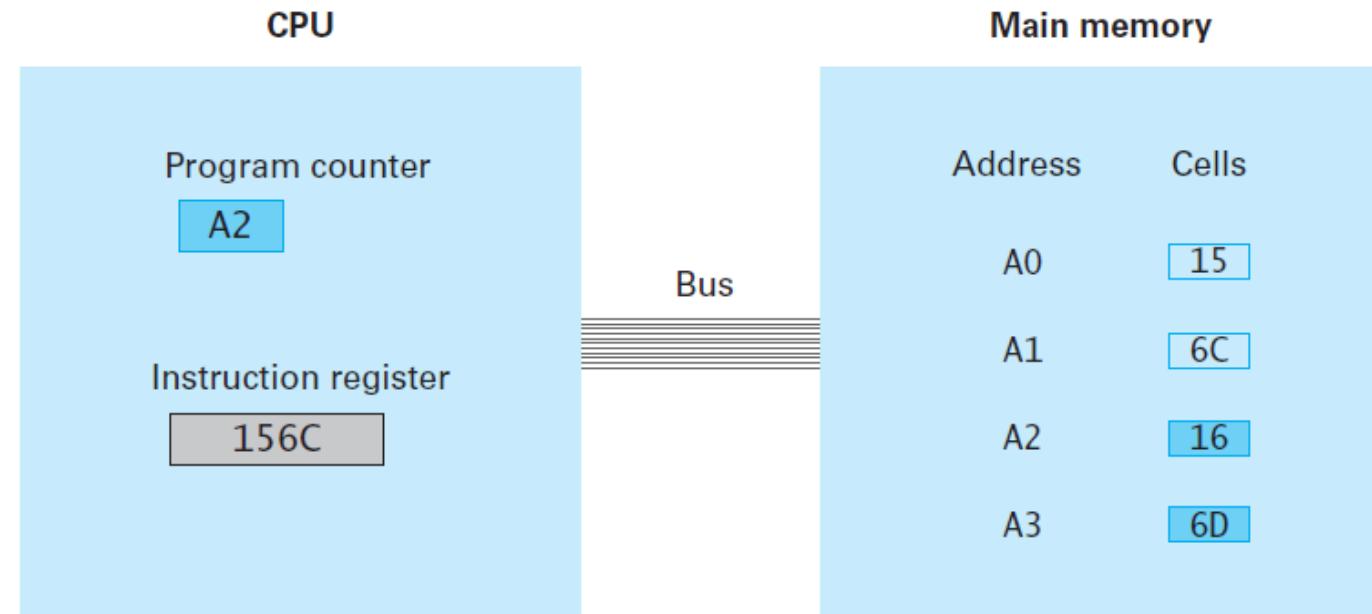


- a. At the beginning of the fetch step the instruction starting at address A0 is retrieved from memory and placed in the instruction register.

# Machine cycle example

- At the end of fetch step, the value of program counter is set to A2
- Decoding the instruction:
  - Op-code 1
  - Operand 56C
- Execute
- Fetch next instruction:  
166D etc.

Op-code	Operand	Description
1	RXY	LOAD the register R with the bit pattern found in the memory cell whose address is XY. <i>Example:</i> 14A3 would cause the contents of the memory cell located at address A3 to be placed in register 4.



b. Then the program counter is incremented so that it points to the next instruction.

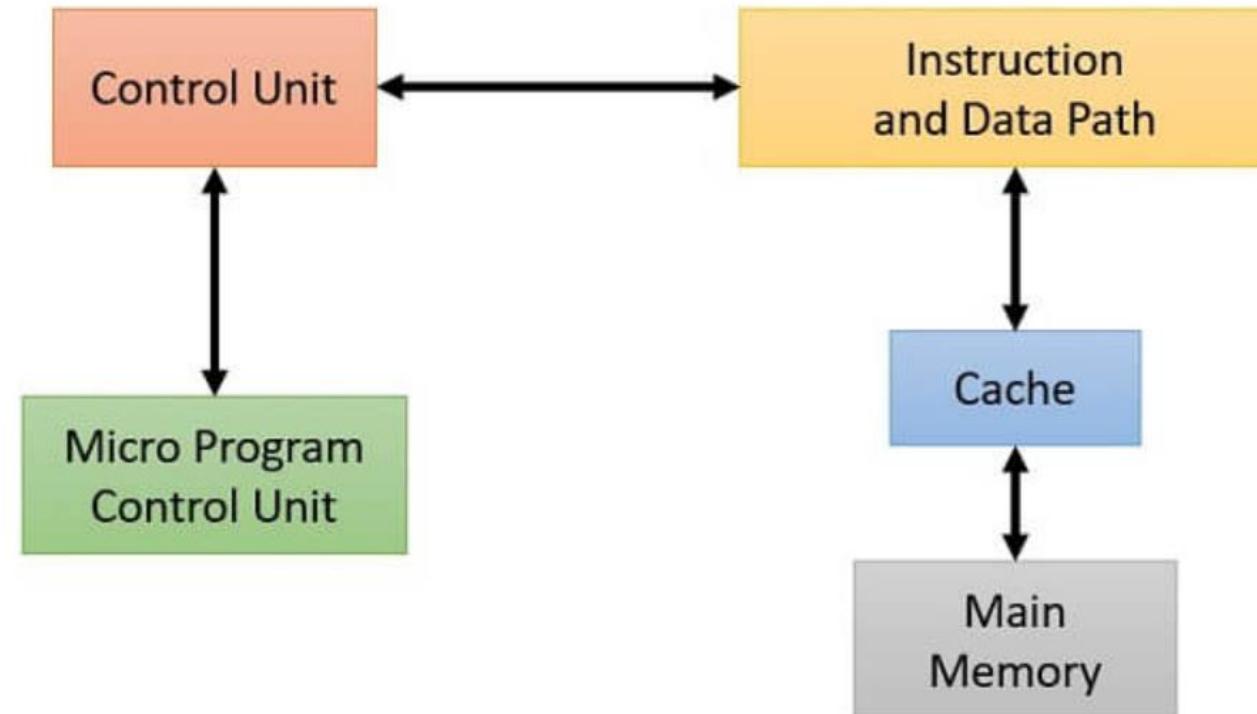


“Load data in memory cell at address 6C to register 5.”

# Thank you for listening!



### 3. Microprogrammed computer



CISC Architecture

# Microprogramming

- First computers were designed and built to execute simple machine language operations
  - Soon it was noticed, that many of these operations resembled each other
- Idea: separate a small very low-level group of basic commands and program each simple machine language command using these
  - This set of commands was named *microinstructions*
- Nowadays most modern computers execute microinstructions
- In order to do this, there needs to be an interpreter that decodes simple machine language to microcode
- Microcode is used to control the hardware of the computer
- Microprograms are often called *firmware*
  - Act as a link between the software and hardware

# Structure of a microprogrammed computer

- In this course, we will familiarize ourselves with the concept of a microprogrammed computer using the following 16-bit example computer, which is comprised of the following components:
  - Registers
  - Memory
  - ALU
  - Control bus & three data buses
  - Five-stage clock
- The components of our example computer can be divided to four “units”
  - Each “unit” plays one part in operation
  - “Units” are not physical parts, as you will soon notice
- See the structure graph of the example computer in Moodle!

# Unit 1: Control

- Microinstruction register (MIR)
  - 22-bit register, which includes the microinstruction that is currently in execution
  - Connected to control bus (CC), bits act as control bits for computer components
  - Control bit list can be found from slides 8-10
- Micro program memory (MPM), ROM
  - 22-bit register; storage unit of microinstructions
  - Read-only; instructions are “burned” on memory by the manufacturer
- Micro program counter (MPC)
  - 8-bit register, which tells the MPM address of the instruction in execution
  - Used in clock stages 4 (new search address) and 5 (new address given by DC3 saved to MPC)
- Clock (5 stages)
  - Gives a pulse (on their turn) for all five wires which activate bits in MIR

# Unit 2: Memory and data transfer

- Memory data register (MDR)
  - 16-bit register, which is used to transport data from main memory to registers A...D
- Main memory (MM), RAM
  - Storage unit of machine coded programs and their data
  - 16-bit memory slots
  - Random-access memory (read and write); much slower than MPM
  - Data transfer from and to registers happens via MDR
- Memory address register (MAR)
  - 12-bit register; used as an address when using MM
- Analogy: MDR is a taxi that transports people (data) from their home (MM) to their workplace (registers A...D) and back; MAR is the taxi central that stores rides

# Units 3 and 4: ALU & special registers

- Arithmetic-logic unit (ALU)
  - 16-bit full adder, which executes the calculations
  - Is capable of addition, subtraction using two's complement (compl) and multiplication by two (shiftleft)
- Special registers A, B, C and D
  - 16-bit registers, which are meant for maintaining data (operands and interim results)
  - Can be read in clock stage 1 and written to in clock stage 2
  - Register A values can also be compared in clock stage 4
  - Comparisons available:  $A < 0$  and  $A = 0$

# Buses & clock stages

- 22-bit control bus (CC)
  - MIR guides data transfer
- Three 16-bit data/address buses
  - DC1 and DC2 are used for providing inputs to ALU
  - DC3 is used to transfer the result from the ALU to the desired register
- In each clock stage, some of the following tasks can be done:
  - Stage 1: contents of MDR or number 1 is written in DC1 and content of the register A, B, C or D will be written in DC2 for arithmetic processing in ALU
  - Stage 2: The result from ALU is written from DC3 to some of the registers MAR, MDR, A, B, C or D
  - Stage 3: Contents of MDR are written to address given by MAR in MM – or contrariwise, contents of the MM address given by MAR will be written to MDR
  - Stage 4: New value of MPC will be calculated.
  - Stage 5: MPC gets a new value (automatically – no control bit!), and the microinstruction specified in MPC will be transferred to MIR register

# Control bits

- Different actions can only be performed during certain clock stages
- Control bits in clock stage 1:
  - c1 = write contents of register A to bus 2 (DC2)
  - c2 = write contents of register B to bus 2 (DC2)
  - c3 = write contents of register C to bus 2 (DC2)
  - c4 = write contents of register D to bus 2 (DC2)
  - c5 = write 1 to bus 1 (DC1)
  - c6 = write contents of MDR to bus 1 (DC1)
  - c7 = complement contents of bus 1 (DC1) (for subtraction purposes)
  - c8 = result of addition in bus 3 (DC3) is multiplied by 2 (shiftleft; see slide 13)

# Control bits

- Control bits in clock stage 2:
  - c9 = read contents of bus 3 (DC3) to register A
  - c10 = read contents of bus 3 (DC3) to register B
  - c11 = read contents of bus 3 (DC3) to register C
  - c12 = read contents of bus 3 (DC3) to register D
  - c13 = read contents of bus 3 (DC3) to MDR
  - c14 = read contents of bus 3 (DC3) to MAR
- Control bits in clock stage 3:
  - c15 = read contents of MM address given by MAR to MDR
  - c16 = write contents of MDR to MM address given by MAR

# Control bits

- Control bits in clock stage 4:
  - c17 = write 1 to bus 1 (DC1)
  - c18 = write 8 most significant bits of MIR to bus 1 (DC1)
  - c19 = if the contents of A is zero, write 1 to bus 1 (DC1); else, write 2 to bus 1 (DC1)
  - c20 = if the most significant bit of A is 1, write 1 to bus 1 (DC1) ; else, write 2 to bus 1 (DC1)
  - c21 = write 4 most significant bits of MDR to bus 1 (DC1)
  - c22 = write contents of MPC to bus 2 (DC2)
- Clock stage 5 contains no control bits

# Microprogramming

- Microprogrammed computer executes a program stored in main memory
- Each task is performed by executing a microprogram stored in MPM
- Fixed list of operations – the desired ones are activated via microinstructions
  - Analogy: music boxes – pins on the drum activate certain notes
  - When the drum rotates, a certain melody is heard
- Microprogramming enables changes in order of operations
  - Conditional skipping of commands (skip)
  - Unconditional jumps (jump)
  - ...these aren't possible in music boxes



# Microprogramming

- Basically microprogramming is regular programming; operations are very simple
- Each microinstruction contains 5 sub-instructions
  - 1 sub-instruction per each clock stage
  - Stage 1: right side of placement sentence (what is being calculated?)
  - Stage 2: left side of placement sentence (where the result is put?)
  - Stage 3: memory handling (MDR to MM or contrariwise)
  - Stage 4: calculate where to proceed next
  - Stage 5: move to next microinstruction
- Each microinstruction performs either a placement sentence or branching

# Example 1: Calculation of 1-bits

- Microprogram, which calculates how many 1-bits the word found in MM in address given by register D has, and saves the result to register C
- Our computer is only capable of performing comparisons in register A, so we need to work within that register
- We only have comparisons  $A < 0$  and  $A = 0$  available
  - Due to two's complement method, if the first bit of our word is 1, it's negative
  - So, if  $A < 0$  is true, the first bit is 1
  - There is no way to investigate the following bits one by one, so we have to modify the word by moving it one bit at a time to the left
- This can be done via the shiftleft ( $x2$ ) operation:
  - 1<sup>st</sup> bit is discarded, others move 1 step left
  - Last bit is set to 0

1	0	0	1	1	0	0	1
---	---	---	---	---	---	---	---



0	0	1	1	0	0	1	0
---	---	---	---	---	---	---	---

# Example 1: Pseudo-code algorithm

```
C := 0
A := get(D)
WHILE A <> 0 DO
    IF A < 0 THEN
        C := C + 1
    ENDIF
    shiftleft(A)
ENDWHILE
```

- This isn't possible to execute in a microprogrammed computer, because there are
  - Subprograms (get)
  - Loops (while)

# Example 1: Microprogram?

- Replace subprograms with microinstructions and loops with jumps:

C := 0

MAR := D

MDR := (MAR)

A := MDR

if: IF A = 0 THEN jump pass ENDIF

    IF A < 0 THEN C := C + 1 ENDIF

    shiftleft(A)

    jump if

pass:

- This is close, but the jump is inside an if clause. Conditional jumps are not ok.

# Example 1: Microprogram

- Change to conditional skips and unconditional jumps

```
C := 0
MAR := D
MDR := (MAR)
A := MDR
if: skip A = 0      If skip condition is true      If skip condition is false
    jump pass
    skip A < 0
    C := C + 1
    shiftleft (A)
    jump if
pass:
```

The diagram illustrates the control flow for the 'if' block. It shows two parallel execution paths. The first path, indicated by a green bracket and labeled 'If skip condition is true', executes the statements 'skip A = 0', 'jump pass', and 'C := C + 1'. The second path, indicated by a red bracket and labeled 'If skip condition is false', executes the statements 'skip A < 0' and 'shiftleft (A)'. Both paths converge at the 'jump if' statement.

## Example 1: Symbolic microprogram

```
1:    0+0→C; ; 1+MPC →MPC
2:    0+D →MAR; (MAR) →MDR; 1+MPC →MPC
3:    MDR+0 →A; ; 1+MPC →MPC
4:    ; ; (A=0) +MPC →MPC
5:    ; ; 10102 →MPC
6:    ; ; (A<0) +MPC →MPC
7:    1+C →C; ; 1+MPC →MPC
8:    (0+A) x2 →A; ; 1+MPC → MPC
9:    ; ; 1002 →MPC
10:
```

## Example 1: Symbolic microprogram

```
1: 0+0→C; ; 1+MPC →MPC
2: 0+D →MAR; (MAR) →MDR; 1+MPC →MPC
3: MDR+0 →A; ; 1+MPC →MPC
4: ; ; (A=0) +MPC →MPC
5: ; ; 10102 →MPC
6: ; ; (A<0) +MPC →MPC
7: 1+C →C; ; 1+MPC →MPC
8: (0+A) x2 →A; ; 1+MPC → MPC
9: ; ; 1002 →MPC
10:
```

Clock stage 1&2  
Clock stage 3  
Clock stages 4&5

# Example 1: Binary microprogram

- Only 1-bits marked (others are zero)

Address	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
1										1							1				1	
2				1										1	1		1					1
3					1				1								1					1
4																		1				1
5						1		1									1					
6																			1			1
7				1		1					1						1				1	
8	1							1	1								1					1
9								1										1				

← Control bits c1...c22

Clock stage 1 = c1...c8  
Clock stage 2 = c9...c14  
Clock stage 3 = c15...c16  
Clock stage 4 = c17...c22  
(Clock stage 5 contains no control bits.)

## Example 2: Multiplication

- ALU of our microprogrammed computer does not have multiplication operation
- So, we need to implement this in another way
- The simplest way to perform multiplication is to convert it to a summation:
  - Sum together multiplier pcs of multiplicands

$$5 \cdot 3 = \underbrace{3 + 3 + 3 + 3 + 3}_{5 \text{ pcs}}$$

- How could we write a microprogram algorithm that does this?

## Example 2: Multiplication

- Algorithm:

```
MDR := 0
WHILE A <> 0 DO
    MDR := MDR + C
    A := A - 1
ENDWHILE
```

- While-loop is not supported in microprograms, so it needs to be replaced with skip/jump commands

## Example 2: Multiplication

- Microprogram algorithm:

```
MDR := 0
1 : skip A=0      If skip condition is true      If skip condition is false
      jump 5 ←
      MDR := MDR+C ←
      A := A - 1 ; jump 1
5 :
```

- Binary program:

Address	1	2	3	4	5	6	7	8	9	0	1	1	1	1	1	5	6	1	1	1	2	2	2
0																							
1																							
2																							
3																							
4	1									1													

## Example 2: Algorithm efficiency

- The repetition phase of this algorithm comprises of 3 microinstructions (1,3,4)
- Our computer is 16-bit, so the maximum value of multiplier is  $2^{16} - 1$
- Therefore, the maximum number of microinstructions to be performed is approx.

$$(2^{16} - 1) \cdot 3 = 196\,605$$

- Each microinstruction takes 5 clock cycles, so the maximum number of clock cycles needed is about 1 million
- CPU's clock speed defines the elapsed time (for example, 10 MHz → 0.1 seconds)
- Can the multiplication be performed quicker by using a more clever algorithm?

## Example 2: Grade school multiplication algorithm

- Let's take a moment to remember how we performed multiplications without a calculator in elementary school:
  - $7 \times 3 = 21$ ; write 1 and carry the 2
  - $7 \times 5 = 35$ , add the carried 2 = 37; write 7 & carry 3
  - $7 \times 4 = 28$ , add the carried 3 = 31
  - Move one step left, start from tens:  $2 \times 3 = 6$
  - $2 \times 5 = 10$ ; write 0 and carry the 1
  - $2 \times 4 = 8$ ; add the carried 1 = 9
  - Move one step left, start from hundreds:  $1 \times 3 = 3$
  - $1 \times 5 = 5$
  - $1 \times 4 = 4$
  - Sum all results together

$$\begin{array}{r} 453 \\ \times 127 \\ \hline 3171 \\ 906 \\ +453 \\ \hline 57531 \end{array}$$

## Example 2: Grade school multiplication algorithm

- The same algorithm works for binary numbers, too!
  - Multiplicand is added to the sum 0 or 1 times
- Due to the limitations set by our computer, we have to perform some modifications:
  - Because only the most significant bit (1<sup>st</sup> bit on the left) can be examined separately, the partial sums (multiplicand \* multiplier) have to be formed in reverse order – from left to right
  - Partial sums are added to the total sum right after they've been formed (not mandatory, but saving partial sums to main memory kills performance advantages)
  - Because we're moving from left to right, the new partial sum should be moved right before adding it to the sum; we can't do this, so we move the old sum to the left instead (shiftleft)

$$\begin{array}{r} \text{Multiplicand} & 000101 \\ \times \text{ Multiplier} & 000110 \\ \hline & 000000 \\ & 000101 \\ & 000101 \\ & 000000 \\ & 000000 \\ + & 000000 \\ \hline & 00000011110 \end{array}$$

## Example 2: Grade school multiplication algorithm

- As a result of these modifications, the multiplication looks like this:
- Because the word size is 16 bits, this multiplication has to be done 16 times in our computer
- So, there will be 16 partial sums
- How to tell our computer when to stop the multiplication?
- The most natural way would be to use a counter which is formatted to have a value 16 in the beginning and then reduce it by 1 each round
- In microcode, we can't set values this easily, so we'll do this another way round: a fake 1 is added to the end of our multiplier
- For this last 1, the partial sum is not calculated

$$\begin{array}{r} 0101 \\ \times 0110 \\ \hline 0000 & 1^{\text{st}} \text{ partial sum} \\ + 0101 & 2^{\text{nd}} \text{ partial sum} \\ \hline 00101 \\ + 0101 & 3^{\text{rd}} \text{ partial sum} \\ \hline 001111 \\ + 0000 & 4^{\text{th}} \text{ partial sum} \\ \hline 0011110 & \text{Total sum} \end{array}$$

## Example 2: Grade school multiplication algorithm

```
MDR:=0
IF A < 0 THEN
    MDR:=MDR+C
ENDIF
shiftleft(A)
A:=A+1
loop: IF A < 0 THEN
    shiftleft(A)
    IF A = 0 THEN
        jump stop
    ELSE
        shiftleft(MDR)
        MDR:=MDR+C
        jump loop
    ENDIF
ELSE
    shiftleft(MDR)
    shiftleft(A)
    jump loop
ENDIF
stop:
```

(\*1<sup>st</sup> bit of A is 1)

(\*fake 1\*)
(\*multiplier bit is 1\*)
(\*move multiplier left\*)
(\*previous multiplier 1<sup>st</sup> bit was the last 1\*)
(\*because the last 1 is the fake 1\*)

(\*move the old sum left\*)
(\*perform addition of sum and partial sum C\*)

(\*multiplier bit is 0\*)
(\*move old sum left\*)
(\*move multiplier left\*)

## Example 2: Grade school multiplication, symbolic microprogram

Address	Microinstruction	Explanation
0	0+0 →MDR; ; (A<0) +MPC →MPC	Set MDR=0. Is 1 <sup>st</sup> bit of A 1?
1	MDR+C →MDR; ; 1+MPC →MPC	Yes; add C to MDR
2	(0+A) x2 →A; ; 1+MPC →MPC	Move A to left
3	1+A →A; ; 1+MPC →MPC	Add fake 1 as the last bit of A
4	; ; (A<0) +MPC →MPC	Is the 1 <sup>st</sup> bit of A 1?
5	; ; 1001+0 →MPC	Yes; jump to address 9
6	(MDR+0) x2 →MDR; ; 1+MPC →MPC	No; move MDR to left
7	(0+A) x2 →A; ; 1+MPC →MPC	Move A to left
8	; ; 100+0 →MPC	Jump to beginning of loop (4)
9	(0+A) x2 →A; ; (A=0) +MPC →MPC	Move A to left; is A=0?
10	; ; 1110+0 →MPC	Yes; stop (jump to address 14)
11	(MDR+0) x2 →MDR; ; 1+MPC →MPC	No; move MDR to left
12	MDR+C →MDR; ; 1+MPC →MPC	Add C to MDR
13	; ; 100+0 →MPC	Jump to beginning of loop (4)

## Example 2: Grade school multiplication, algorithm efficiency

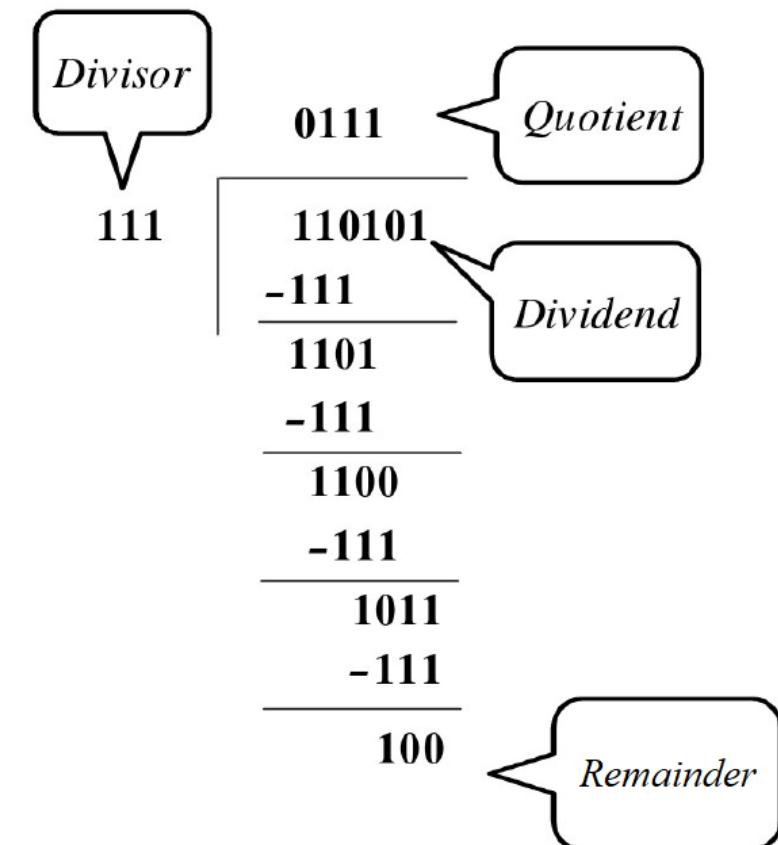
- The microprogram of 1<sup>st</sup> option was much simpler; why is this better?
- The repetition phase of this algorithm comprises of 4 (4,6,7,8) or 6 microinstructions (4,5,9,11,12,13)
- The initiation is 4 microinstructions (0,1,2,3) and the last repetition is 4 microinstructions (4,5,9,10)
- Because the computer only has to perform 16 multiplications, this multiplication algorithm needs in total  $4+15*6+4 = 98$  microinstructions
  - Quite a lot less than the original maximum of 196 605!
- Quick calculation tells us that this algorithm can be 2000 times faster
- Also, the time needed to complete the multiplication is not much dependent on the magnitude of multiplier (unlike the previous algorithm)

# Division of binary numbers

- Division of binary numbers can be implemented in a similar fashion as division of decimal numbers
- The method is called “shift-and-subtract”
- More efficient ways have also been invented
  - Fast division algorithm (and its variations)

Binary division:

1. Align the divisor Y with the most significant end of the dividend. Let the portion of the dividend from its MSB to its bit aligned with the LSB of the divisor be denoted X.
2. Compare X and Y.
  - a) If  $X \geq Y$ , the quotient bit is 1 and perform the subtraction  $X-Y$ .
  - b) If  $X < Y$ , the quotient bit is 0 and do not perform any subtractions.
3. Shift Y one bit to the right and go to step 2.



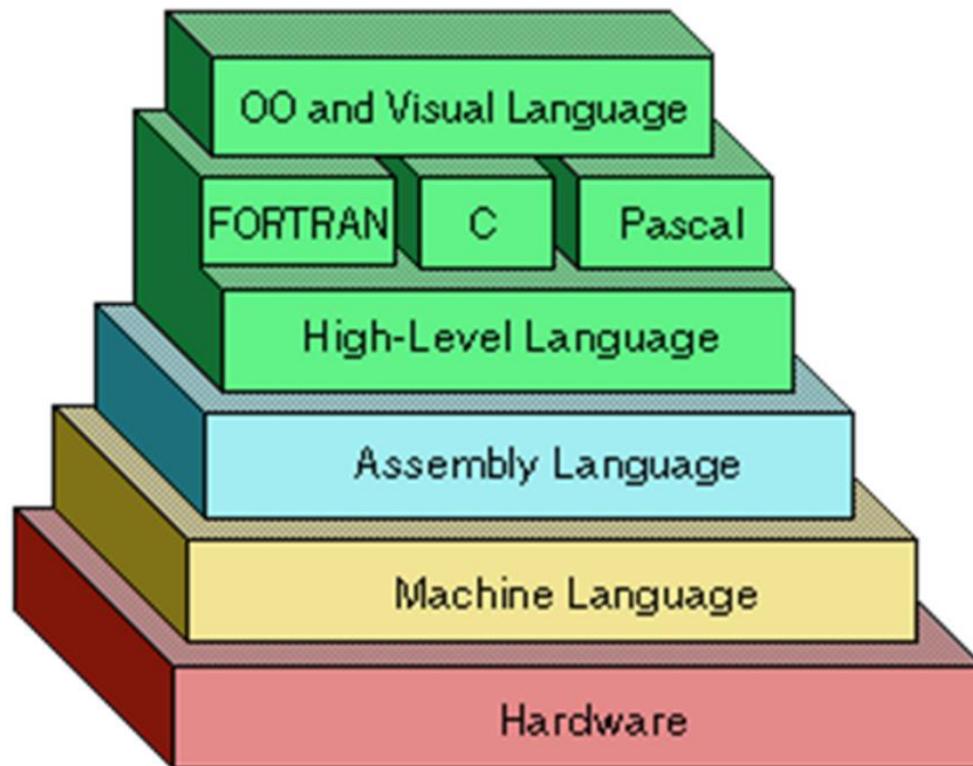
# Summary

- Information that has been stored in the memory of a micro-programmed computer affects the operation of the computer, so it is capable of executing different algorithms
- The structure (logic circuits made of logic gates) of the micro-programmed computer defines the operations that the computer includes; all other functions must be performed via microprograms
- Presentation methods of microprograms:
  - Pseudocode
  - Symbolic microprogram
  - Binary microprogram

# Thank you for listening!



# 4. Machine-language programming



## Levels of Programming Languages

High-level program

```
class Triangle {  
    ...  
    float surface()  
    return b*h/2;  
}
```

Low-level program

```
LOAD r1,b  
LOAD r2,h  
MUL r1,r2  
DIV r1,#2  
RET
```

Executable Machine code

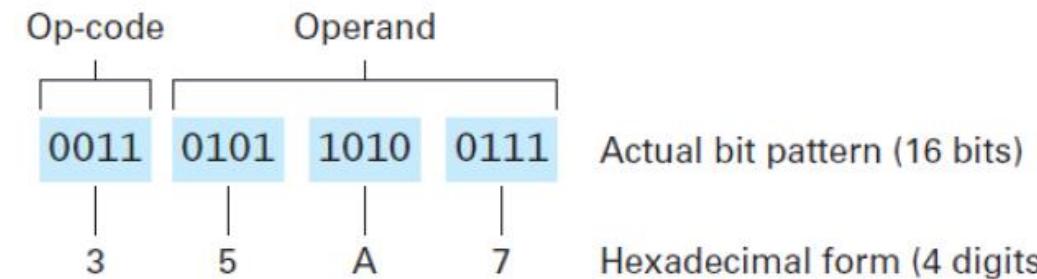
```
0001001001000101  
0010010011101100  
10101101001...
```

# Machine language

- Writing programs in microcode is very time-consuming:
  - Performing even simple tasks (e.g. multiplication) requires multiple microinstructions
- Machine language provides a (little) bit more user-friendly way
  - One step higher than microcode in abstraction level
  - Still quite far removed from higher level languages (C, Python)
  - No need to care about what happens in which clock cycle
- In order to understand machine language programs, microprogrammed computers need an interpreter
  - Interpreter reads the program and translates it “on the fly” to microcode
  - Interpreter program is written on microcode and stored in MPM

# Symbolic machine language

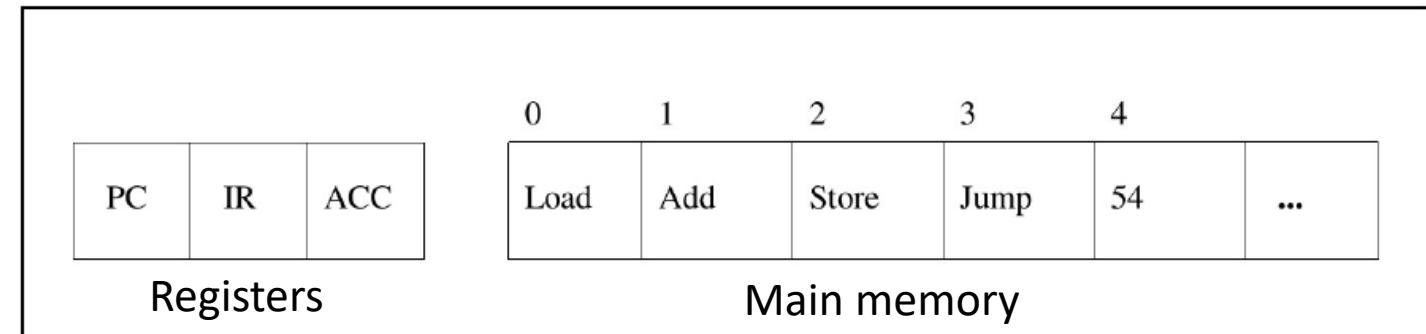
- True machine code is raw binary data, that consists of an operation part and an operand part (as we learned before)
  - The binary code can be shortened to octal or hexadecimal form to improve readability



- The same code can be represented in symbolic machine language:
  - Operation codes are replaced by (more descriptive) operation names
  - Operands are given in decimal form
  - Further improved readability

# Machine language example

- As the name implies, machine languages are “tied” to the machine: they are processor-specific (not necessarily in general, but at least in detail level)
- Let’s investigate how machine language works via an example computer that has the following features:
- Special registers
  - PC (program counter; similar to MPC+MPM in microprogrammed computer)
  - IR (instruction register; similar to MIR in microprogrammed computer)
  - ACC (accumulator; includes the data that is currently in process)
- Main memory (RAM)
  - Machine language programs
  - Data to be handled



# Machine language example: operations

- Our example computers machine language includes the following commands
- This language uses single operands: one is given, the other one (if needed) is always the accumulator (ACC)
  - Two- or even three-operand languages exist, though

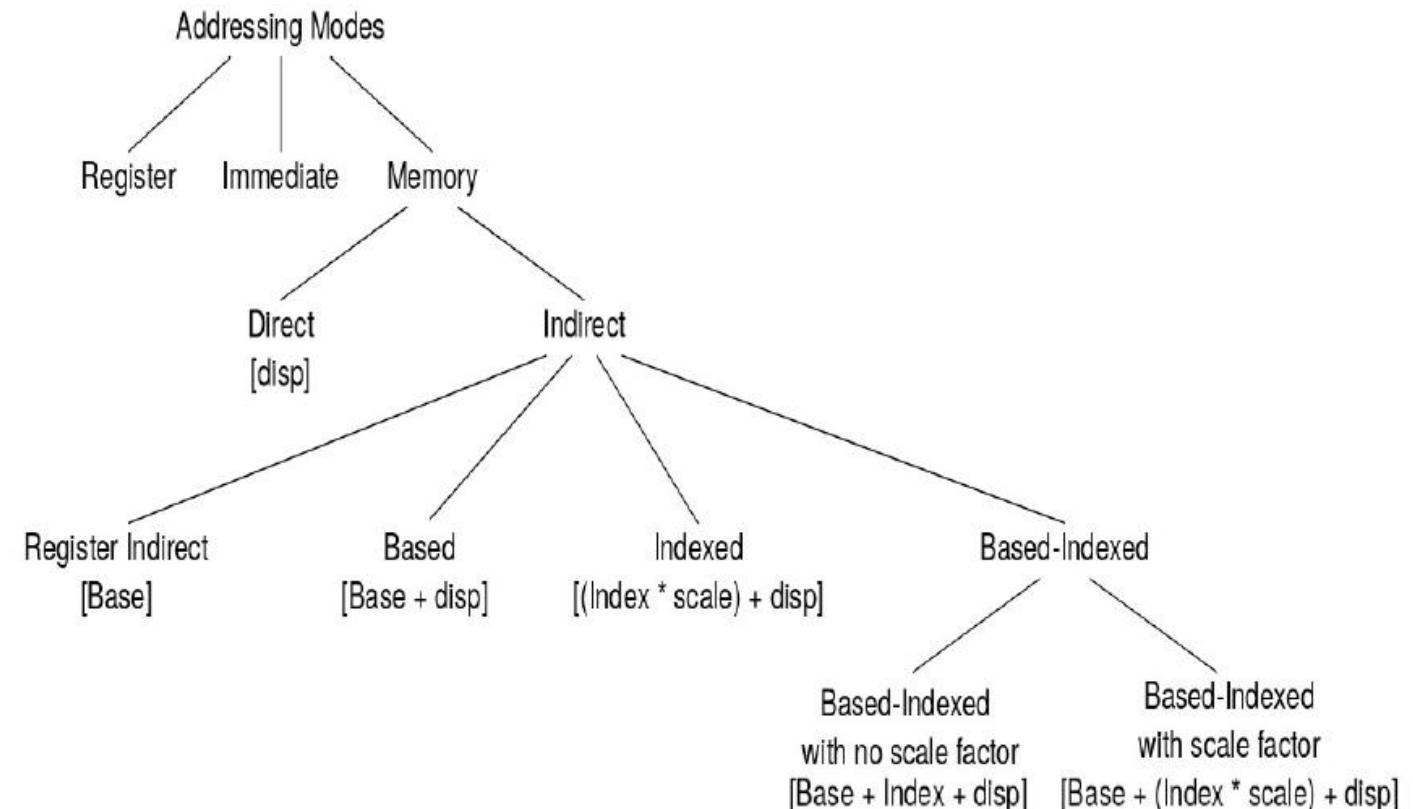
Symbolic command	Action
LOAD M	$ACC \leftarrow (M)$
STORE M	$(M) \leftarrow ACC$
ADD M	$ACC \leftarrow ACC + (M)$
SUBTRACT M	$ACC \leftarrow ACC - (M)$
MULTIPLY M	$ACC \leftarrow ACC * (M)$
DIVIDE M	$ACC \leftarrow ACC / (M)$
JUMP M	Jump to M
JUMPZERO M	Jump to M, if $ACC = 0$
JUMPNEG M	Jump to M, if $ACC < 0$
JUMPSUB M	Jump to subprogram that starts from M
RETURN M	Return from subprogram that started from M

# Machine language example: features

- Commands are stored in main memory as binary code
- Commands are executed one at a time in given order, unless the order is specifically changed (for example via JUMP commands)
- Word size 16 bits, memory address 12 bits (= 4096 different addresses)
  - 4 bits left for operation code, so 16 different operations possible (in theory)
- In our example language, constant numbers are not available
  - All numbers that we use in calculation must be stored in main memory
- Language can be expanded to cover also “immediate addressing” with an operation called LOADI
  - This way numbers can be given as operands

# Addressing modes

- Addressing mode means the way how the operand part of the command defines what the true operand is
- Possibilities:
  - Immediate
  - Direct
  - Indirect
    - Indirect indexed
    - Indirect based-indexed



# Addressing modes

- Immediate addressing:
  - Operand specifies the data that will be loaded to ACC
  - $\text{LOADI } M \Rightarrow \text{ACC} \leftarrow M$
- Direct addressing:
  - Operand specifies the memory address of the data that will be loaded to ACC
  - $\text{LOAD } M \Rightarrow \text{ACC} \leftarrow (M)$
- Indirect addressing:
  - Operand specifies the memory address that contains the memory address of the data that will be loaded to ACC
  - $\text{LOADID } M \Rightarrow \text{ACC} \leftarrow ((M))$

# Addressing modes

- In some cases, the data that we want to use may change depending on which stage of program execution we're in
- We can use indexed addressing to take this into account
- Indexed addressing:
  - The sum of operand and index register value gives the memory address of the data that will be loaded to ACC
  - $\text{LOADIX } M \Rightarrow \text{ACC} \leftarrow (\text{IR} + M)$
- Indirect indexed addressing:
  - The sum of value found in memory address given by the operand and index register value gives the memory address of the data that will be loaded to ACC
  - $\text{LOADIDX } M \Rightarrow \text{ACC} \leftarrow (\text{IR} + (M))$

# Addressing modes: example

- State of memory & IR are following:

	Index register value								4
Address	280	281	282	283	284	285	286	...	
Content	282	87	13	27	16	66	77	...	

- What are the ACC values after following commands?

Addressing mode	Command	ACC value after command
Immediate	LOADI 280	
Direct	LOAD 280	
Indirect	LOADID 280	
Indexed	LOADIX 280	
Indirect indexed	LOADIDX 280	

# Addressing modes: example

- State of memory & IR are following:

	Index register value								4
Address	280	281	282	283	284	285	286	...	
Content	282	87	13	27	16	66	77	...	

- What are the ACC values after following commands?

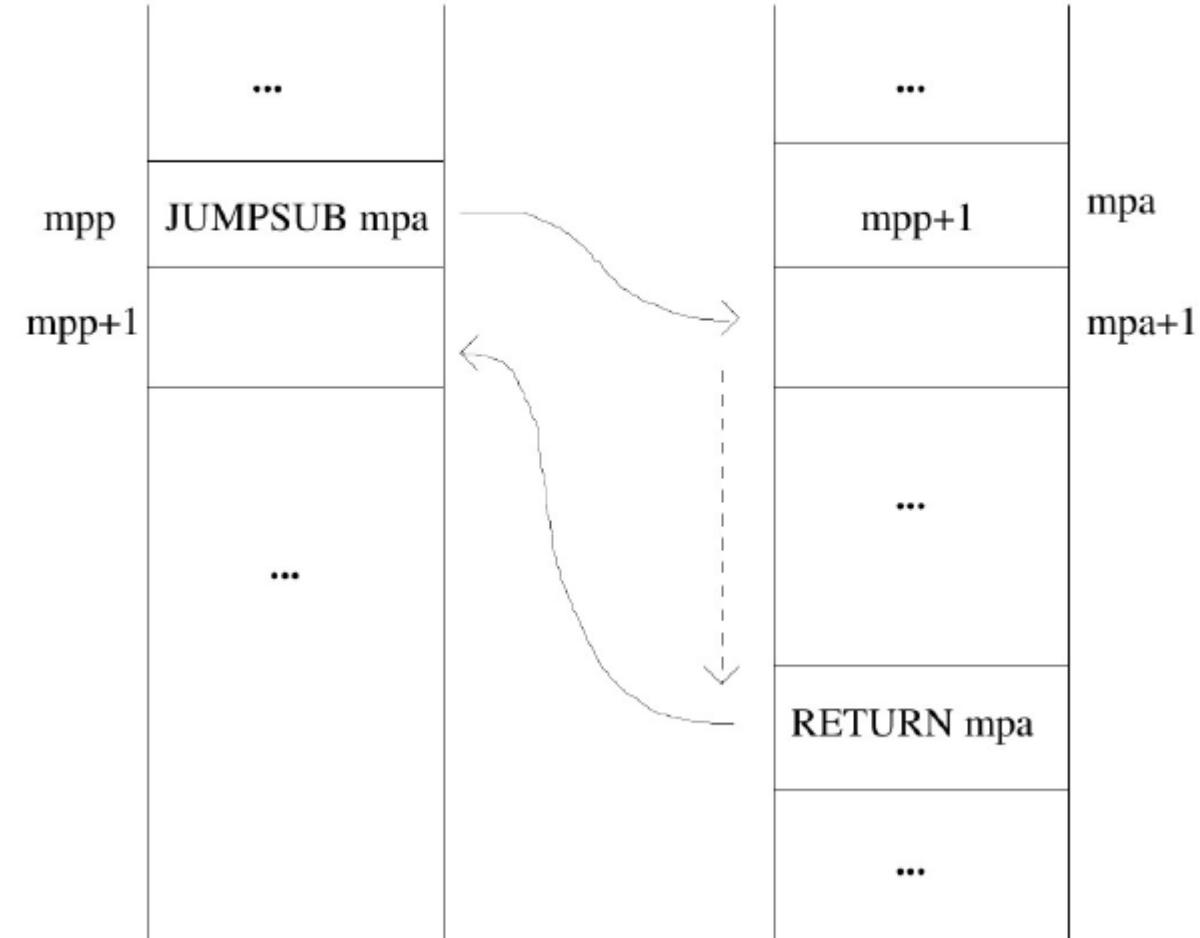
Addressing mode	Command	ACC value after command	
Immediate	LOADI 280	280	
Direct	LOAD 280	282	(280) → 282
Indirect	LOADID 280	13	((280)) → (282) → 13
Indexed	LOADIX 280	16	(280+4) → (284) → 16
Indirect indexed	LOADIDX 280	77	((280)+4) → (282+4) → 77

# Execution order of commands

- Execution order of commands can be changed by jumps
- Unconditional jump: JUMP M
  - Jumps to memory address M, no matter what
- Conditional jumps:
  - JUMPZERO M – jumps to memory address M, if ACC = 0
  - JUMPNEG M – jumps to memory address M, if ACC < 0
- Jump to subprogram and back:
  - JUMPSUB M – jumps to address M, where the subprogram begins
  - RETURN M – returns from subprogram that started from memory address M

# Jumps to subprograms

- When execution proceeds to  $mpp$ , we jump to subprogram that starts from  $mpa$
- Execution continues from  $mpa+1$
- After the subprogram has been executed, final RETURN tells where it started from ( $mpa$ )
- From this address we find the information, where we should return in the original program ( $mpp+1$ )
- Same subprogram can be called multiple times



## Example: Two to the power of n

- Let's write a program that calculates the value of two to the power of n
- First, a pseudo-code:

```
MODULE exp(n) RETURNS 2^n
    value := 1
    WHILE n > 0 DO
        n := n - 1
        value := value + value
    ENDWHILE
    RETURN value
ENDMODULE
```

Idea: output value starts from 1 and  
is doubled (= value + value) n times

Why in this order?

# Example: Two to the power of n

- Then the program in symbolic machine language:

```
MODULE exp(n) RETURNS 2^n
  value := 1
  WHILE n > 0 DO
    n := n - 1
    value := value + value
  ENDWHILE
  RETURN value
ENDMODULE
```

Now we see the reason for “strange” order; n was already in the ACC, so changing it now means not having to load it again later!

Memory address	Command	Explanation
371	LOAD 383	Load 1 to ACC
372	STORE 382	Store 1 as initial value of F
373	LOAD 381	Load n to ACC
374	JUMPZERO 384	If n = 0, jump to 384 (aka. Stop)
375	SUBTRACT 383	Subtract 1 from n
376	STORE 381	Save new value of n
377	LOAD 382	Load function value F to ACC
378	ADD 382	ACC = F + F
379	STORE 382	Save new value of F (F = ACC)
380	JUMP 373	Jump to beginning of iteration
381	n	Parameter
382	0	Function value F
383	1	Number constant

## Example: $2^x + 2^y$ using subprogram

- If we want to calculate what is  $2^x + 2^y$ , a natural solution would be to use the previous program twice (1st as  $n = x$  and 2nd time as  $n = y$ )
- For educational purposes, let's see how this could be implemented using a subprogram!
- First the pseudo-code:

```
MODULE sum(x,y) RETURNS 2^x + 2^y
    RETURN exp(x)+exp(y)
ENDMODULE
```

# Example: $2^x + 2^y$ using subprogram

- Main program on the left, subprogram + data on the right

Address	Command	Explanation
287	LOAD 314	Load x to ACC
288	STORE 311	Save x as subprogram input
289	JUMPSUB 299	Execute subprogram (i.e. calc. $2^x$ )
290	LOAD 312	Load $2^x$ to ACC
291	STORE 316	Save $2^x$ as end result value
292	LOAD 315	Load y to ACC
293	STORE 311	Save y as subprogram input
294	JUMPSUB 299	Execute subprogram (i.e. calc. $2^y$ )
295	LOAD 312	Load $2^y$ to ACC
296	ADD 316	Add $2^x$ to ACC value ( $2^y$ )
297	STORE 316	Store the end result
298	JUMP 317	Stop program execution

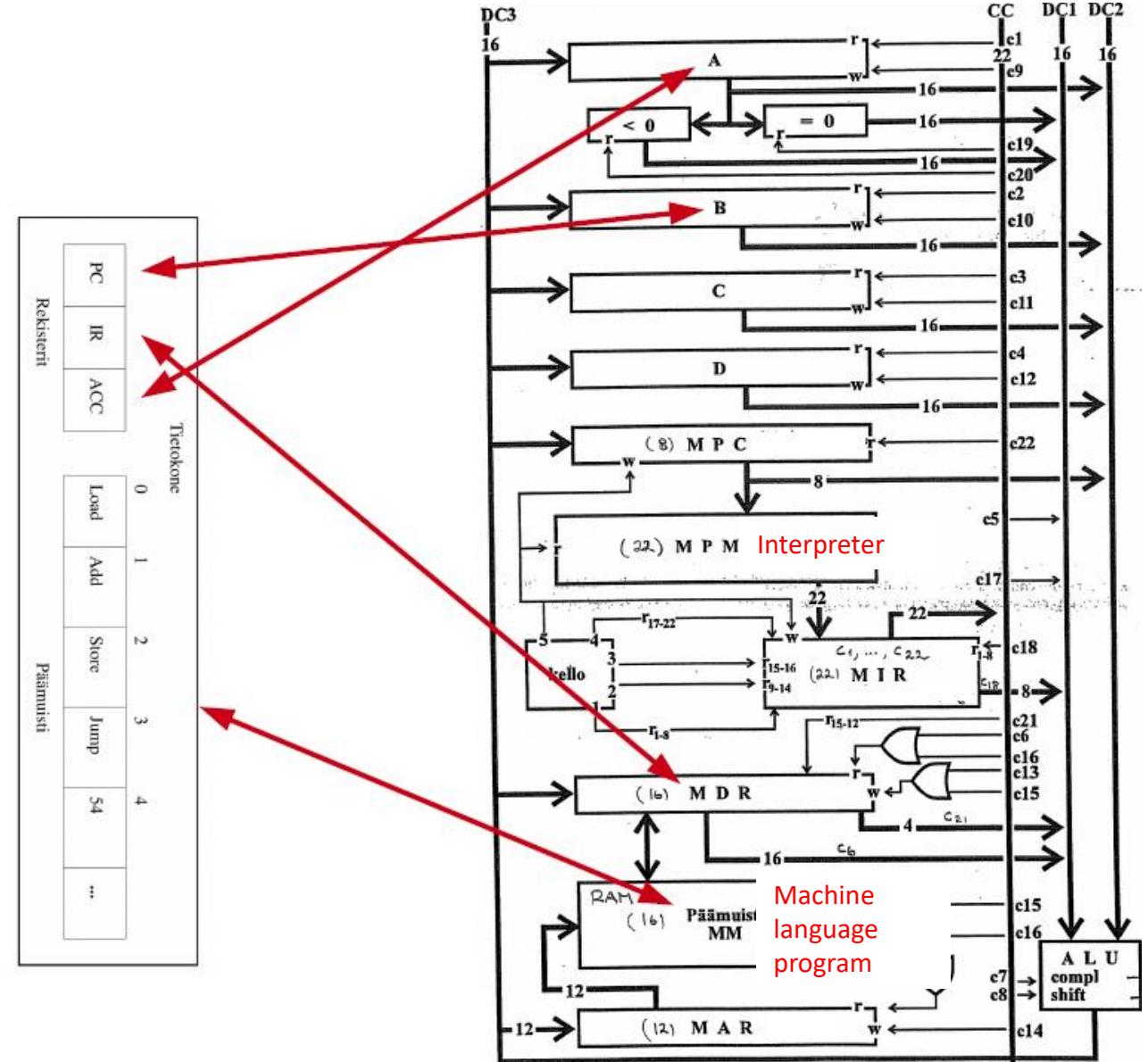
Address	Command	Explanation
299	0 / 290 / 295	Begin / Return1 / Return2
300	LOAD 313	Load 1 to ACC
301	STORE 312	Save 1 as subprogram result w
302	LOAD 311	Load subprogram input to ACC
303	JUMPZERO 310	If n = 0, jump to end of subpr.
304	SUBTRACT 313	Subtract 1 from n
305	STORE 311	Save new n value
306	LOAD 312	Load subprogram result w
307	ADD 312	ACC = w + w
308	STORE 312	w = ACC
309	JUMP 302	Jump back to iteration start
310	RETURN 299	Return to main program
311	0	Subprogram input n
312	0	Subprogram result w
313	1	Constant value
314	x	Main program input x
315	y	Main program input y
316	0	End result

# Machine language interpreter for our microprogrammed computer

- Interpreter is a microprogram, which understands and executes machine-language commands
- Operating principle:
  - Fetch the command from main memory
  - Find out the contents of the command
  - Execute the command
- In order to complete this task, the interpreter must remember the address of the next command
- From our example machine language to example microprogrammed computer:
  - ACC = Register A
  - IR = MDR as command register (4 most significant bits for the command)
  - PC = Register B as program counter (we could use C or D too, though)

# Machine language interpreter for our microprogrammed computer

- Graph illustrates the relationships between the registers



# Summary

- Machine language is processor- or computer-specific programming language
  - Can be presented in either symbolic or numerical (binary, hexadecimal...) form
- Several symbolic machine languages can be implemented for the same processor
- The machine language of “real-world” computers is more complex than of our example computer
  - Larger word size, more operations
- The interpreter program has been stored in read-only memory (MPM) beginning from memory address 0
  - Interpreter program execution is started by setting MPC to zero
- Programs written on high-level languages will be converted to machine language via a compiler (or an interpreter) and the machine language program will be executed using the microcode interpreter

# Thank you for listening!



# 5. From high-level language to machine language



# Assembly languages

- Machine language is raw number language in binary form (or, in shorter notations, hexadecimal form), so it's not very readable
- In the previous lecture we used symbolic machine language
- More advanced symbolic machine languages are actually called assembly languages
  - Key improvements include i.e. labels for program and memory locations & expressions (no need to think about memory addresses, calculations can be done with one command...)
  - Code written using these can be transformed to binary machine code using an assembler
- Assembly languages have words that specify the operations better than numerical op-codes, but these words still correspond to operations that are performed by the computer's physical components
- Still quite low level of abstraction

# High-level languages

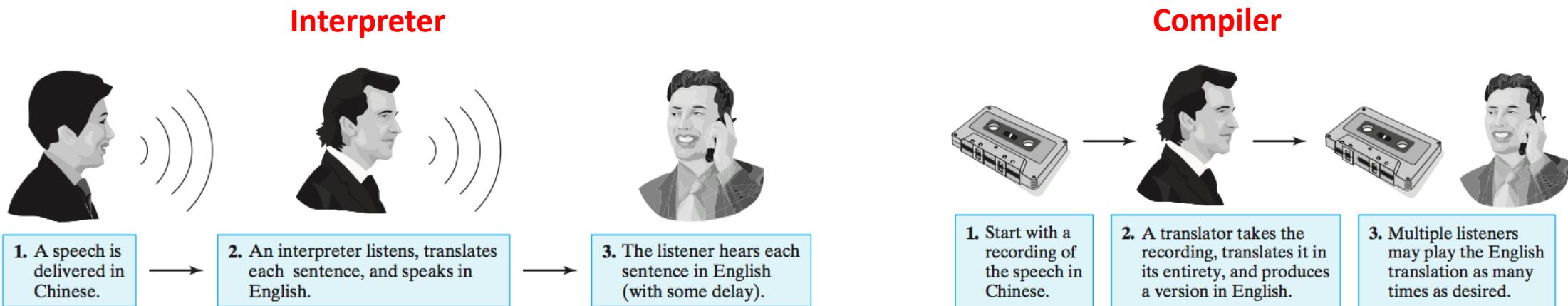
- On order to make programming easier, higher-level programming languages have been developed – first ones already in the late 1950s (FORTRAN, LISP)
- More advanced ones soon followed (C/C++, Python)
- High-level languages contain advanced control- and data structures
  - No need for primitive JUMP-commands
  - No need to think about memory addresses or contents of the accumulator
  - Emphasis on logic ("what happens") instead of execution details ("how it happens")
- Programming is easier, programs are shorter and easier to debug & modify
- Programs written on high-level languages are portable to nearly any device
- ...so, have high-level languages made lower ones obsolete?

# High-level vs. low-level languages

- Short answer is no
- Lower-level programs require less memory and execute quicker
- Also, lower-level programs are capable of making better use of the resources available (memory, registers, processor use)
- Therefore, if the programs needed are simple but require
  - As fast execution time as possible
  - Low energy consumption
- ...it's good to consider a lower-level program
  - For example: Automotive applications (ECU, ESP system), wearables (sports watches etc.)
- Otherwise, high-level languages are strongly favorable

# Compiler vs. interpreter

- A program written on high-level language must be translated to machine code in order to execute it
- This translation can be done in two ways: using a compiler or by using an interpreter
  - The difference can be understood by using an analogy: translating a speech from Chinese to English



# Compiler vs. interpreter

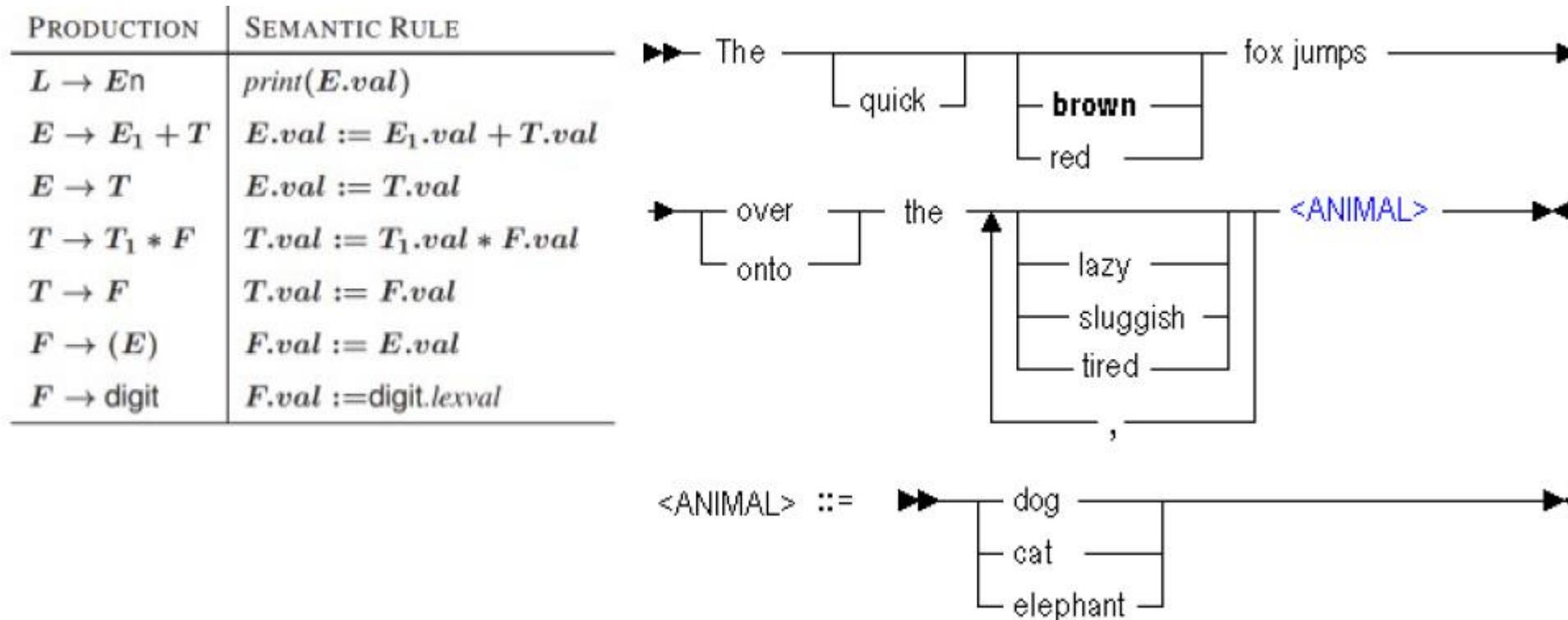
- Both have their advantages and disadvantages
- Interpreter:
  - + Translation is (almost) immediate
  - Translation is not stored anywhere for further use
- Compiler:
  - + Translation must be done only once; after it's done, the translation can be listened to multiple times by multiple people
  - Translation takes some time to complete
- Interpreter is a good choice in development phase, when it's important to test and debug the program quickly
- When the program is ready, compiling it is a good idea due to increased speed
  - Also, end user doesn't need an interpreter to run the program; it runs as a stand-alone
- Interpreters are commonly used in web pages (JavaScript etc.)

# Syntax and grammar

- In order to function automatically, the translation process requires a well defined set of rules for converting a long string of characters to a program
- This set of rules is called the syntax of the language
- The syntax is defined by grammar (like with natural languages)
- Grammar rules are called productions, which specify which kind of characters & character strings are included in the language
  - If some character string is not included in the language, the translation process is halted
- We'll take a closer look at these a bit later...

# Syntax and grammar

- Grammar of programming languages is not that different from natural languages
  - Definitions are more accurate, though
  - Also, there are no exceptions to rules

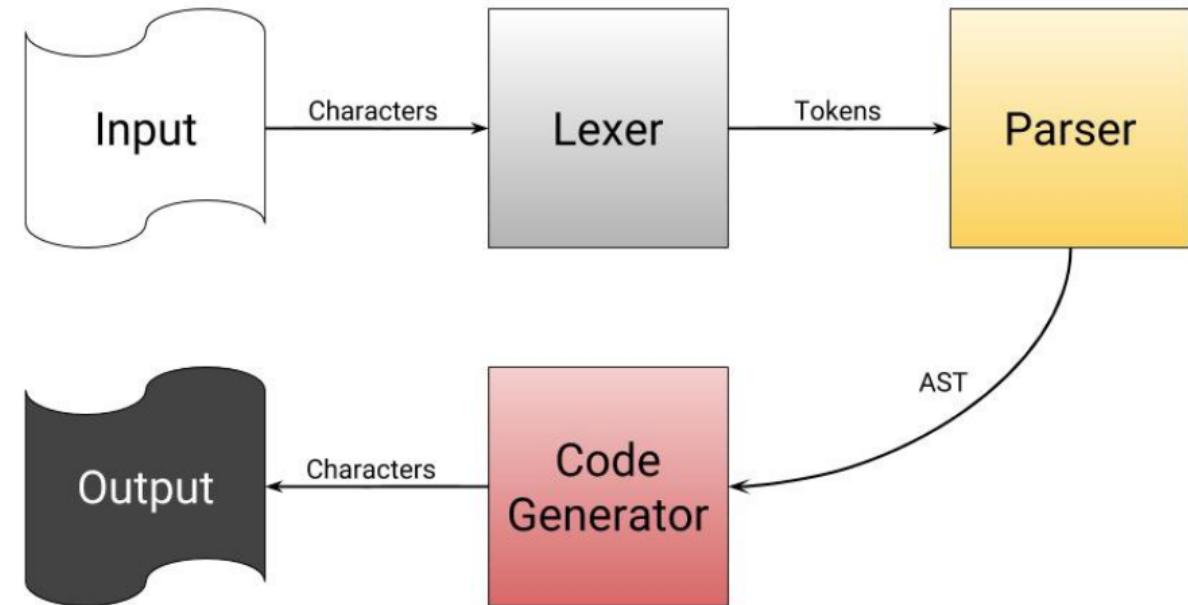


# Context-free grammar

- The syntax of programming languages is often described using context-free grammar (also known as Backus-Naur-Form grammar; BNF)
- This means that the structuring options of components is not dependent on their surroundings
  - Natural languages are not context-free; for example, English word "lead" can be interpreted in at least three possible ways (noun, verb, adjective ("lead singer"))
- Context-free grammar is comprised of four elements:
  - Terminals
  - Nonterminals
  - Start symbol (actually a nonterminal, but a special case)
  - Production rules
- In BNF, nonterminals only appear singly on the left sides of productions

# Compiling process

- A compiler translates the high-level language program (input) to machine level language (output) in three stages:
  - Lexing (lexical analysis)
  - Parsing (syntactic analysis)
  - Code generation
- Optimization of output is possible



# Lexing

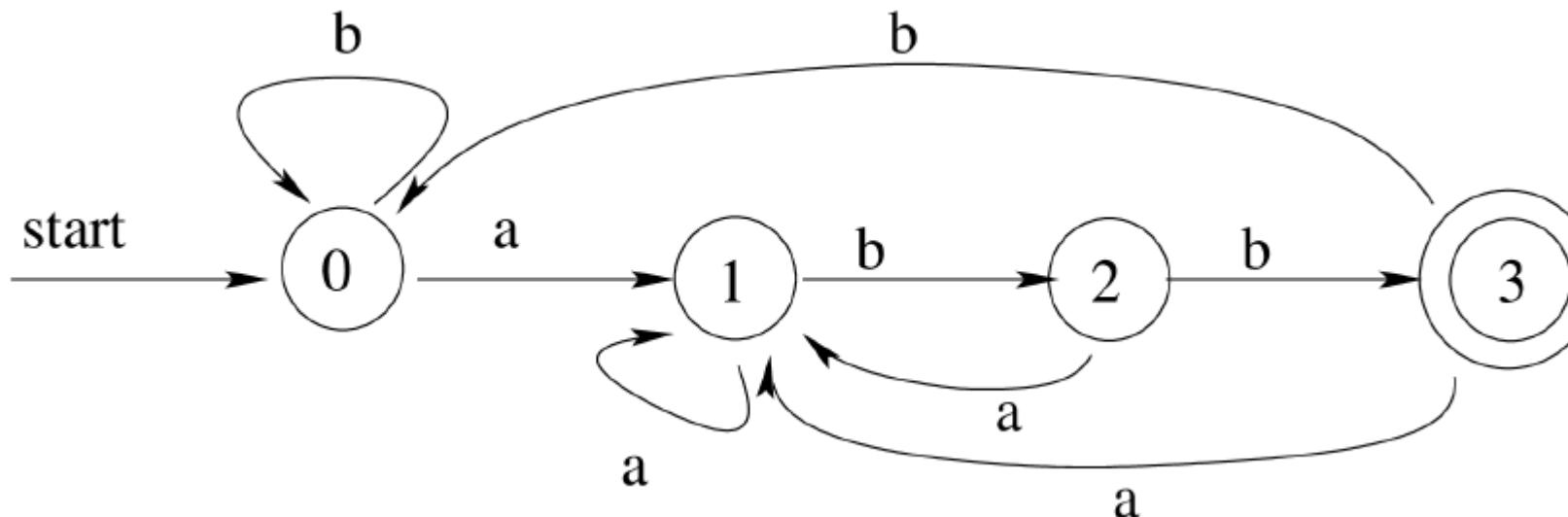
- Lexing identifies the characters that belong to a single text element
- These text elements are called tokens, which can be categorized as follows:
  - Keywords (IF, WHILE, FOR, END)
  - Operators ( +, -, \*, < )
  - Separators ( [ , ; ) : )
  - Identifiers (x, y, val)
  - Literals (7, 523, pi)
  - Comments (optional)
- These tokens represent the terminals
- Identifiers are stored in a symbol table

IF x < 5 THEN x := x + 1  
K I O L K I O I O L

# State machines

- Lexer is basically a state machine that returns the identified tokens
- State machine can be defined using regular expressions
  - State machine below identifies, for example, inputs  $(a \mid b)^*abb$
  - Try inputs i) ababb ii) abbbaabb iii) abbab. Which ones of these it identifies as tokens?

This means "a or b, repeated 0 to infinite number of times"!

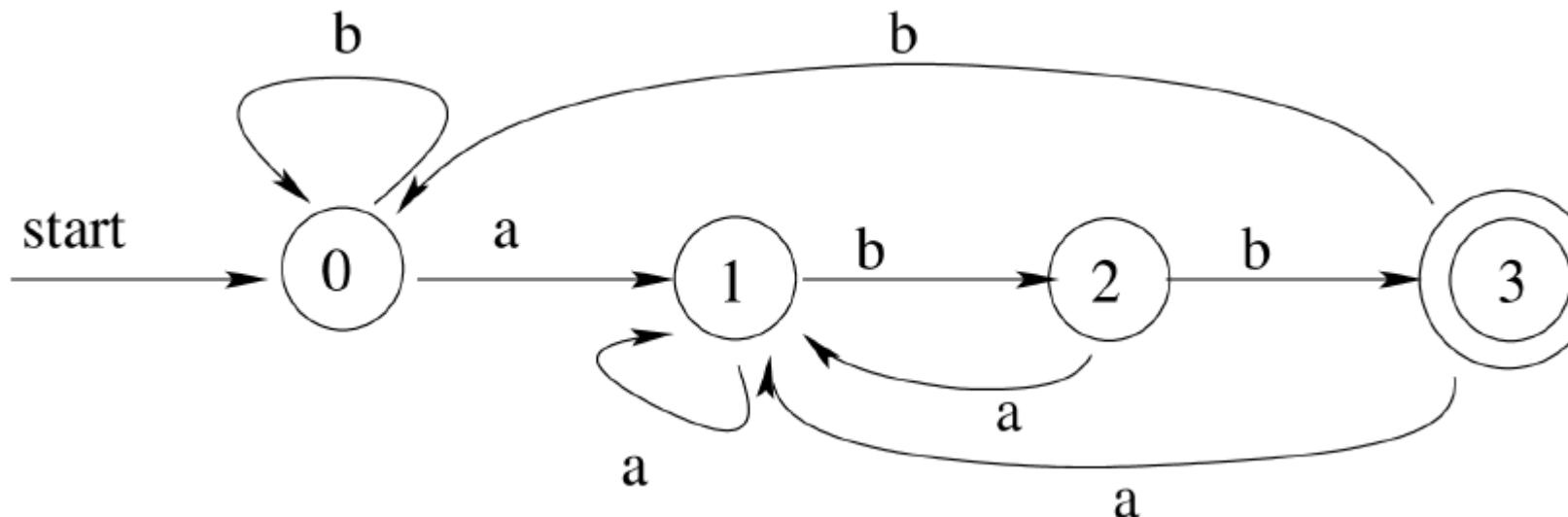


Inputs that end in double circled dots are identified!

# State machines

- Lexer is basically a state machine that returns the identified tokens
- State machine can be defined using regular expressions
  - State machine below identifies, for example, inputs  $(a \mid b)^*abb$
  - Try inputs i) ababb ii) abbbaabb iii) abbab. Which ones of these it identifies as tokens?
    - Inputs i and ii, but not iii!

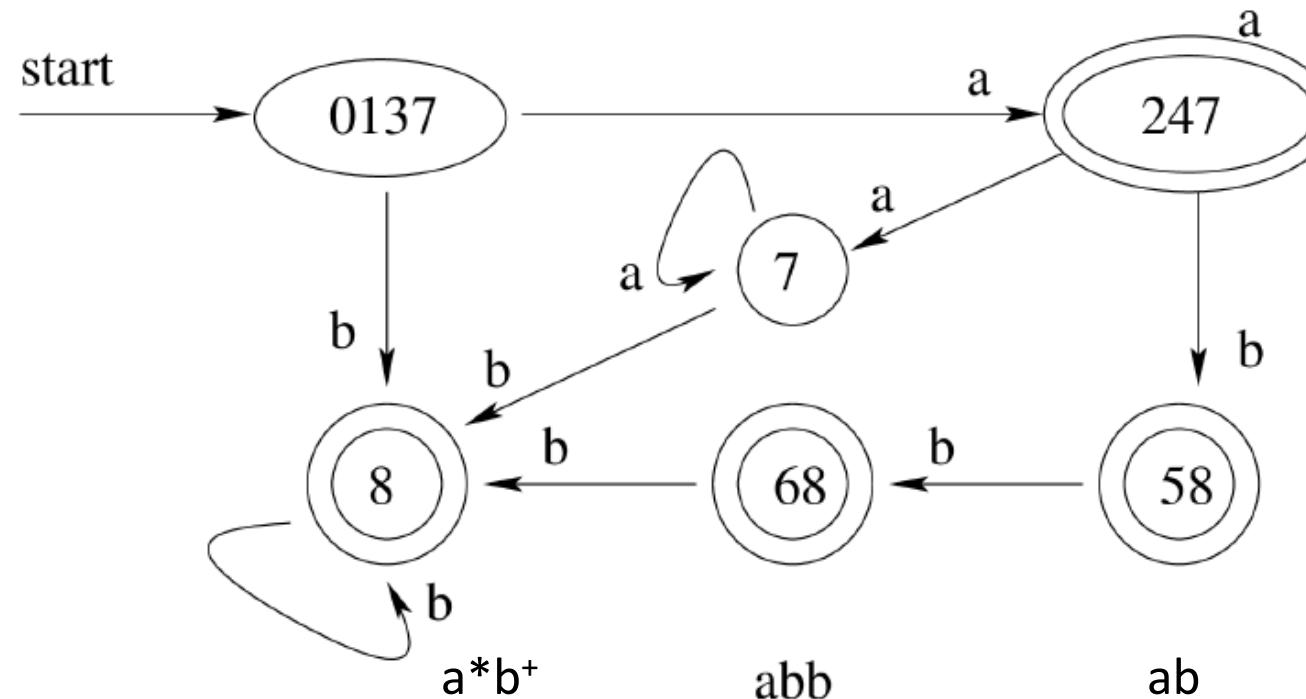
This means "a or b, repeated 0 to infinite number of times"!



Inputs that end in double circled dots are identified!

# State machines

- A state machine can identify multiple different tokens
- For example, the one below identifies tokens of forms a and  $a^*b^+$ 
  - State 247 identifies a, state 58 ab, state 68 abb and state 8 all others



$a^* = 0 \text{ to infinite}$   
pcs of a's  
 $b^+ = 1 \text{ to infinite}$   
pcs of b's

# Parsing

- Lexing produces us a list of identified tokens
- The idea of parsing is to find out the syntactic structure of tokens
- The result is usually shown as a parse tree
  - Start symbol is at the root of the tree
  - Leaf nodes are terminals
  - Interior nodes are non-terminals
  - Resulting sentence is read from left to right
- If the grammar of the language is properly defined, the resulting parse tree should be unambiguous

# Parsing

- There are two possible strategies for parsing:
  - Top-down (from root to leaves)
  - Bottom-up (from leaves to the root)
- Top-down parsing is a logical choice
  - Grammar productions are used in order to modify the root
  - If the root can be modified to match the input, the parse tree can be formed
- Bottom-up parsing goes the other way
  - Grammar productions are used in order to modify the input
  - If the input can be modified to match the root, the parse tree can be formed
- Parsing process (especially bottom-up) is quite nondeterministic, if the precedence of productions is not defined in the grammar
  - Need to back off from “bad guesses”

# Parse tree

- Example:  
grammar is  
defined as

“id” is a  
terminal, E  
acts as a start  
symbol as  
well as a  
non-terminal

$$E \rightarrow E+E \mid E^*E \mid -E \mid (E) \mid id$$

# Parse tree

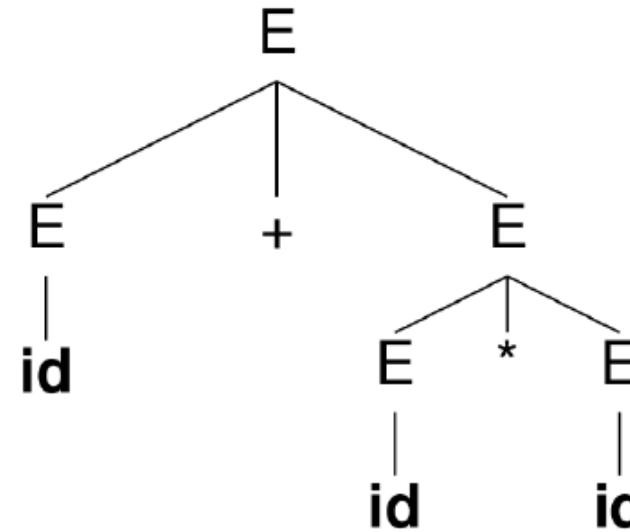
- Example: grammar is defined as

**“id” is a terminal, E acts as a start symbol as well as a non-terminal**

$$E \rightarrow E+E \mid E^*E \mid -E \mid (E) \mid id$$

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E^*E \\ &\Rightarrow id + id^*E \\ &\Rightarrow id + id^*id \end{aligned}$$

- Parse tree from input id+id\*id:



# Parse tree

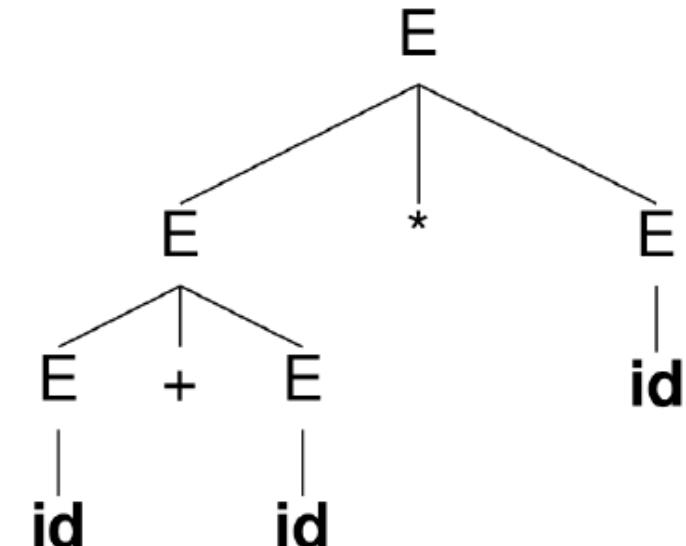
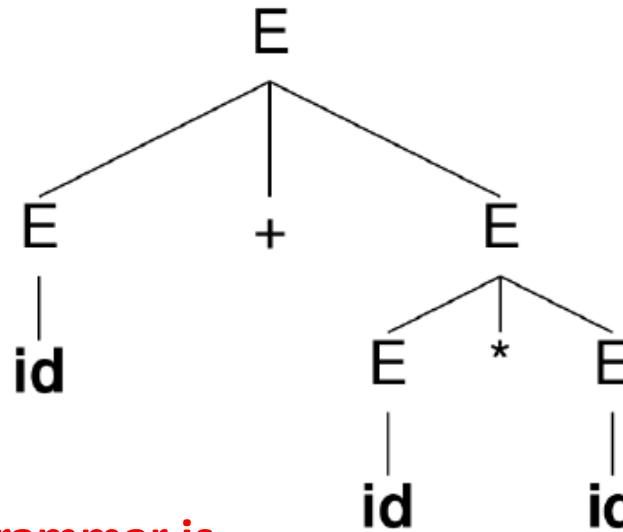
- Example: grammar is defined as

$$E \rightarrow E+E \mid E^*E \mid -E \mid (E) \mid id$$

"id" is a terminal, E acts as a start symbol as well as a non-terminal

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow id + E \\ &\Rightarrow id + E^* E \\ &\Rightarrow id + id^* E \\ &\Rightarrow id + id^* id \end{aligned}$$

$$\begin{aligned} E &\Rightarrow E^* E \\ &\Rightarrow E + E^* E \\ &\Rightarrow id + E^* E \\ &\Rightarrow id + id^* E \\ &\Rightarrow id + id^* id \end{aligned}$$



Two different parse trees! So, the grammar is poorly defined and parsing will result in an error.

# Parse tree

- The previous example was top-down; let's use bottom-up now for a change
- In the previous example, the problem was that all the productions were at same level; let's try a grammar which has more non-terminals
- Can we now make a parse tree out of, say, input  $\text{id}^*\text{id}$ ?

$$\begin{aligned} E &\rightarrow E+T \mid T \\ T &\rightarrow T^*F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

# Parse tree

- The previous example was top-down; let's use bottom-up now for a change
- In the previous example, the problem was that all the productions were at same level; let's try a grammar which has more non-terminals
- Can we now make a parse tree out of, say, input  $\text{id}^*\text{id}$ ?
  - YES!

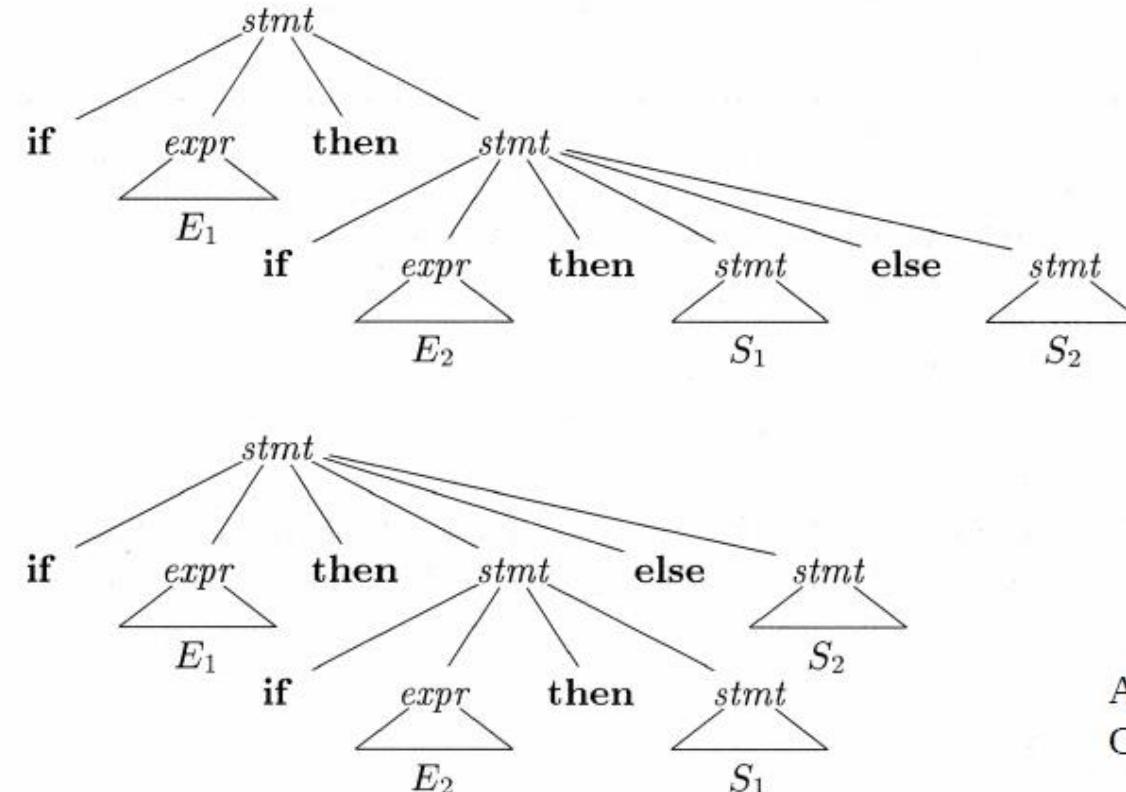
$$\begin{array}{ccccccccccc} \text{id}^*\text{id} & \rightarrow & F^* \text{id} & \rightarrow & T^* \text{id} & \rightarrow & T^* F & \rightarrow & T & \rightarrow & E \\ \text{id} & & F & & F & & id & & T^*F & & T \\ & & id & & id & & & & F & & id \\ & & & & & & & & id & & T^*F \\ & & & & & & & & & & F & id \\ & & & & & & & & & & id & id \end{array}$$

Old (replaced) productions  
shown below.

## Parse tree

**if E<sub>1</sub> then if E<sub>2</sub> then S<sub>1</sub> else S<sub>2</sub>**

- Especially a case of multiple ifs can result in ambiguous parse trees
- Is the last “else” associated with E<sub>1</sub> or E<sub>2</sub>?
- Grammar rules are needed to make the distinction!
  - Brackets
  - Tabulations



Aho, Lam, Sethi, Ullman:  
Compilers. Pearson, 2007.

Figure 4.9: Two parse trees for an ambiguous sentence

# From parsing table to parse tree

- Grammar rules can be expressed in the form of a parsing table
  - During this course, we will not consider how to formulate a parsing table from grammar; we will only learn how to use the table in order to build a parse tree
  - Example grammar:
- |      |               |                      |      |               |                      |
|------|---------------|----------------------|------|---------------|----------------------|
| $E$  | $\rightarrow$ | $TE'$                | $T'$ | $\rightarrow$ | $*FT' \mid \epsilon$ |
| $E'$ | $\rightarrow$ | $+TE' \mid \epsilon$ | $F$  | $\rightarrow$ | $(E) \mid \text{id}$ |
| $T$  | $\rightarrow$ | $FT'$                |      |               |                      |

- Parsing table (top-down):

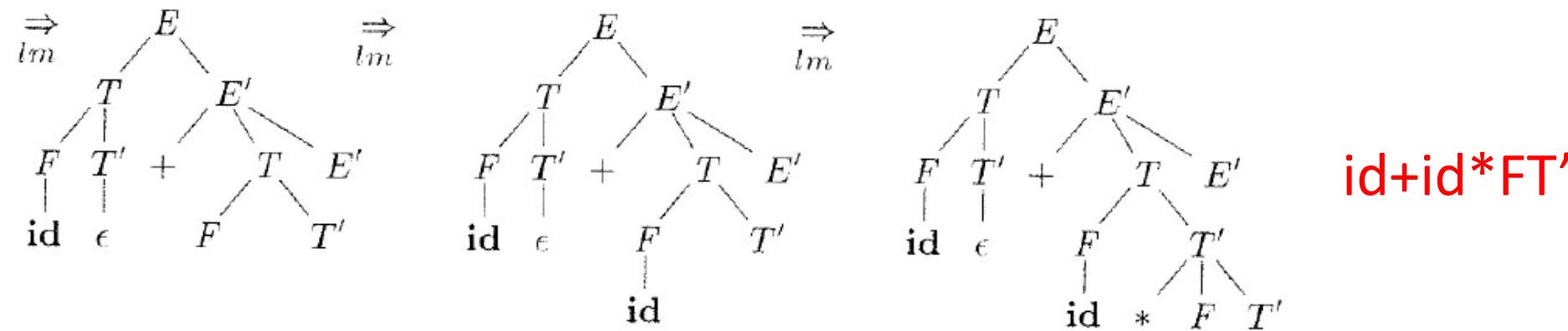
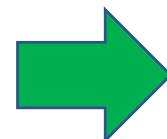
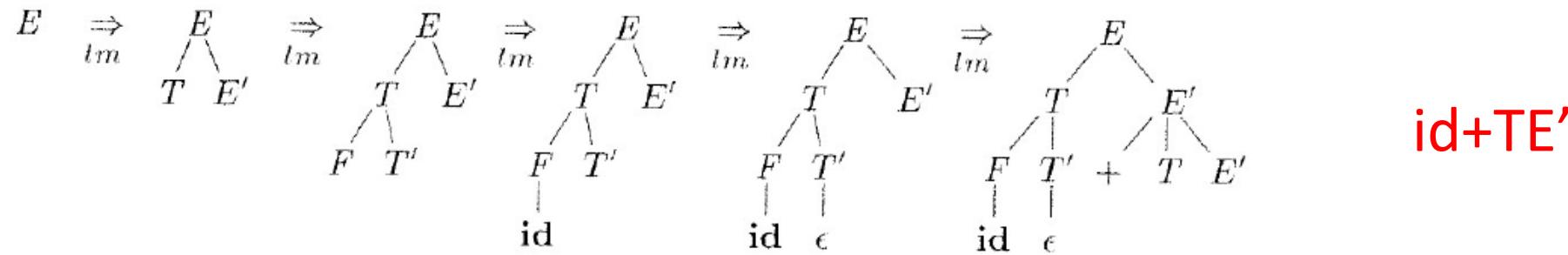
Non-terminal	Input Symbol						
	<b>id</b>	+	*	(	)	\$	
$E$	$E \rightarrow TE'$			$E \rightarrow TE'$			
$E'$		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$	
$T$	$T \rightarrow FT'$			$T \rightarrow FT'$			
$T'$		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$	
$F$	$F \rightarrow \text{id}$			$F \rightarrow (E)$			

**ε is a “zero symbol” which is used to terminate branches we don’t want**

**Parsing table row = what should be replaced**  
**Parsing table column = what is our target → specifies which production should be used, so the process is now deterministic!**

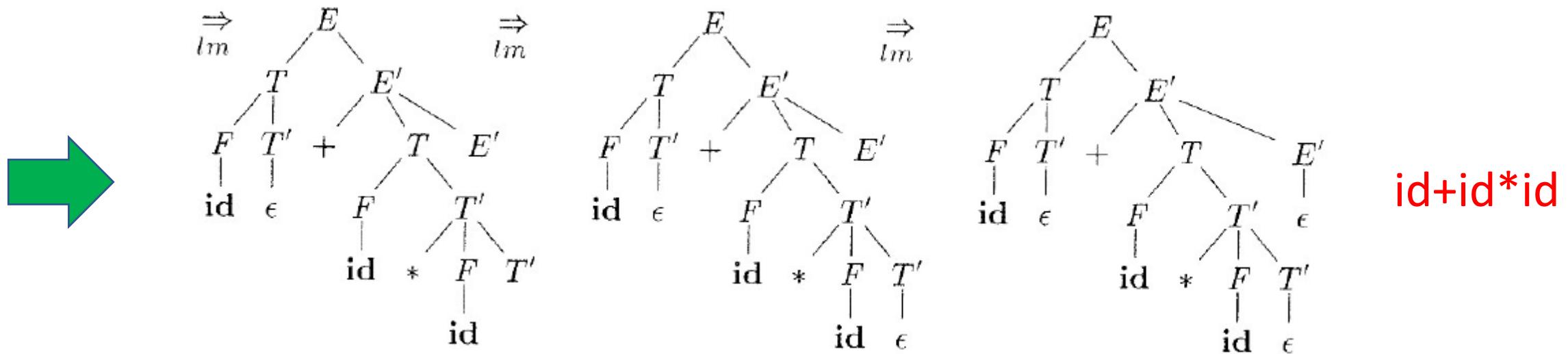
# From parsing table to parse tree

- Parse tree iteration for input id+id\*id in the previous grammar (top-down):



# From parsing table to parse tree

- Parse tree iteration for input  $\text{id}+\text{id}^*\text{id}$  in the previous grammar (top-down):



- Success!

# Condensed parse trees

- As you can see, parse trees for even simple inputs look a bit complex
- However, the syntactic structure of an expression can be determined without writing all terminals and non-terminals
- Parse trees can be condensed
  - Simpler look
  - Easier to read and draw
  - Sufficient for illustration purposes

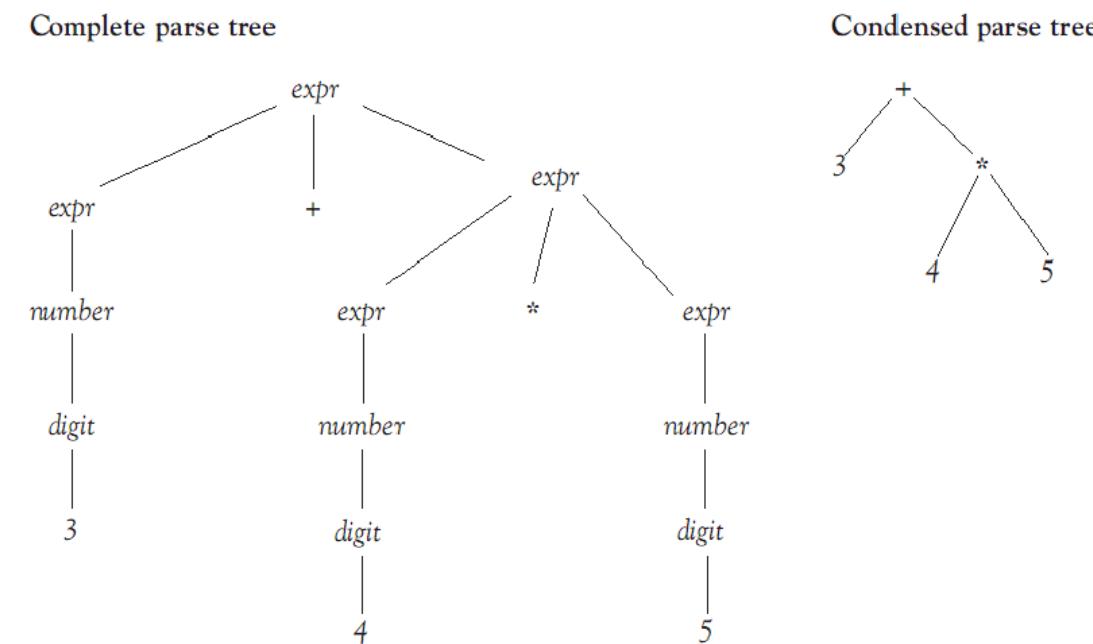
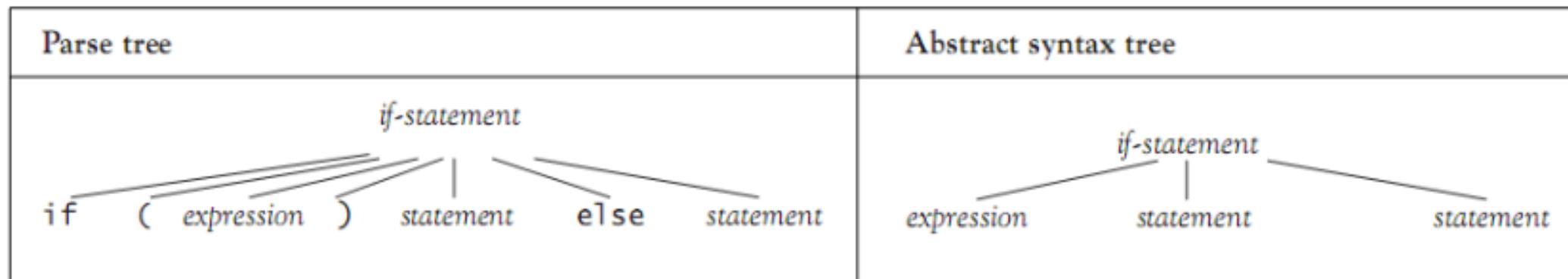


Figure 6.10: Condensing parse tree for  $3 + 4 * 5$

# Abstract syntax tree (AST)

- Abstract syntax trees (ASTs) are a special case of condensed parse trees
- Require understanding of the structure (so, not just the syntax, but also semantics)
- Redundant terminals can be removed

*if-statement* → if ( expression ) statement else statement



# Code generation

- After parse tree (or AST) of the program is formed by the parser, comes the 3<sup>rd</sup> phase – generating the actual machine language program:
  - Memory allocation
  - Formulation of machine language commands
- It is common that programs are not translated straight from high-level language to machine language, but the program is first translated to some mid-level language
  - Machine language programs are computer-specific
  - Mid-level language program is easier to translate further to several different computers
  - Beneficial, if the same program will be launched to different platforms (for example, 32-bit & 64-bit versions – or a desktop version & tablet/cellphone version)

# Memory allocation & symbol table

- Memory will be allocated to identifiers according to their type & value
  - Floating-point numbers, integers etc. require different amounts of memory
- Memory addresses of identifier values will be decided
  - Addresses can be physical or virtual
- These will be written in the symbol table that was generated during lexing phase

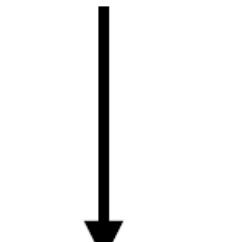
Identifier	Type	Size	Address
X	Integer	4	245
Y	Integer	4	249
Sig	Sign	1	253
Rem	Float	4	254

# Formulation of machine language commands

- From parse tree to symbolic machine language

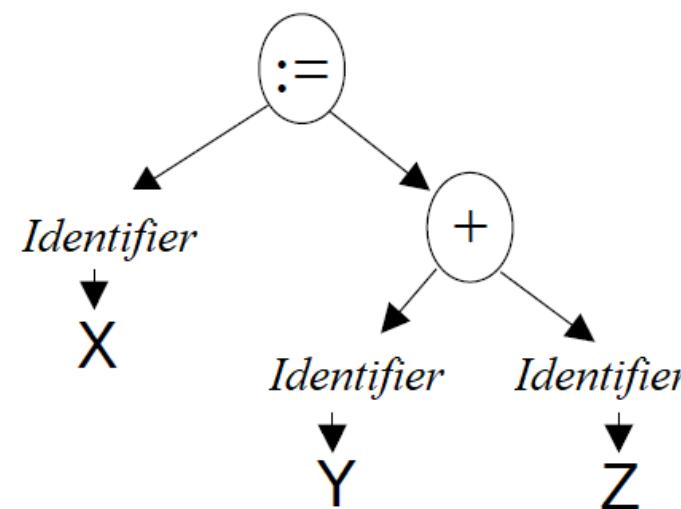
**Input & output:**

$x := y + z$



LOAD y  
ADD z  
STORE x

**Parse module:**



**Code generation module:**

MODULE set(x, y, o, z)

Print "LOAD y"

CASE o of

'+' : print "ADD z"  
'-' : print "SUBTRACT z"  
'\*' : print "MULTIPLY z"  
'/' : print "DIVIDE z"

ENDCASE

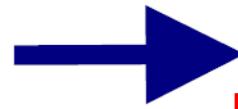
Print "STORE x"

ENDMODULE

# Optimization

- Mechanically generated code can usually be improved
- Different compilers vary in their efficiency to optimize the code they generate
  - Identifying and removing unnecessary parts of the programs
  - Use the operations which are most efficient to perform on the destination computer
- Example:

$X=Y+Z;$   
 $B=X*A;$



LOAD Y  
ADD Z  
STORE X  
LOAD X  
MULTIPLY A  
STORE B



LOAD Y  
ADD Z  
MULTIPLY A  
STORE B

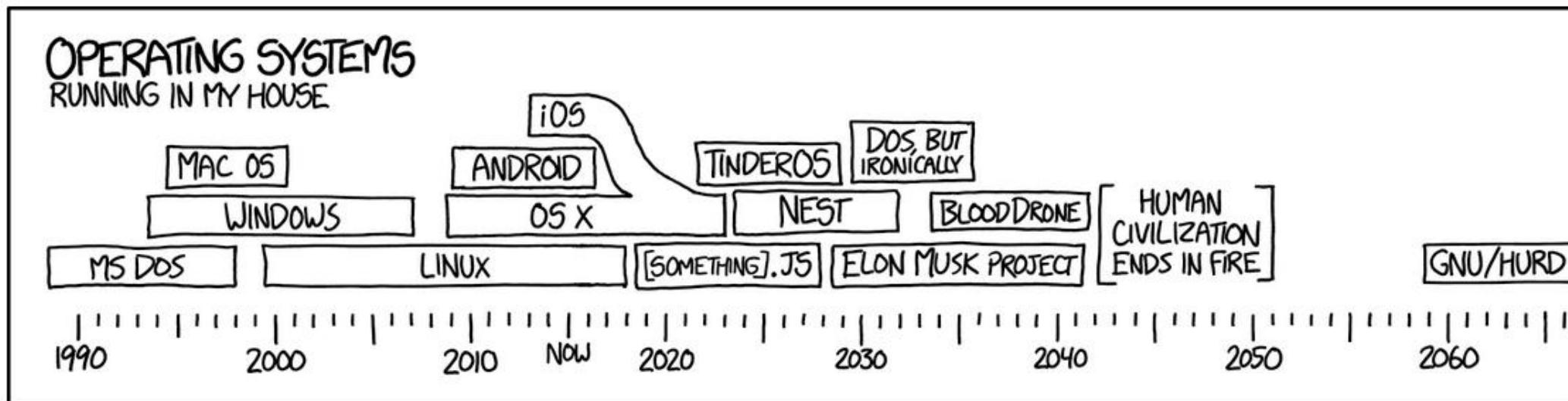
No need to store and then again load X, because X is already in ACC after 2<sup>nd</sup> line!

Possible pitfall:  
now no value is stored in X, so if we later would need it for something else, this step shouldn't be done!

# Thank you for listening!

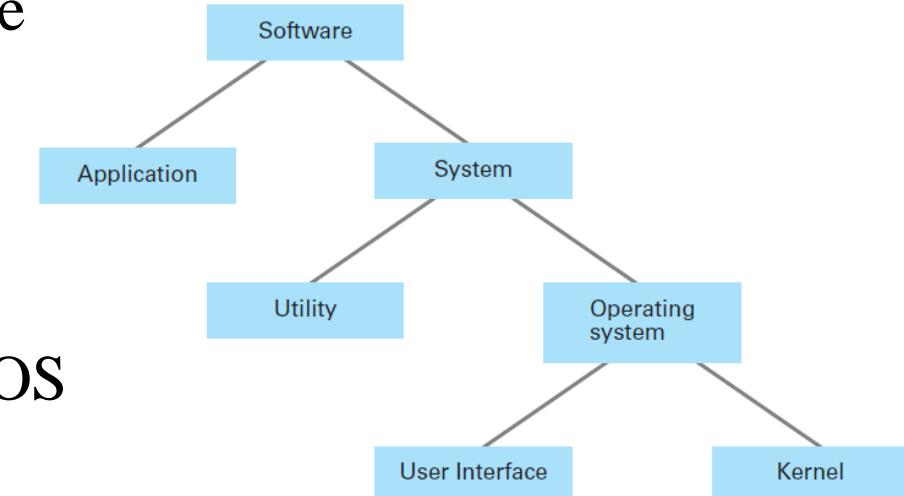


# 6. Operating system: tasks, process control and scheduling



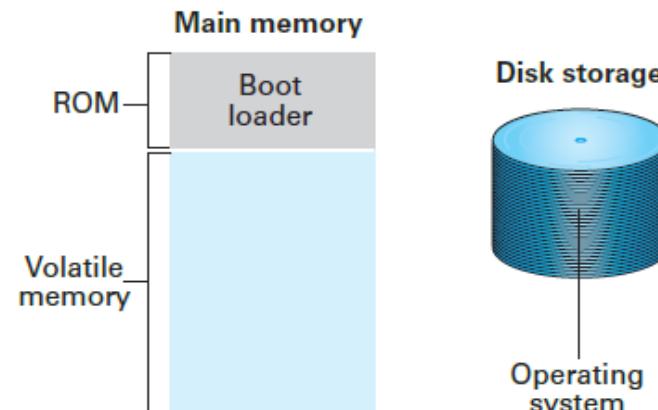
# Types of software

- Computer software can be divided into two categories: system software and application software
- Application software = “what the computer is used for”
  - Office programs (text processing, spreadsheets)
  - Calculation and analysis programs
  - Production control programs etc.
- System software provides the infrastructure for these
  - Operating system (OS) and utility software
- Operating system controls the use of resources
  - Kernel contains the basic functions of OS
  - Users communicate with kernel through user interface
- Utility software extends/customizes the features of OS
  - Defragmentation tools, network communications etc.

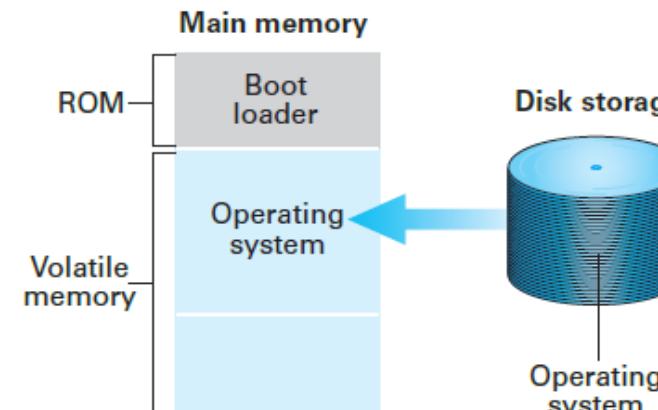


# Booting

- Each time when the computer is started, it has to be able to start the OS
- Starting the OS is known as booting
  - When a PC is started, BIOS checks the system and then initiates the boot loader
  - Boot loader starts and loads the OS from disk storage to RAM
  - Strictly speaking, boot loader is not in ROM – it can be altered under special circumstances



**Step 1:** Machine starts by executing the boot loader program already in memory. Operating system is stored in mass storage.



**Step 2:** Boot loader program directs the transfer of the operating system into main memory and then transfers control to it.

# Operating system

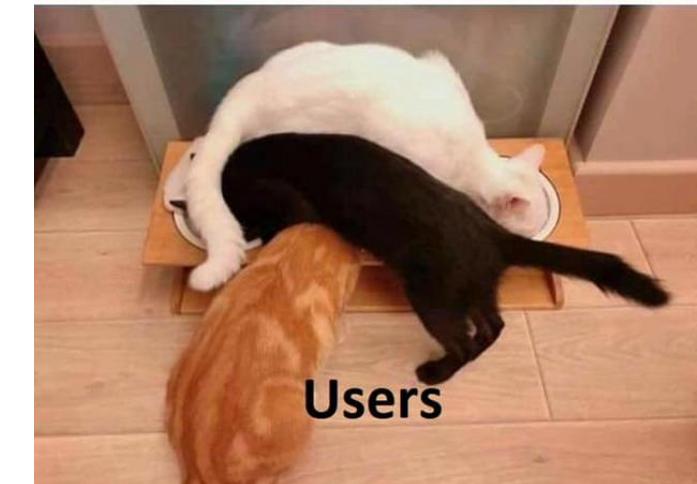
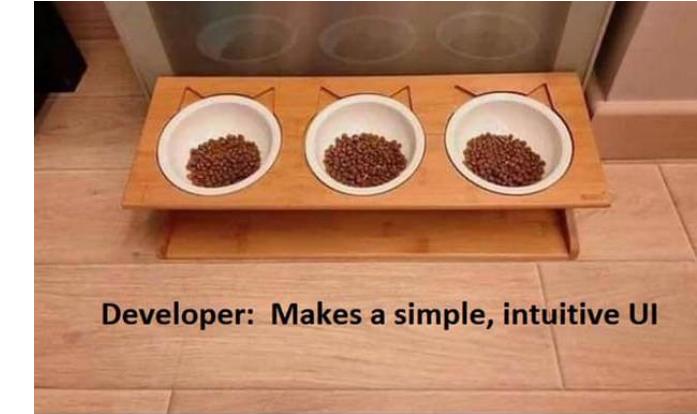
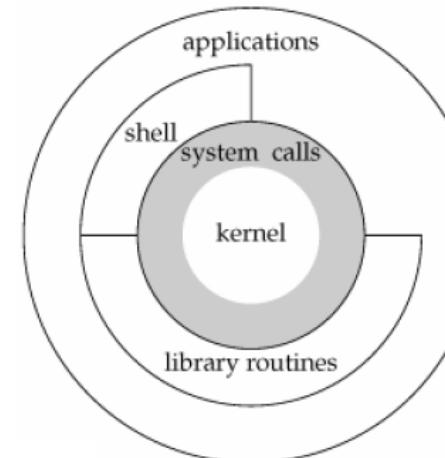
- Basically all computers are delivered with some kind of an operating system
  - DOS, Windows, macOS, Linux, Chrome OS...
- Operating system is the software used for controlling the hardware of the computer as well as execution of application software
  - Application software is run via the operating system
- Executing a program under control of an OS is called a *process*
- Processes are coordinated by controlling the allocation of resources:
  - CPU time scheduling
  - Memory allocations in different level memories
- Input/output (I/O) is controlled via handling I/O requests and interrupts

# Operating system tasks

- Resource allocation
  - Memory, processor cycles, I/O devices
- Dispatching
  - Exchanging the process currently running in CPU
- Scheduling
  - Keeping track of processes (in queue or in execution) via process table
  - Decision on which process will be selected for execution next
  - Multiple possible decision criteria for selection (known as scheduling methods)
- Resource protection
  - Making sure that a process can't access a resource it hasn't reserved
- Interrupt handling
- Handling of I/O requests

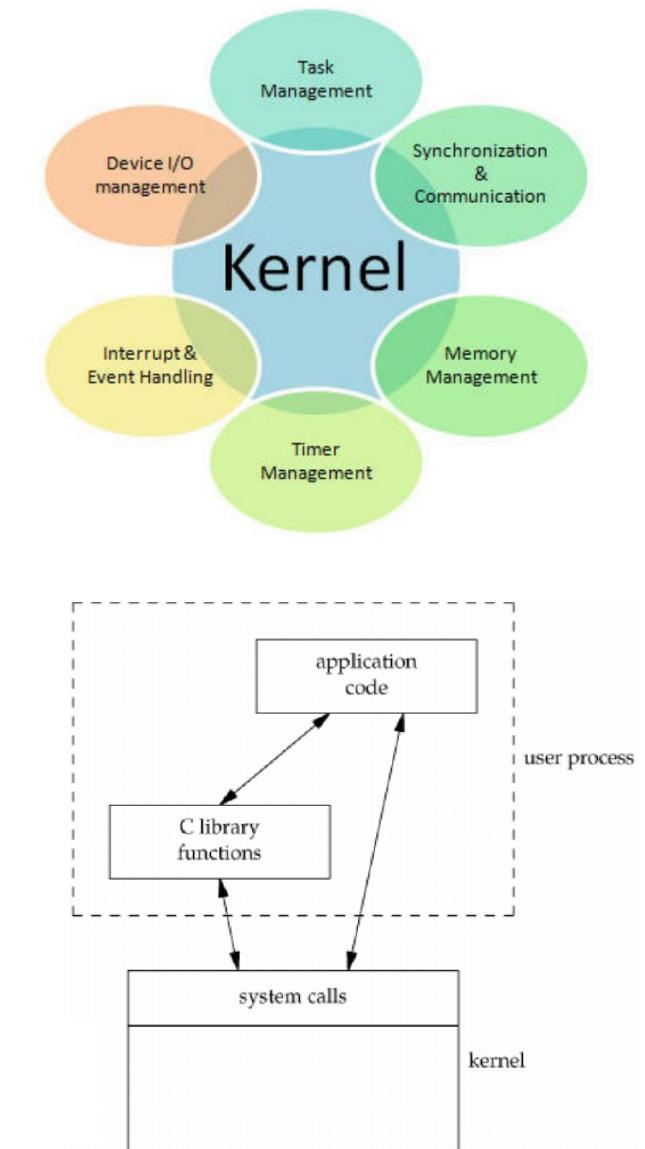
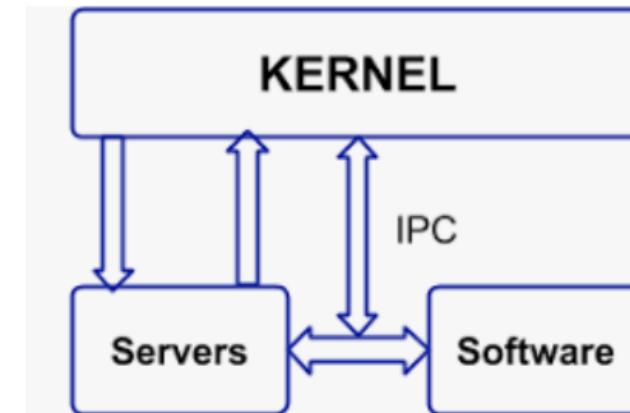
# Layers of an operating system

- Kernel is a program set that contains the basic functions
  - Kernel instructions are run in privileged state
- Shell & library routines, or user interface (UI) in general
  - Intermediary between users and the kernel
  - Shell is an old-fashioned, text-based UI
    - Extended by library routines
  - Nowadays the vast majority of operating systems use some kind of Graphical User Interface (GUI)



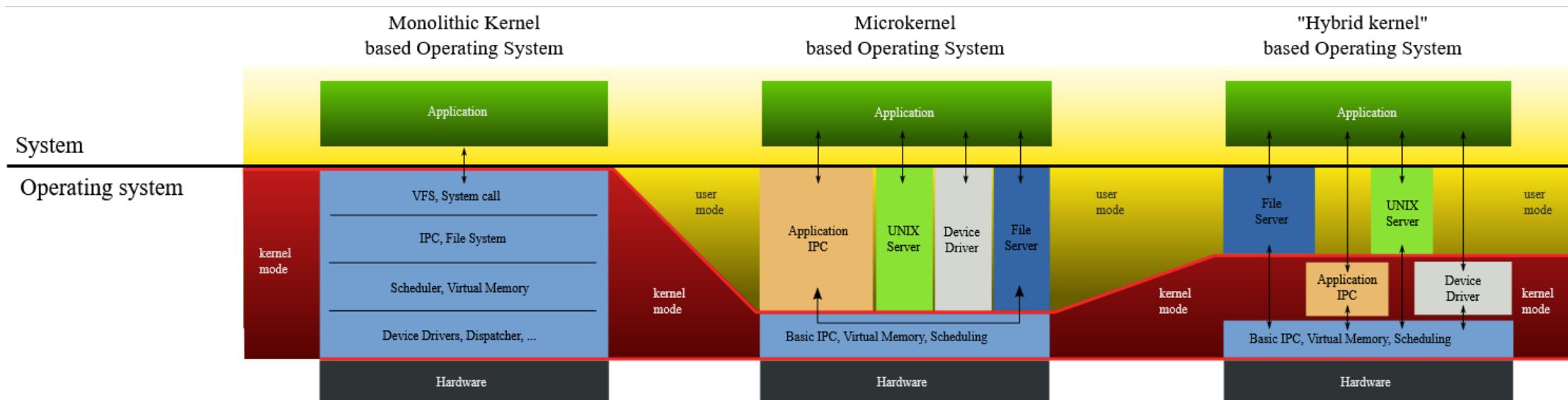
# Tasks of a kernel

- System call interface
- Process control
  - Creating and removing processes
  - Scheduling between processes
  - Conveying messages between processes (Inter-Process Communications, IPC)
  - Memory control (allocation)
- I/O control
  - File system
  - Buffering
  - Device management



# Kernel architectures

- There are two basic kernel architectures: monolithic and microkernel
  - In monolithic kernel design, basically the whole OS is placed in kernel
  - In microkernel, the kernel only contains the bare minimum
- Hybrid kernels combine features from both



(Excellent illustration by Golftheman from Wikipedia)

# Microkernel architecture

- Only most essential functions to kernel
- Instructions started by kernel are executed in privileged state
  - Initiation of interrupt handling (what caused the interrupt?)
  - Dispatching functions (usually just copying registers)
  - Memory control functions (memory control unit settings, protection)
  - Inter-process communications (conveying requests, copying data)
  - I/O functions (use of disk drives)
- Other OS instructions are normal processes, which are executed in user state
  - Device drivers, file system
- Benefits of microkernel architecture:
  - Modularity and flexibility of OS (adding new modules requires no changes to kernel)
  - Stability and reliability are easier to attain (user state processes can't crash the computer)

# Comparison and use of kernel architectures

- Monolithic kernels have better performance, because communication with applications and devices are done using system calls
  - Downside is that system stability is harder to reach (for example, one bad device driver can cause the whole computer to crash)
  - Also, adding new features is limited, because it always requires changes to the kernel
  - Examples: Linux, DOS, Windows 9x
- Microkernels offer better stability and security, but performance is worse due to slower communication method (message passing)
  - Popular choice in small devices due to minimal size
  - Examples: Horizon (Nintendo Switch), L4 (embedded systems)
- Most modern OS use a hybrid-type kernel
  - Examples: Windows 7/10, macOS

# Processes

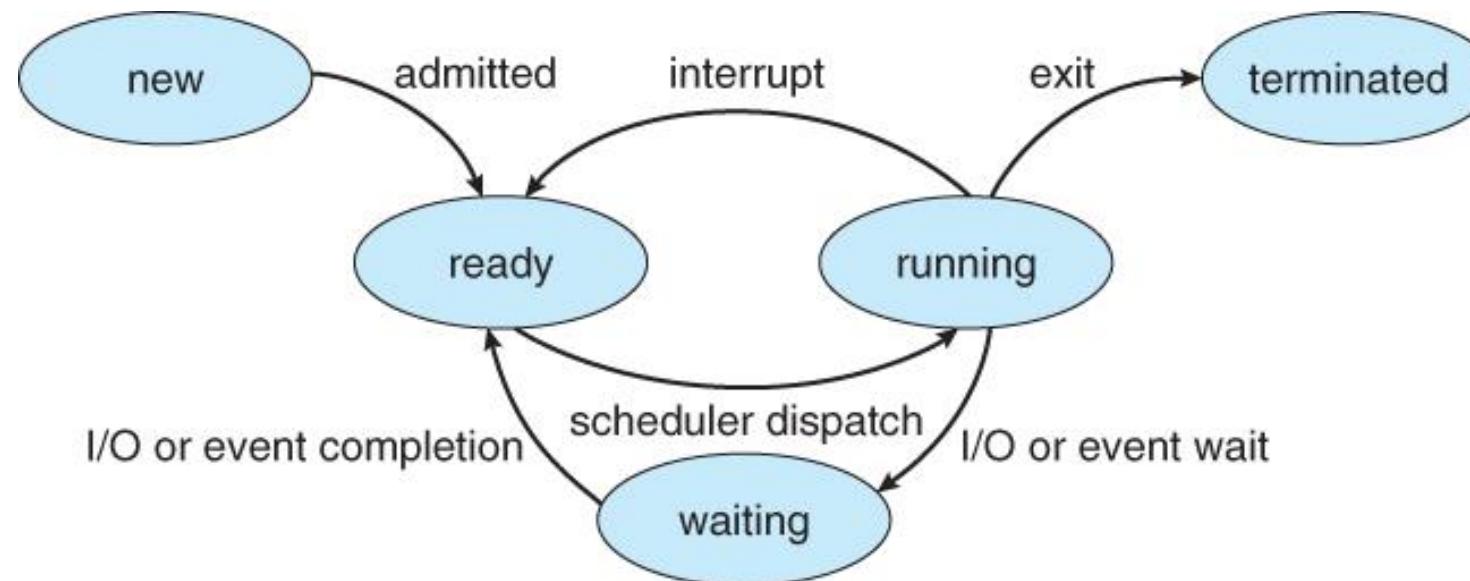
- Program = a series of commands, stored on hard disk
- Process = execution of a selected program in selected environment
  - Analogy: sheet music of a song vs. a song performed by an artist on a gig
- Process state = current (momentary) state of execution
  - Ready / running / waiting for resources
- In a computer, there are usually dozens of processes going on
  - Some initiated by the user, some by the OS
- Processor time and other resources must be divided between processes
  - Rapid switching of running process creates the user a sense of multitasking, even though there is always only one process (per core) actually running

# Dispatching

- Single task of a program can be comprised of several processes
- Only one process can be in running state per processor core, so the CPU resources (clock cycles) have to be dealt between processes
- Dispatcher picks up the selected process from process queue, moves it to running state and gives it a permission to use the CPU
  - Analogy: TV singing contest – processes are contestants, dispatcher is the production assistant who fetches the next contestant from backstage and gives him/her the tools needed (mic etc.)
- Dispatching can be pre-emptive (process halted when still running) or not pre-emptive (process is changed only after the previous one has terminated)
- Dispatcher is activated when
  - Currently running process terminates or reaches the time limit it was given
  - Currently running process performs an I/O request

# Process states

- Because I/O requests take some time to complete, a process performing such a request is put on waiting mode
- After I/O request is responded to, the state of the process is changed to ready
- Dispatcher takes ready processes from queue to running when scheduler sees fit



# Scheduling of resources

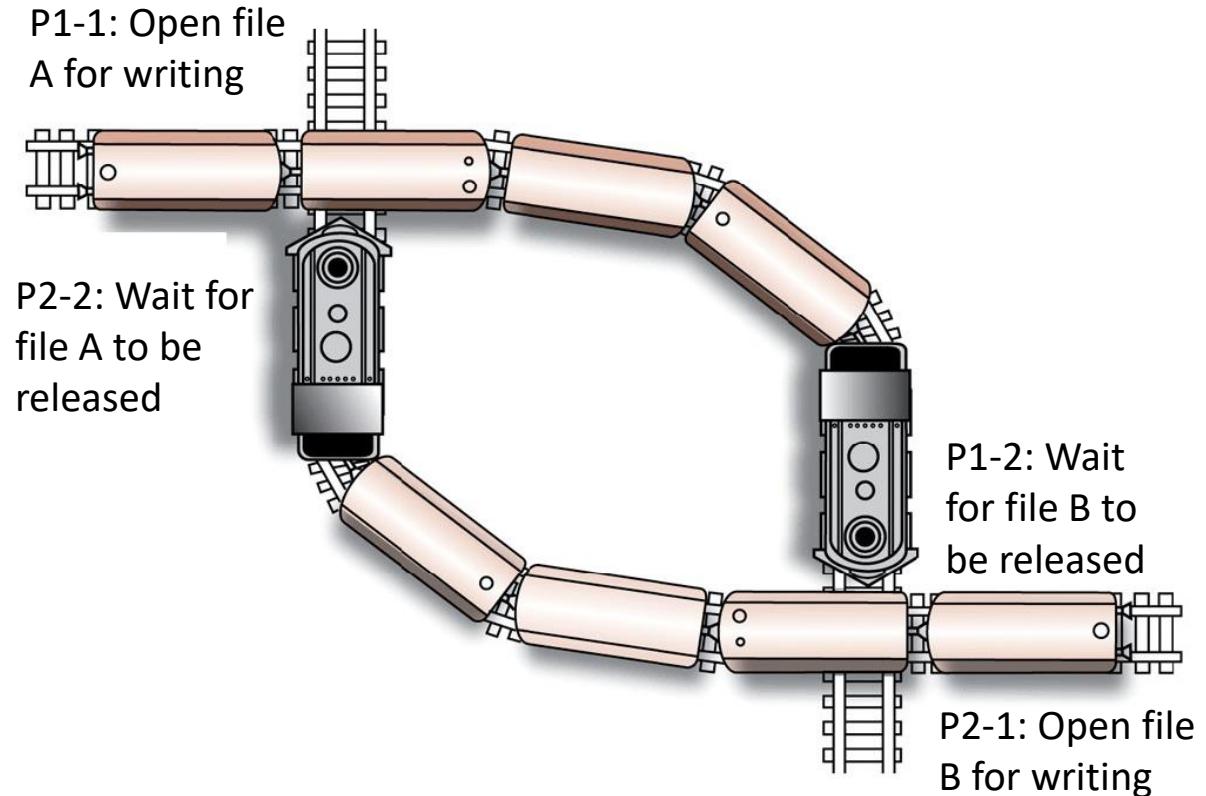
- The order in which queued processes are taken to running state is specified by the scheduler
  - TV singing contest analogy: broadcast director is the scheduler (decides who goes on stage next)
- Scheduler has to take into account:
  - Resources needed
  - Resources currently available (free)
  - Priority level of task
  - Expected waiting time before execution
- Allocation of resources can be done in static or dynamic fashion:
  - Static: all resources ready before process starts to run
  - Dynamic: resources will be reserved during the run
- Dynamic allocation can lead to a deadlock situation

# Deadlock

- In a deadlock, two processes are waiting for resources that are reserved for each other
- Deadlock can occur, if the following conditions are satisfied:
  - There is competition for non-sharable resources.
  - The resources are requested on a partial basis (so, not all at once)
  - Once a resource has been allocated, it can't be forcibly retrieved
- Prevention: kill commands, spooling

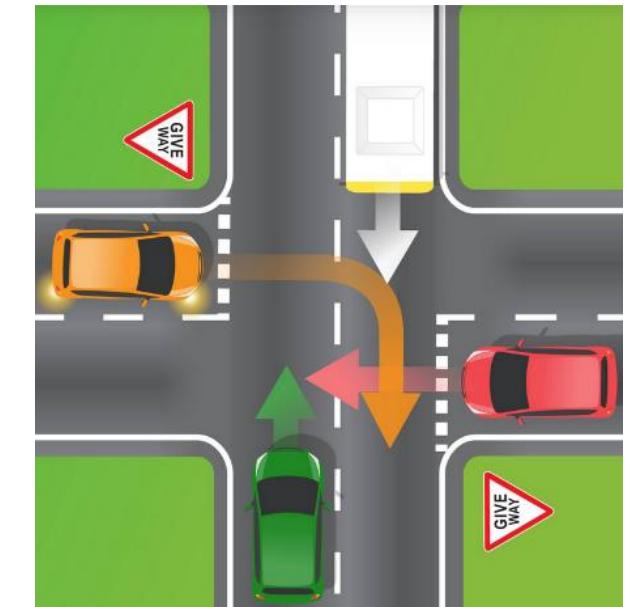
Example:

- Process 1 tries to copy contents of B to A
- Process 2 tries to copy contents of A to B
- Scheduler deals CPU cycles in order P1, P2, P1, P2



# Starvation

- Another possible problem that might occur when dealing resources is starvation
- This means a situation, where some process is constantly denied necessary resources
- Real-life example: busy major road
  - Cars coming from side roads have to give way
  - Theoretically, if major road traffic is constant, it's never their turn
- Usually a passing problem, doesn't continue forever
- Good scheduling algorithms are starvation-free
- Exploited by hackers
  - Some denial-of-service (DoS) attack types aim at starvation

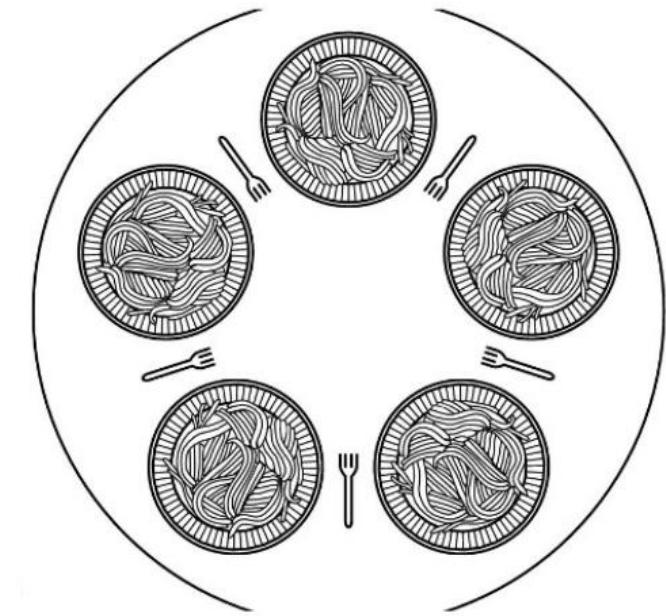


# Example: Dining philosophers (Tanenbaum, 2015)

- Premises:
  - A philosopher always either thinks or eats
  - A philosopher needs two forks for eating spaghetti\*
  - There are 5 philosophers, each has a plate and one fork
- Eating algorithm: does this work?

```
#define N 5

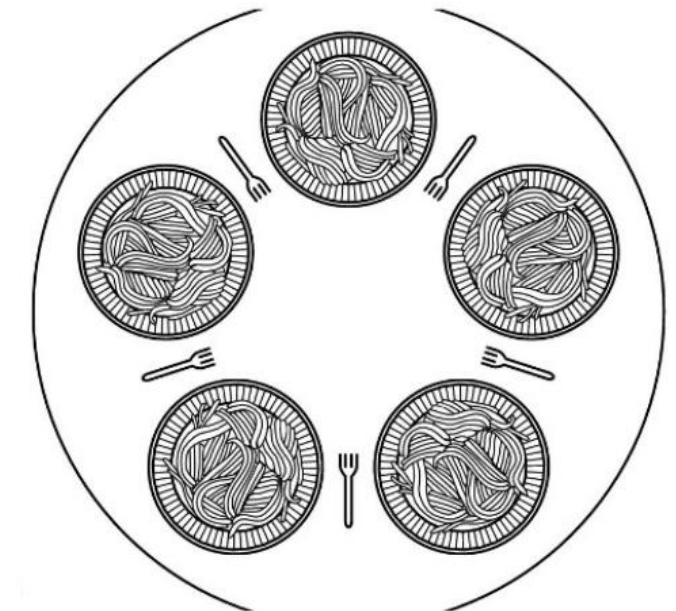
void philosopher(int i)          /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think();                  /* philosopher is thinking */
        take_fork(i);             /* take left fork */
        take_fork((i+1) % N);    /* take right fork; % is modulo operator */
        eat();                     /* yum-yum, spaghetti */
        put_fork(i);              /* put left fork back on the table */
        put_fork((i+1) % N);    /* put right fork back on the table */
    }
}
```



\*Actually, this Tanenbaum's version is a clumsy translation: in original version it was noodles and chopsticks, which makes a lot more sense.

# Example: Dining philosophers (Tanenbaum, 2015)

- If the philosophers run the whole algorithm one person at a time (no interrupts), then yes
  - One eats, others just wait; inefficient
- If the philosophers run the algorithm simultaneously – or one instruction at a time for each, this results in deadlock
  - Everybody will sit idle with a left fork in hand
- If the philosophers don't run the algorithm simultaneously but stagger the starting time by an interval of  $t$ , the algorithm works (if  $t$  is different for everyone)
  - If  $t$  is the same for all, two pairs of philosophers can starve the fifth one, because they will alternate eating turns
- Many ways of solving the problem (Google if you're interested)



# Scheduling algorithms

- As we noticed from the previous example, scheduling resources for processes is not an easy task but contains several pitfalls (mostly regarding efficiency).
- Numerous algorithms have been designed for scheduling:
  - FIFO (First-In-First-Out) or FCFS (First-Come-First-Served)
  - LIFO (Last-In-First-Out)
  - RR (Round Robin) – uses division to time slices (quanta)
  - SJF (Shortest Job First) or SPN (Shortest Process Next)
  - HRRN (Highest Response Ratio Next)
  - Feedback
  - FSS (Fair-Share Scheduling)
- Algorithms differ in efficiency and priority criteria
  - Efficiency can be evaluated using several metrics

# Scheduling algorithms

- Comparison table of most common scheduling algorithms
- “Efficiency” is relative to application of the computer (what to favor?)

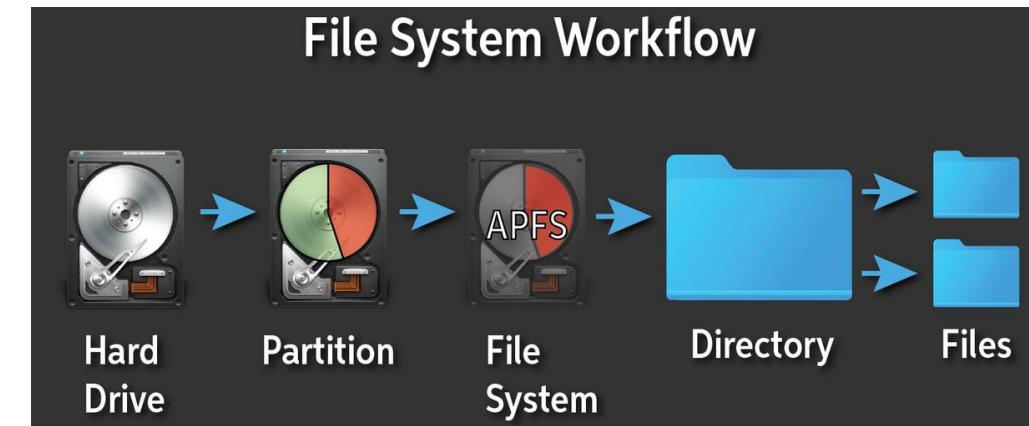
	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s – e]		(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Through-Put	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

# Fair-share scheduling

- FSS divides processor resources to time quanta just like Round Robin
- The difference lies in equal division of resources between users instead of processes (different users may have differing number of processes in queue)
- For example, if user X has processes A, B, C and D and user Y has process E in queue
  - RR: A, B, C, D, E, A, B, C, D, E, A, B,...
  - FSS: A, E, B, E, C, E, D, E, A, E,...
- Possibility to grant also different share of resources to users – if in previous example resources are divided in 2:1 share between users X and Y
  - FSS: A, B, E, C, D, E, A, B, E, C, D, E, A,...
- Prevents greedy users from hogging all resources
- Used in Linux (especially due to server use)

# File system

- One important task of an OS is to control and maintain the file system
- Hard drive: physical disk where information is “permanently” stored
- Partition: logical section of the hard drive
- File system: method how data on the partition is stored and organized
  - FAT = File Allocation Table; simple and compatible, but limited features
  - NTFS = Newer replacement; supports larger files and enhanced security
  - Several others (APFS, SquashFS, HPFS,...)
- Directory (folder): collection of files
- Path: location of file in folder hierarchy
- File descriptor: unique file identifier
  - “Keys” to use the file



# Information security and data protection

- Identification of users
  - Traditionally, only users who know the username and password are allowed access
  - Nowadays also other means of identification (fingerprint, face recognition, retina scanning, mobile identification applications)
- Every process has an owner, and it uses resources only by owner's permission
  - Either some user or OS ("System" in Windows PCs)
- Rights to access resources
  - Files have owners, who specify permissions
  - Only the owner of a file can alter the permissions
- Programs and their data must be sheltered from other programs
  - Especially critical to shelter the OS from applications
- Mutual use of resources must be allowed in certain cases

# Desired features of an OS

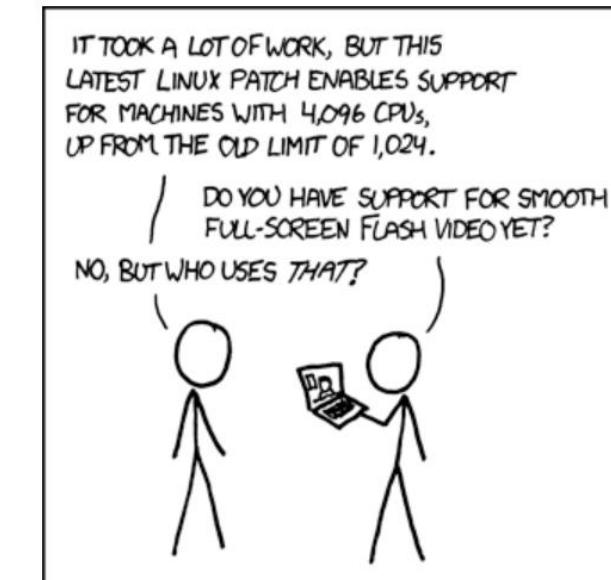
- Good performance (short response time, high throughput)
- Stability (MTBF, mean time between failures)
- Data protection and security (resistance to data breaches)
- Scalability to different environments
- Extensibility (possibility to add new features easily)
- Portability to multiple devices
- Safety and reliability (low chance of user errors)
- Interactivity (ease of communication with users)
- Usability (user-friendliness)



GraphJam.com

# OS development and administration

- No operating system is “finalized” at any point
- Computers and devices evolve
  - Switches, punch cards, magnetic tapes, disks, solid state memory...
  - From Text User Interface (TUI) to Graphical User Interface (GUI)
  - Massively increased amounts of memory (all types), improved bus speeds
  - Support for virtual memory
  - Increased clock speeds, multiple core processors
- Information processing methods evolve
  - Interactive real-time systems (require massive data transfer speed)
  - Graphical windowing environments
  - Image processing and video editing
  - Local area networks and Internet, dealing with large user amounts
  - Machine learning, Big Data, pattern recognition, neural networks...



# OS development and administration

- Due to continuous need for development, prefer
  - Modular structure
  - Clear interfaces between modules
  - Possibly object-based implementation
  - Internal vs. public data (internal data not visible to users)
- All operating systems contain deficiencies and mistakes, so they need to be updated as these get fixed
  - Patches and service packages
  - New OS versions
- Completely new OS, when it's time
  - Need for new module structure
  - Code inefficiency due to multiple patches



# Present and the future

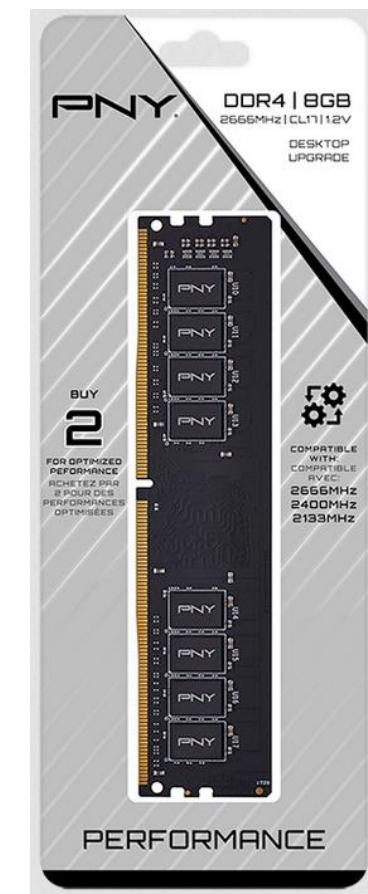
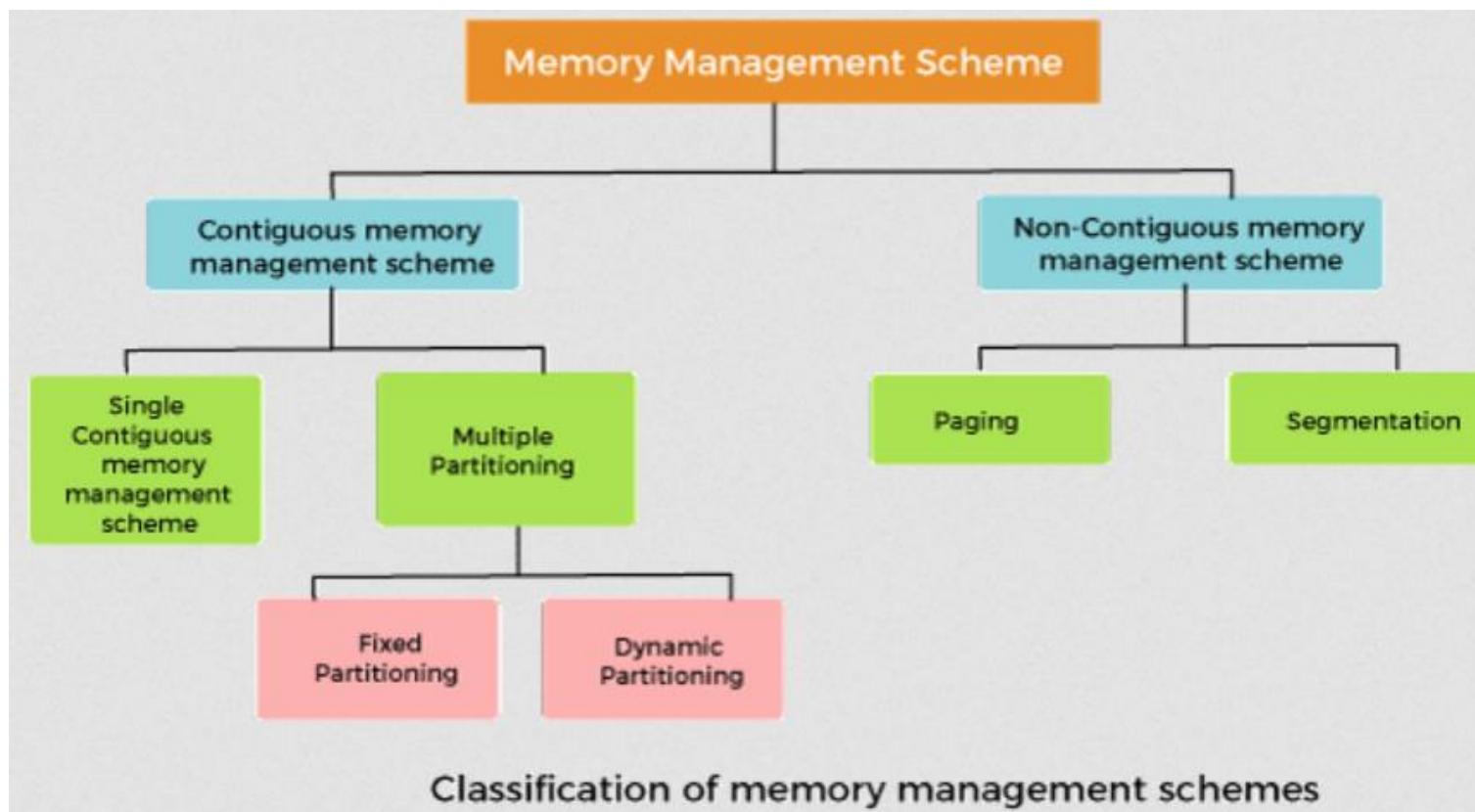
- Trends in hardware development
  - Multiprocessor systems, increased use of GPU (Graphics Processing Unit)
  - Fast telecommunications networks
  - Increased number of cores in processors, optimization
  - Improved memory, quicker disk storage (M.2 SSD, thousands of megabytes per second)
- Trends in software usage
  - Customer/server-model; IaaS (Infrastructure as a Service; programs and databases are located on a server leased from a server company)
  - Mining of cryptocurrencies (Proof-of-Work vs. Proof-of-Stake)
  - Streaming services (4K or even 8K quality)
  - Metaverse?
- Mobile OSs and their developer ecosystems
  - Basically a duopoly between Android and iOS – are there no challengers?



# Thank you for listening!

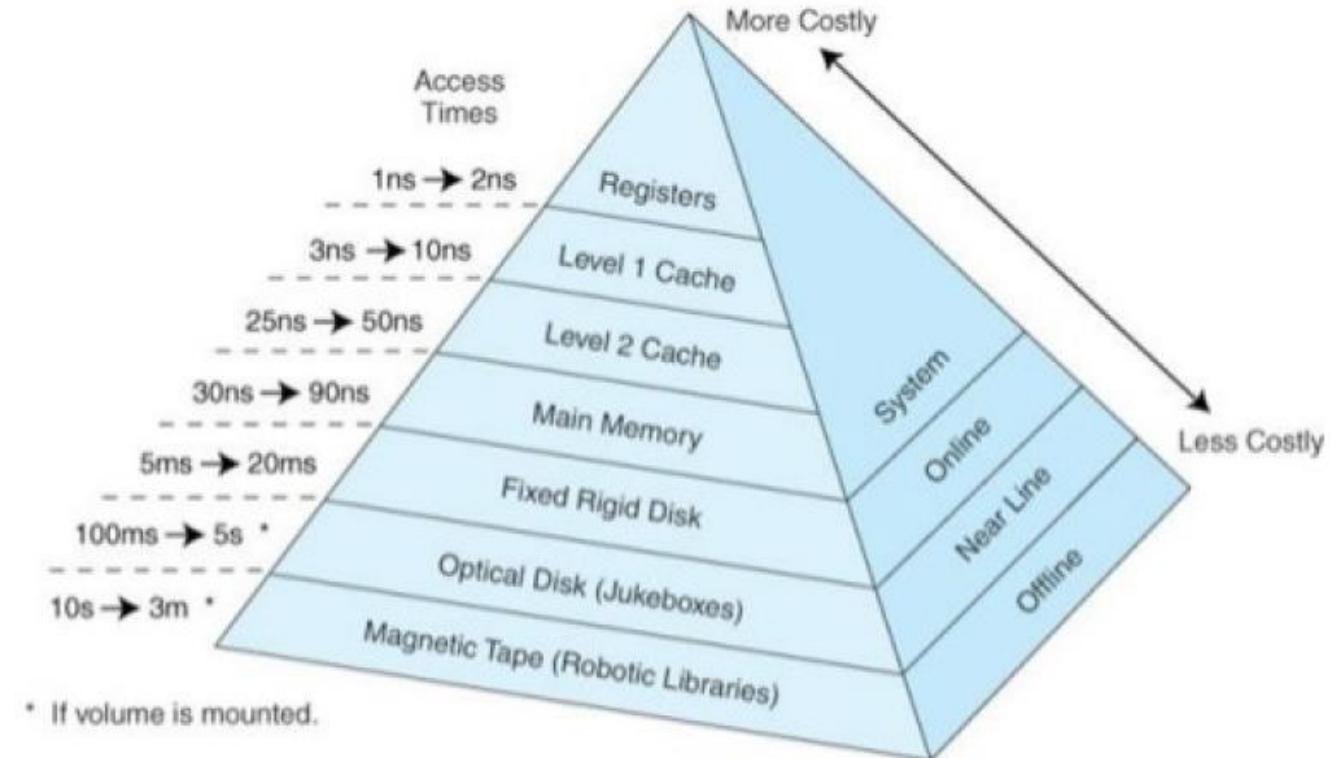


# 7. Operating system: memory control and I/O



# Memory hierarchy

- There are 3 different types of memory in a computer:
  - CPU memory (registers & cache)
  - Main memory (RAM)
  - Auxiliary memory (disk/tape drives)
- The faster the memory, the more expensive it is
- Auxiliary memory can be located also offline outside the computer
  - Naturally, accessing this memory requires it to be mounted

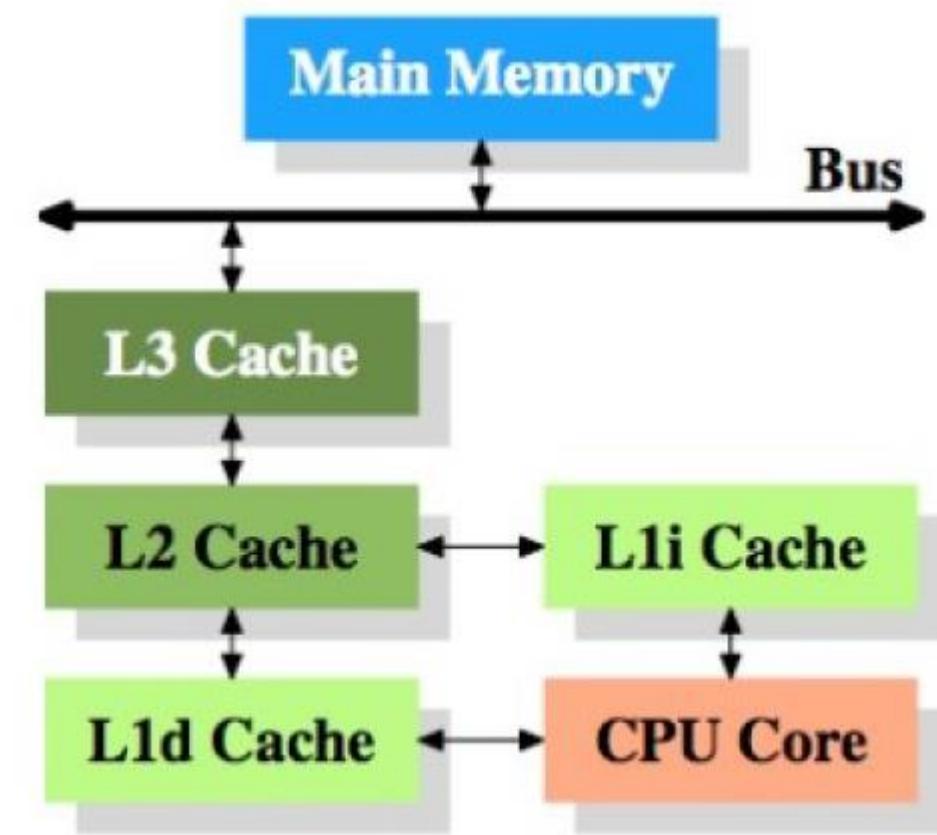


\* If volume is mounted.

Magnetic tape drives are not ancient history – they're still used today! IBM is a big manufacturer of tape libraries. Good read here: <https://spectrum.ieee.org/why-the-future-of-data-storage-is-still-magnetic-tape>

# Caching

- Size of memory has positive correlation with access time
- For this reason, CPU cache has been divided to several levels (nowadays usually 3)
  - If the desired information is not found from the nearest cache, search continues to next one (and onwards to main memory, if needed)
  - Information that is needed most often is therefore stored nearest the CPU core
- For example, Intel core i7-12700:
  - L1 cache: 960 KB (80KB per core)
  - L2 cache: 15 MB (1.25 MB per core)
  - L3 cache: 25 MB (shared between cores)



Level 1 cache is, in some designs, divided to instruction (L1i) and data (L1d) parts.

# Speed, latency and bandwidth

- When we're talking about memory performance (and especially main memory), three important concepts that often emerge are *speed*, *latency* and *bandwidth*
- Each data transfer (read or write) is performed on a clock cycle
- RAM “speed” = RAM frequency [MHz]; how many million data transfers the memory module is able to complete per second
  - In SDR (single data rate) memory, data is only transferred on one edge of the clock, so 1 transfer per clock cycle
  - In DDR (double data rate) memory, data is transferred on both edges of the clock, so 2 transfers per clock cycle
- Latency = response waiting time while moving the data [ms or ns]
- Bandwidth = how much data can be transferred per time unit [Mbits/s or MB/s]

# Calculation of latency and bandwidth

- RAM modules are usually given latency values as CAS latency (CL)
  - CL = number of clock cycles it takes for the memory module to respond
- Actual latency (L) in [ns] can then be calculated using the latency equation:
  - Example: 2400 MHz DDR4, n = 2, CL15 → L = 12.5 ns

$$L = \frac{1000 \cdot n \cdot CL}{S}$$

CL = CAS latency

S = RAM speed [MHz]

n = Transfer factor (for SDR = 1, for DDR = 2)

- Theoretical maximum bandwidth B in [MB/s] similarly using the bandwidth equation:
  - Example: 2400 MHz DDR4, b = 64 bits/s, c = 4 → B = 76 800 MB/s

$$B = \frac{S \cdot b \cdot c}{8}$$

S = RAM speed [MHz]

b = Memory bus width [bits/s]

c = Number of channels supported by CPU

# Memory control & fragmentation

- Alongside process scheduling, memory control is one of the most important tasks of an operating system
- Things to consider during this process include
  - Keeping track of the state of each memory location (is it free or not?)
  - Allocation of memory according to the memory requirements of the process
  - Protection (process can only access memory that has been allocated to it)
  - Efficient use of memory, so that processes don't have to wait for memory in vain
- Inefficient use of memory is usually caused by fragmentation issues
  - Internal fragmentation = amount of memory allocated to a process is greater than required
  - External fragmentation = free memory consists of small slots which are unusable due to their size being too small to fit a process
  - External fragmentation can be "cured" by rearranging the memory at time intervals (compaction); internal fragmentation, on the other hand, is harder to tackle

# Memory addresses

- When a program is translated, compiler (or interpreter) generates the memory addresses for all data and instructions
- These addresses are called program addresses or logical (virtual) addresses
- When a process is prepared for execution in a computer, the data and instructions must be loaded to the main memory (in some addresses)
- These actual addresses are called memory addresses or physical addresses
- So, in order to execute the program, the OS has to:
  - Allocate a sufficient amount of memory to the process(es)
  - Convert the logical addresses to physical addresses
- This conversion can be done either in static or dynamic fashion

# Static vs. dynamic conversion

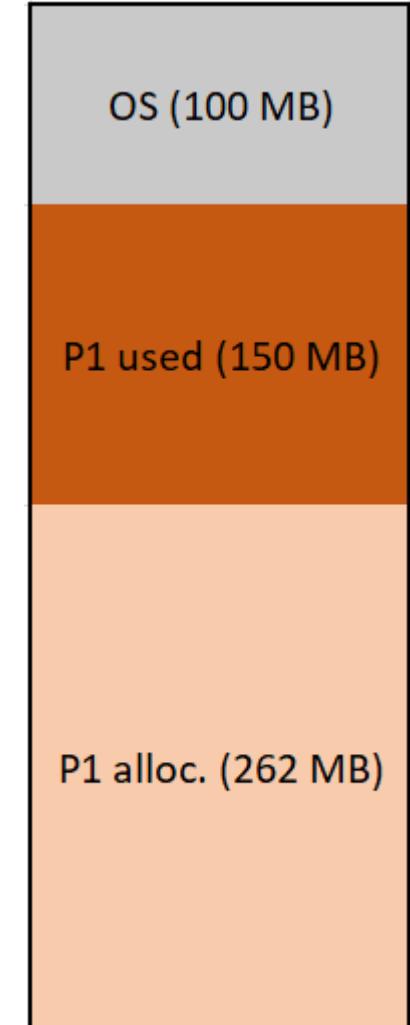
- In static address conversion, all data and instructions needed by the process must be loaded to physical memory at once
  - So, the amount of (free) memory in our system limits the number & size of processes
  - Address conversion is done by the loader before execution starts
- In dynamic address conversion the CPU (or MMU, to be exact) converts the program addresses to physical addresses during the execution
- Therefore, it enables us much more freedom with our memory usage:
  - Instructions can be loaded as the execution proceeds
  - Old instructions can be discarded and replaced with new ones
  - Amount of memory will not be a “hard” limit (although constant loading and discarding is slow)
- Dynamic conversion methods that enable this partial loading are *paging* and *segmentation*

# Memory allocation techniques

- There are several techniques to implement memory allocation
- In the following slides, these techniques are demonstrated by using an example computer, that has 512 MB of main memory – of which 100 MB is reserved for the operating system – and a couple of processes running
- Note: do not confuse static/dynamic address conversion with static/dynamic memory allocation!
  - These share some similarities, but they don't mean the same thing
  - Dynamic memory allocation means that we can change the size of memory areas that we allocate to processes
  - Dynamic memory allocation therefore enables us to eliminate (or at least minimize) internal fragmentation

# Single contiguous allocation

- All memory (that is not used by OS) is allocated to a single process, no matter how much it actually requires
- Very inefficient way of memory management
- No (external) fragmentation, though
- Also, very secure method
- Bad idea for computers that have to run several processes at the same time
- Plausible method for simple computers that do just one thing at a time (for example, embedded systems)



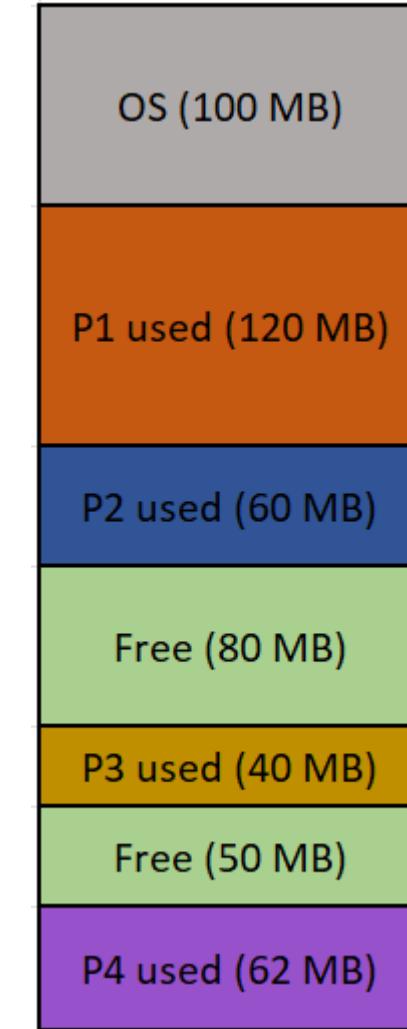
# Partitioned contiguous allocation (static)

- Main memory is divided into fixed-size partitions
  - Example on the right: 4 partitions – 100, 130, 70 and 212 MB
- All memory in a single partition is allocated to a single process
- Number of partitions limits the number of processes that can be run simultaneously
- Better alternative, but not very flexible
- Both external and internal fragmentation
  - Example on the right: system could initiate a 3rd process (one free partition), but if the 3rd process requires more than 70 MB of memory, it can't fit in memory and it has to wait



# Partitioned contiguous allocation (dynamic)

- Memory is divided in partitions, but the division is done dynamically – i.e. partitions are created “on the fly”
  - Size of partitions not fixed; process can be allocated a partition that matches exactly the amount that the process requires
  - Number of partitions is not limited in any way
- Free parts are called *holes*
- New processes are placed in holes, if the hole is large enough to fit the process
  - Leftover part of the hole forms a new (smaller) hole
  - Example on the right: if P5 which requires 65 MB of memory is initiated, it will be placed in the 80-MB hole → results in a 65-MB partition for P5 and a 15-MB hole
- External but no internal fragmentation

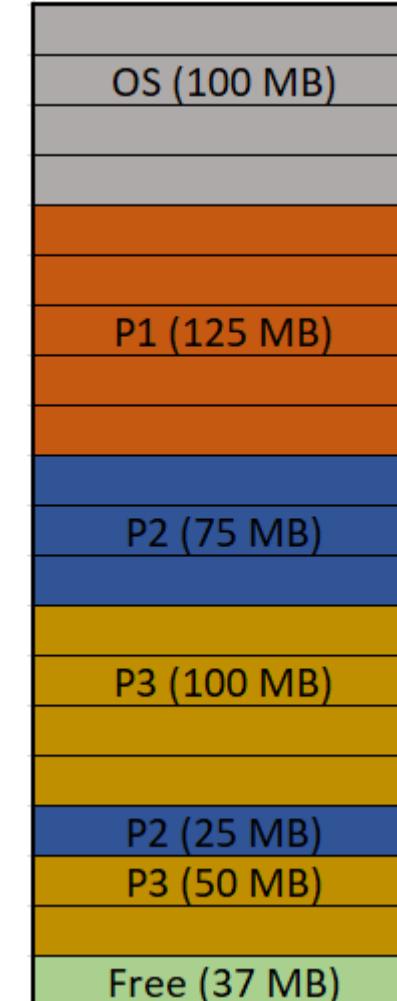


# Hole fitting techniques

- If there are multiple holes, where should we place the next process?
  - Several techniques for hole selection
- First fit = start search from beginning, place the process in the first hole that is large enough
- Next fit = as previous, but when the next process arrives, we continue hole search from the point of last allocation (instead of starting from beginning again)
- Best fit = search the whole memory for holes, place the process in the smallest hole that is large enough
  - Produces the smallest leftover hole
- Worst fit = search the whole memory for holes, place the process in the largest hole
  - Produces the largest leftover hole

# Paging

- In order to eliminate external fragmentation, we have to get rid of the contiguousness constraint
- In paging, the program memory is divided into fixed-size parts called pages, and the physical memory is divided into same-size parts called frames
- This allows the program memory to be contiguous, but the information can be stored anywhere in physical memory – in as many parts as needed
- A common page size is  $4\text{ KB} = 4096\text{ B}$
- Internal fragmentation remains a (limited) problem
  - Last page of the program is left partially empty (example: if the program size is  $9000\text{ B}$ , it uses 3 pages – last page is only  $808\text{ B}$ )



# Page table

- Computer naturally needs to be able to link pages and frames together
- This is done via a *page table*, that is maintained by OS
- Page table contains information about all pages:
  - If the page is in main memory, what frame is it linked to
  - If the page is not yet in main memory, where is it located on hard disk (so that it can be fetched to main memory)
  - Possible additional Y/N fields (read-only, modified, etc.)
- Decreasing page size reduces internal fragmentation, but increases the number of pages → increases the size of the page table
  - Therefore, page size choice is always a compromise

P1 PMT	
Page	Frame
0	5
1	12
2	15
3	7
4	22

P2 PMT	
Page	Frame
0	10
1	18
2	1
3	11

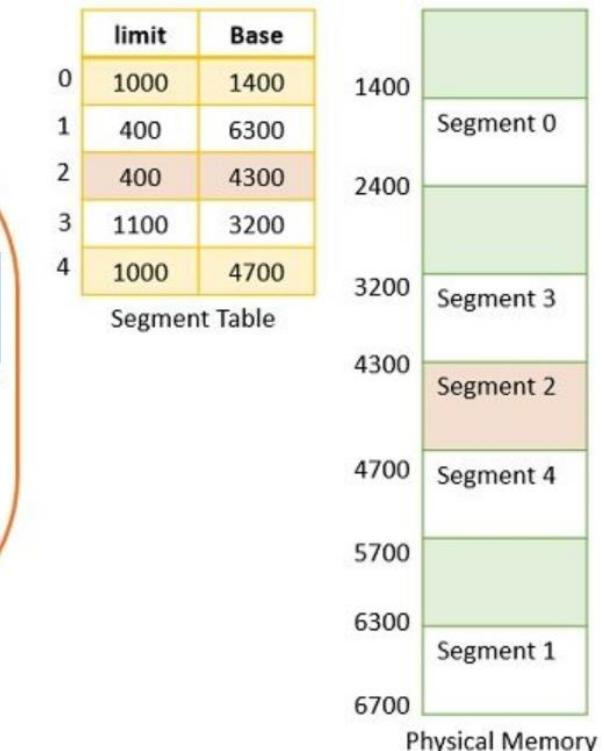
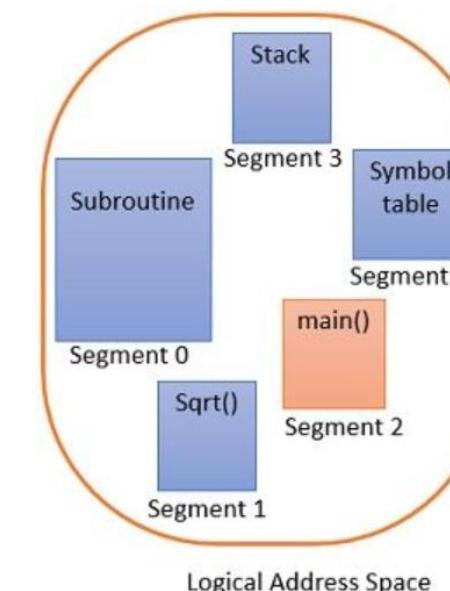
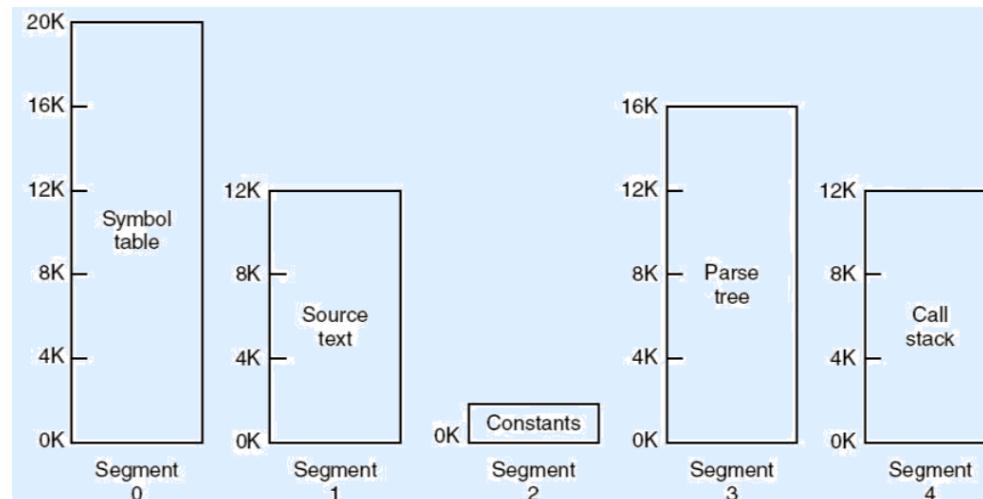
Memory	
Frame	Contents
0	
1	P2/Page2
2	
3	
4	
5	P1/Page0
6	
7	P1/Page3
8	
9	
10	P2/Page0
11	P2/Page3
12	P1/Page1
13	
14	
15	P1/Page2

# Segmentation

- Programs are seldom just one entity, but they comprise of several elements:
  - Main program, shared libraries, data stack etc.
  - Size of these elements may change during the execution
- Especially if the memory address conversion is done in dynamic fashion, this causes problems: the main program can easily be loaded to main memory “little by little”, but the shared libraries and data stack are used constantly
  - Paging results in all these elements getting mixed up in different pages, so there may be a need to load and then re-load the same pages several times
- Segmentation aims to solve this issue by creating segments (of variable size) where all these elements are stored (one element per segment)
  - Segments that have shared libraries can be kept in memory all the time
- Eliminates internal fragmentation, but external remains a bit of a problem
- Solution: combined segmentation and paging

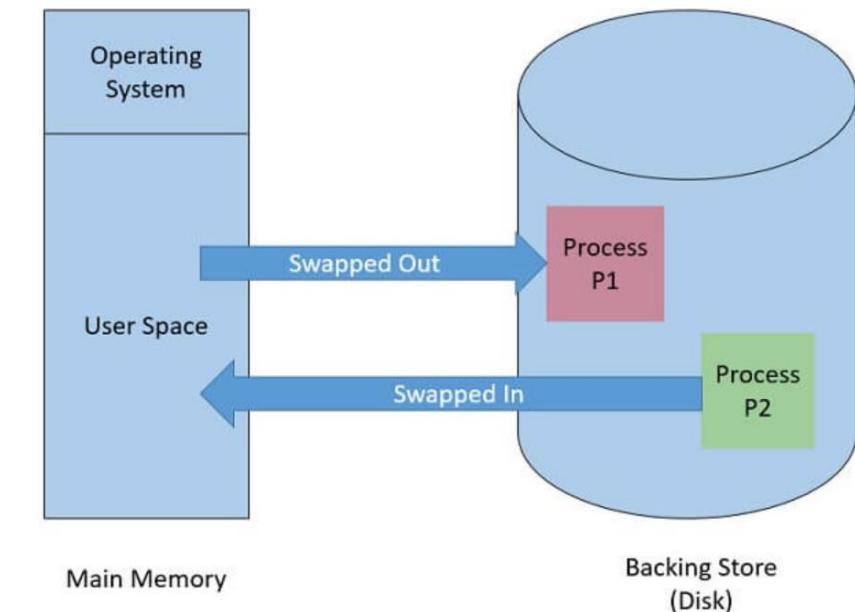
# Segmentation

- Segmentation allows each table (segment) to grow or shrink independently
- Segmentation is the only memory management method where even the program memory is not contiguous
- Information of physical address starting point (*Base*) as well as size of segment (*length* or *limit*) are stored in *segment table*
  - Memory address = Base + offset



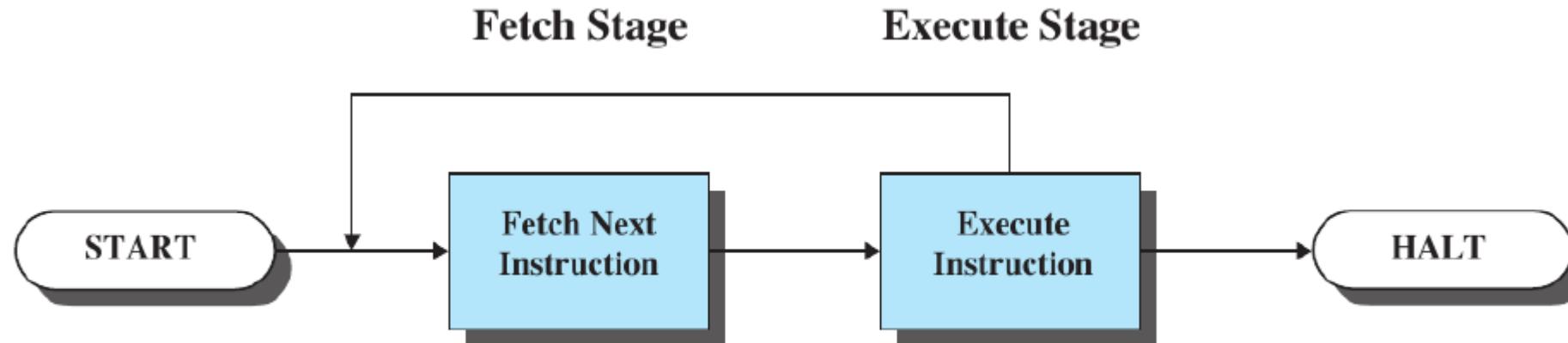
# Swapping

- If we don't have enough space in our (physical) memory for the process we'd need to execute, we can swap out information that is not needed right now to hard disk
- This frees us main memory space so that the information needed by the executing process can be swapped in
- Swapped-out processes are usually ones that have performed a (probably lengthy) I/O request and hence are in waiting state
- Multiple possible criteria for swap-out process selection



# Simple process execution cycle

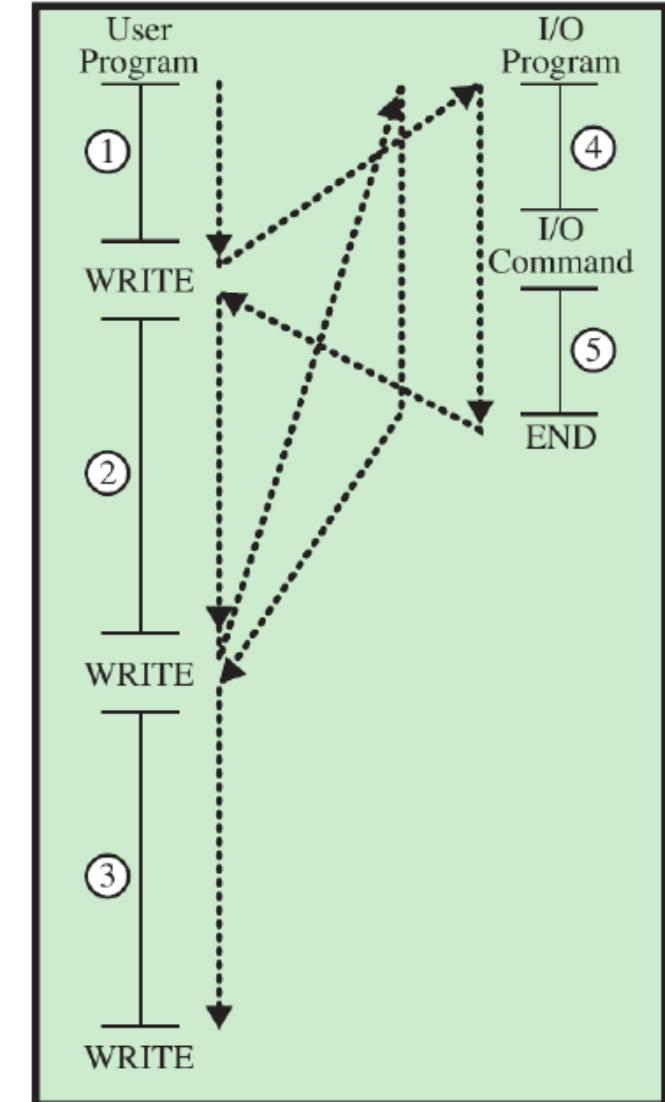
- CPU fetches an instruction from a memory address given by the program counter
- Program counter value is increased and needed operands fetched
- Instruction is executed and the result is placed in memory or register
- Continue to the next instruction



Stallings: Operating Systems, 9e, Prentice Hall

# I/O in a simple program

- In a simple non-interrupting cycle, I/O works like this:
  - User program encounters “write” command and makes an IPC request that gives control to an I/O program (1)
  - I/O program initiates the I/O process and gives the I/O command (4)
  - I/O program waits (polls) that the I/O operation is complete (5)
  - I/O program performs the end tasks and returns the solution information about its state (2)
  - Process continues until another “write” command surfaces – then the previous steps (4) and (5) will be repeated
- The CPU waits a “long” time in phases (4) and (5)!!
  - Inefficient use of CPU resource



# Interrupts

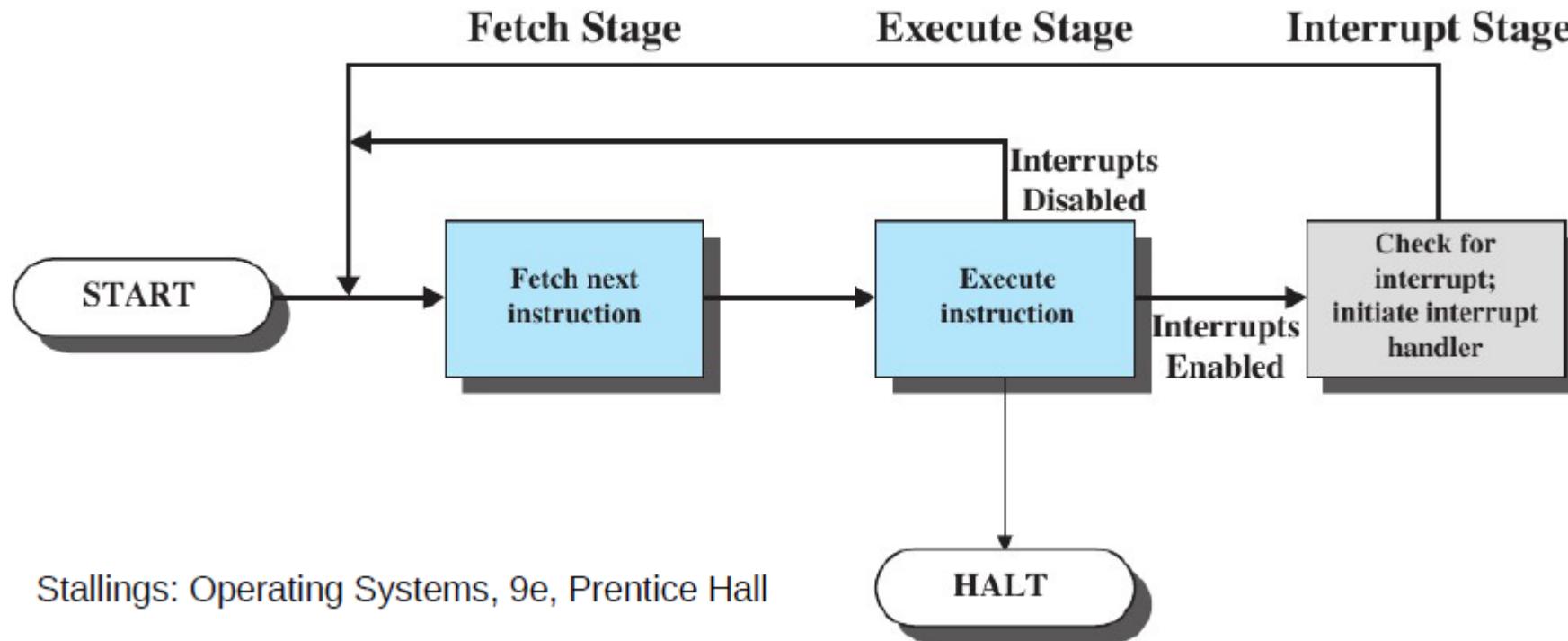
- Unlike in real life, in computers interrupts are a “good” thing: they enable the simultaneous use of CPU and I/O programs
- Data transfer is slow (especially if data is fetched from hard drive) and the original process often can’t continue until it’s done
  - The goal is to execute other process(es) meanwhile waiting for I/O to finish
- CPU only initiates data transfer – after that it’s free to continue execution of other processes (no CPU idle time)
- When the I/O is completed, the I/O program interrupts the CPU
- CPU then starts interrupt handling
- When the handling is done, the process that was waiting for I/O completion can be taken back to execution

# I/O control methods

- Programmed I/O
  - No interrupts from the I/O program; CPU checks whether I/O is done
  - CPU must wait in idle for I/O completion
- Interrupt-driven I/O
  - When an I/O event is started by process 1, the CPU continues to execute other processes
  - When I/O has been completed, the I/O program interrupts the execution of other processes and saves the data of the process currently in execution
  - Data from I/O is loaded to the CPU
- Direct memory access (DMA)
  - Like interrupt-driven I/O, but when the I/O has been completed, the I/O program transfers the data directly to the main memory (without going through CPU)
  - Nowadays the most common method due to effectiveness

# Interrupting program execution cycle

- If our control method enables interrupts, CPU checks for interrupts before fetching the next instruction
- If there's an interrupt in the interrupt register, interrupt handler is initiated



Stallings: Operating Systems, 9e, Prentice Hall

# Classification of interrupts

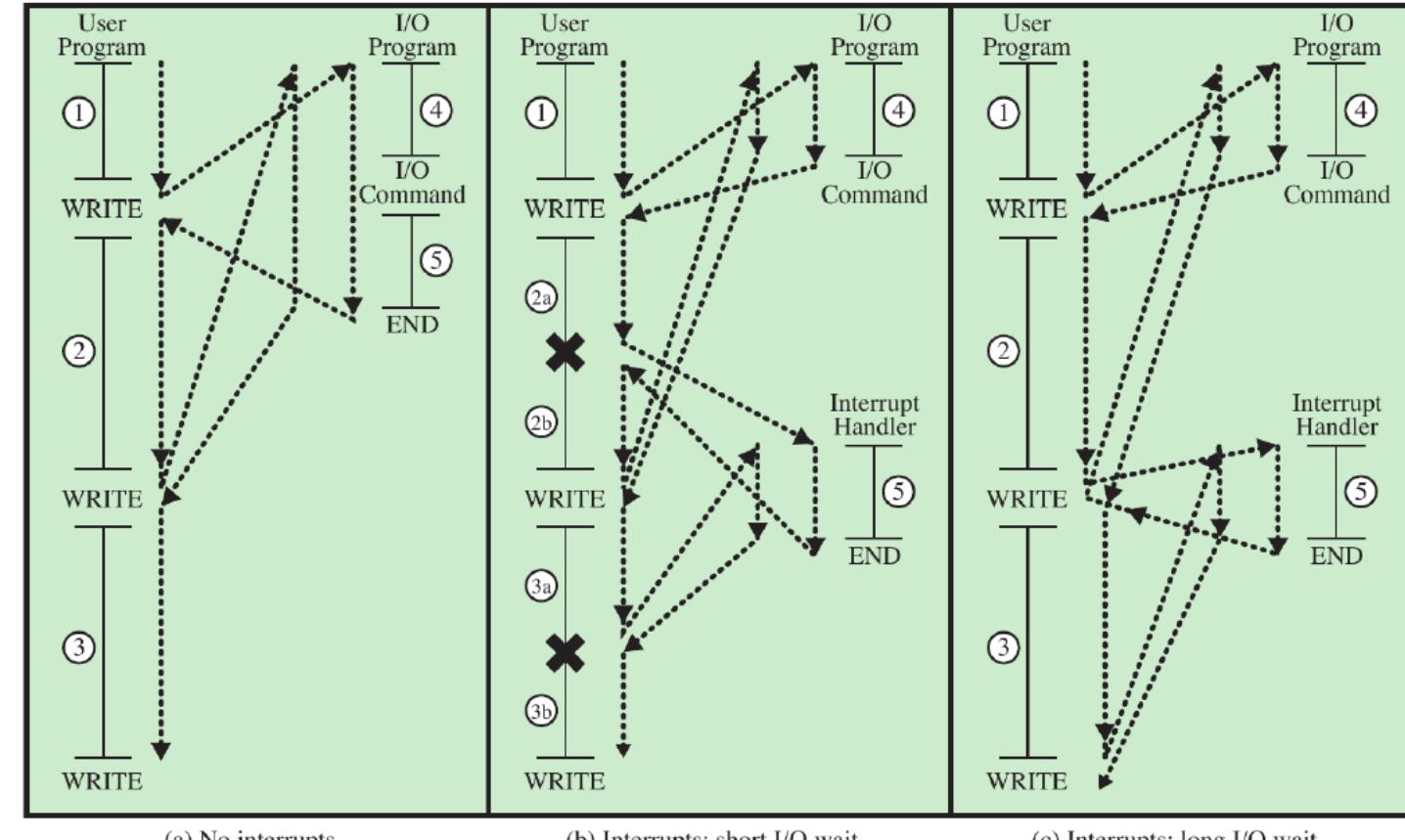
- Program interrupt (normal or exceptional)
  - Normal: generated by a program sending an IPC or by user request
  - Exceptional: generated by some condition that occurs as a result of an instruction execution, such as arithmetic overflow/division by zero/illegal machine instruction/etc.
- Timer interrupt
  - Generated by a timer within the CPU; allows the OS to perform functions on a regular basis
  - Time-slice based scheduling, timed processes (f. ex. Check for updates or viruses)
- I/O interrupt
  - Generated by an I/O controller, to signal normal completion of an operation OR to signal a variety of error conditions
- Hardware interrupt
  - Generated by hardware input or a failure (power failure, memory parity error, etc.)
  - Maskable (can wait if higher priority interrupt occurs) or non-maskable (must be handled ASAP)

# Interrupt handler

- Interrupt handler is a process ran in kernel mode in OS
- Initiated every time an interrupt occurs
- Finds out the reason for interrupt (classification)
- Initiates actions for handling the situation
  - Desired interrupt (normal program/timer/I/O) → switch the process
  - Undesired interrupt → what went wrong, how to recover?
- In order to enable continuation of the interrupted process later, its registry values must be stored:
  - Program counter, program status word & other registers used by the process
- Note! Interrupt handling can be interrupted, too – this risk must be addressed
  - For example, a hardware failure during file write leads to a file corruption hazard

# Short I/O wait vs. long I/O wait

- Short I/O wait = the interrupt is handled right after it occurs
- Long I/O wait = the interrupt is handled when the next I/O request arrives
- When long I/O wait is used, there can only be 1 I/O request in line at a time
- “Microwave example”



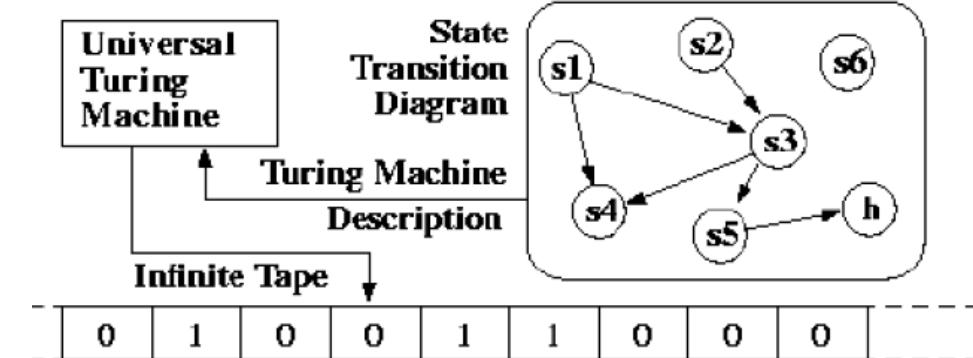
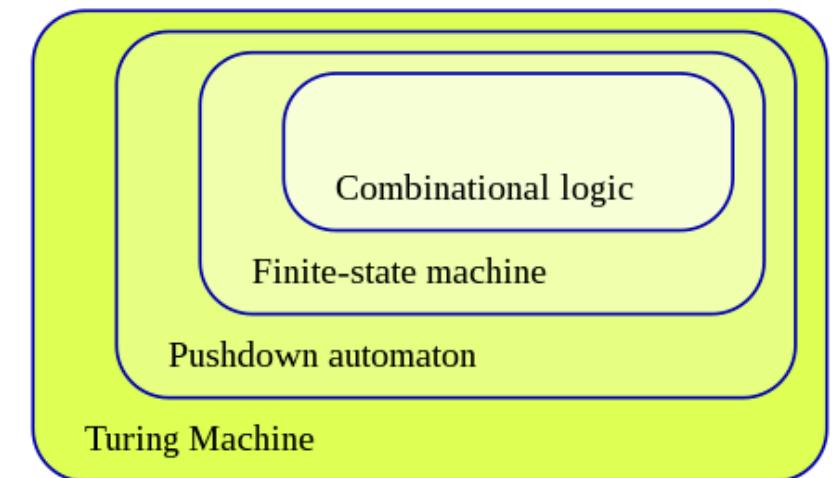
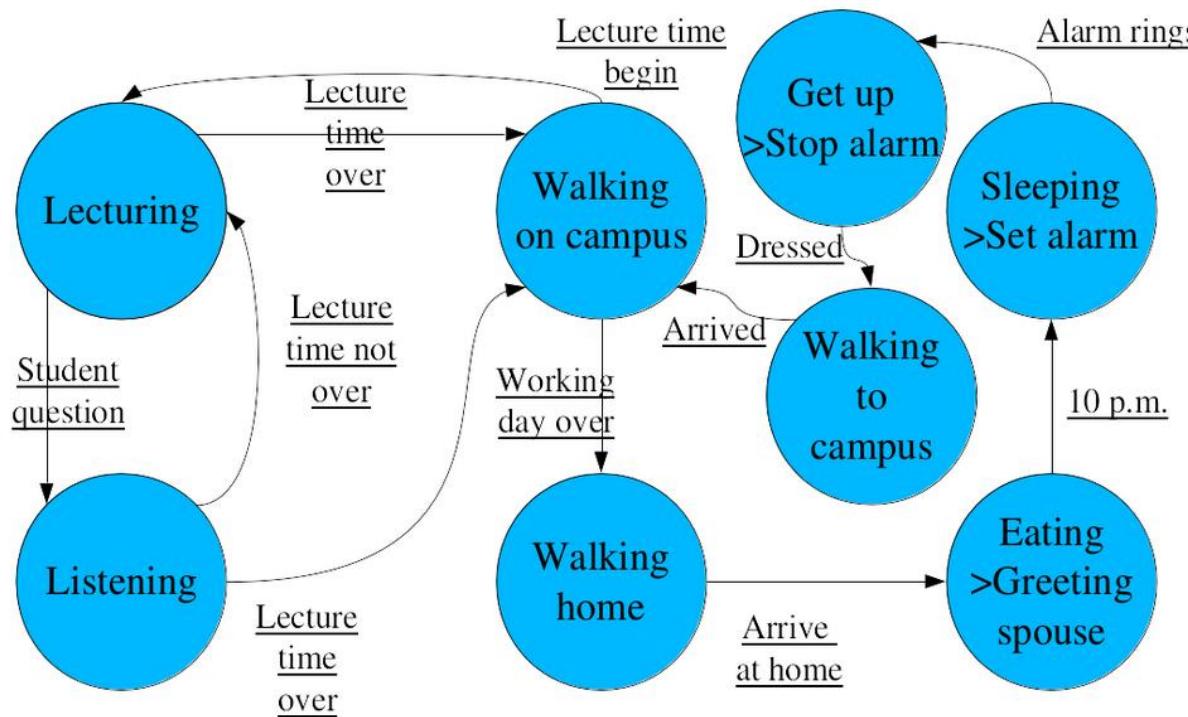
✖ = interrupt occurs during course of execution of user program

Stallings: Operating Systems, 9e, Prentice Hall

# Thank you for listening!

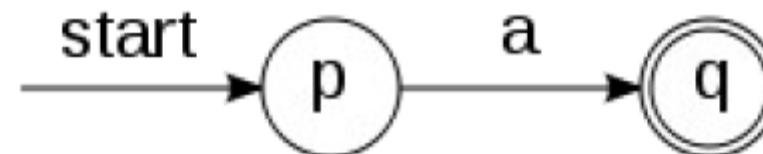


# 8. Automata and Turing machines



# Finite state machine

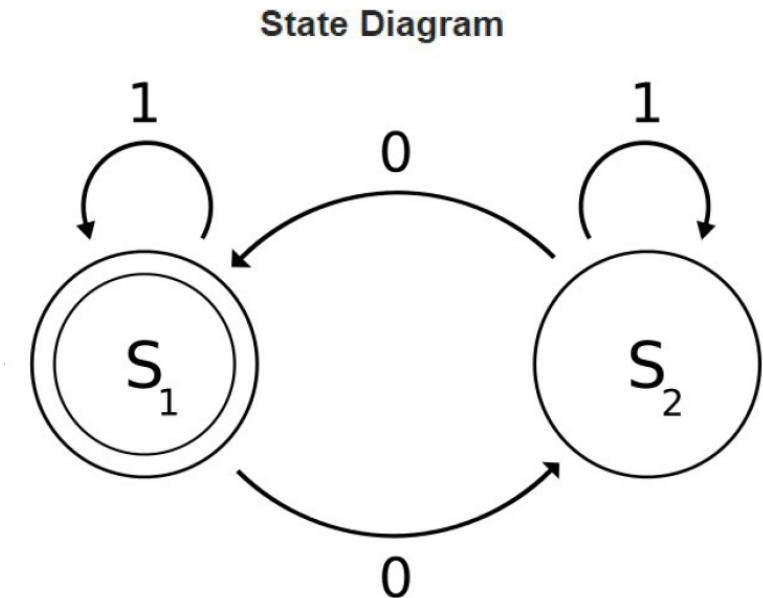
- We already encountered state machines when we discussed the grammars and lexing phase in compiling; let's dive a bit deeper there now
- A *finite state machine* is a way to model a task, language or data as a group of states and transitions between them
  - An *automaton* of one kind
  - Commonly presented in the form of a state diagram
  - Automaton processes the input one symbol at a time
  - Initial state(s) are represented by input arrows
  - States are circled, transitions are shown as arrows from one state to another
  - Accept (end) states are presented by double circles (or output arrows)



Example of a simple finite state machine  
p = start state  
a = transition ( $p \times a \rightarrow q$ )  
q = accept state

# State transition table

- Transitions between states can be represented by a *state transition table*
- Current states as rows, input symbols as columns
  - Table cell value tells the next state
- This notation has some weaknesses, though:
  - Initial state has not been marked in any way
  - No info on which states are accept states
- Needs improvement!



**State Transition Table**

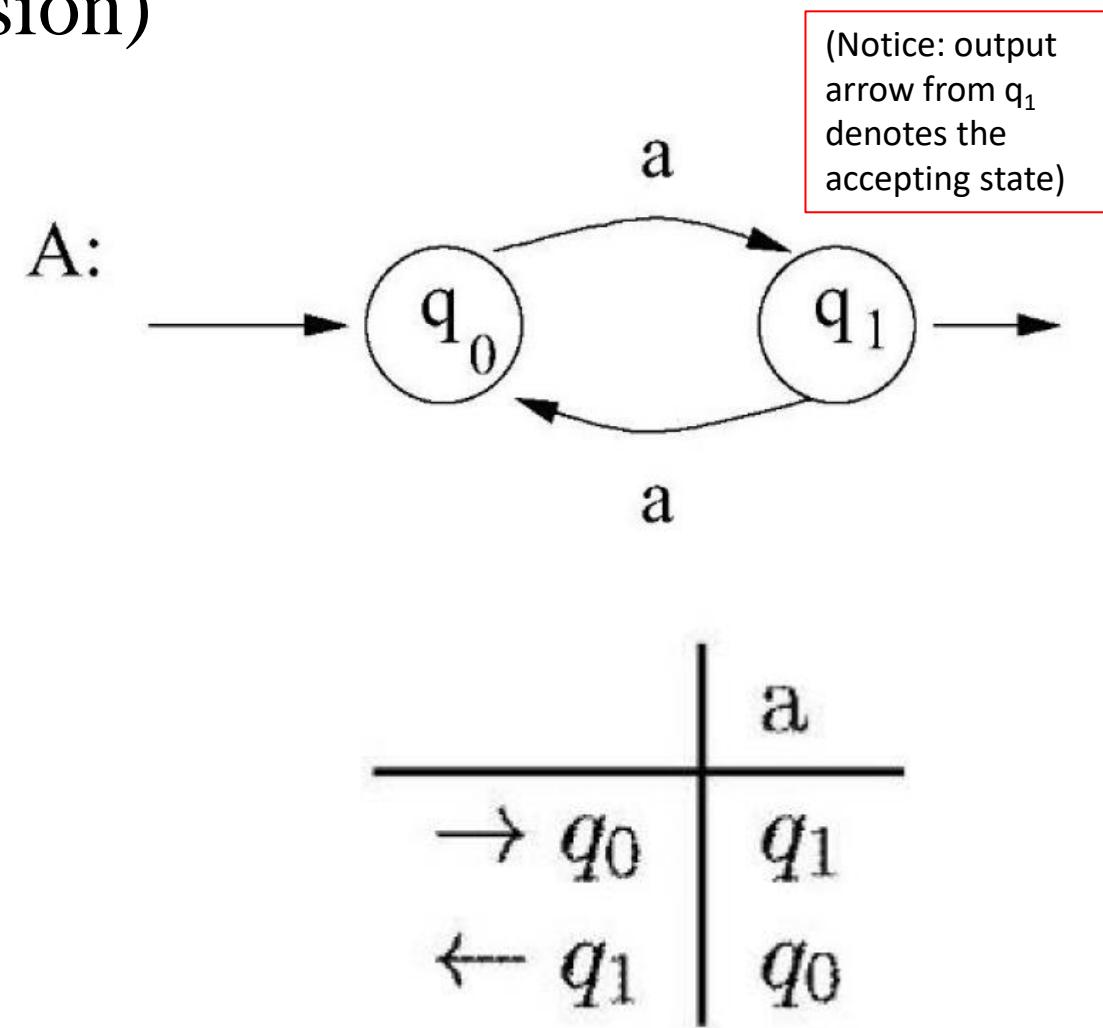
Input	1	0
State		
S1	S1	S2
S2	S2	S1

# State transition table (improved version)

- Automaton that accepts an input which consists of an odd number of a's
- Mathematically speaking:

$$\{aa^{2n} \mid n \geq 0\}$$

- Now the state transition table holds all information that is needed:
  - Start state is marked with a rightwards arrow
  - Accept state(s) are marked with a leftwards arrow  
(in some notations, also an asterisk (\*) is used)



# Types of automata

- An automaton can be *deterministic* or *non-deterministic*:
  - Deterministic = the state transitions are unambiguous – there is only one possible transition for each symbol
  - Non-deterministic = more than one possible transition in some state for at least one symbol
- Non-deterministic automaton must make guesses, so it needs to have an “escape route” in case it makes a bad guess
- An automaton can also be *finite* or *infinite*:
  - Finite = there is a finite amount of possible states
  - Infinite = amount of possible states is not limited
- Usually finite automata work with finite input strings; finite automata that can handle infinite inputs are called  $\omega$ -automata

# Deterministic finite-state automaton (DFA)

- A *deterministic finite-state automaton* (DFA) is defined by a 5-tuple  $(Q, \Sigma, \delta, q_0, F)$ :
  - $Q$  = a finite group of states
  - $\Sigma$  = the alphabet of the language
  - $\delta$  = a transition function that specifies the transitions  $Q \times \Sigma \rightarrow Q$  (or alternatively:  $\delta[Q], \Sigma = [Q]$ )
  - $q_0$  = initial state (Note:  $q_0 \in Q$ , naturally)
  - $F$  = group of accept states (Note:  $F \subseteq Q$ , naturally)
- DFA accepts an input string if reading it leads from initial state to accept state
  - If reading the string doesn't end in an accepting state, the string is not accepted
- If there is no transition for some character of the string, the input is disqualified
  - Note! A different situation than "not accepted"!
  - Results in an error and termination of the process
  - How can the automaton recover from the error?

# Grammar definition using an automaton

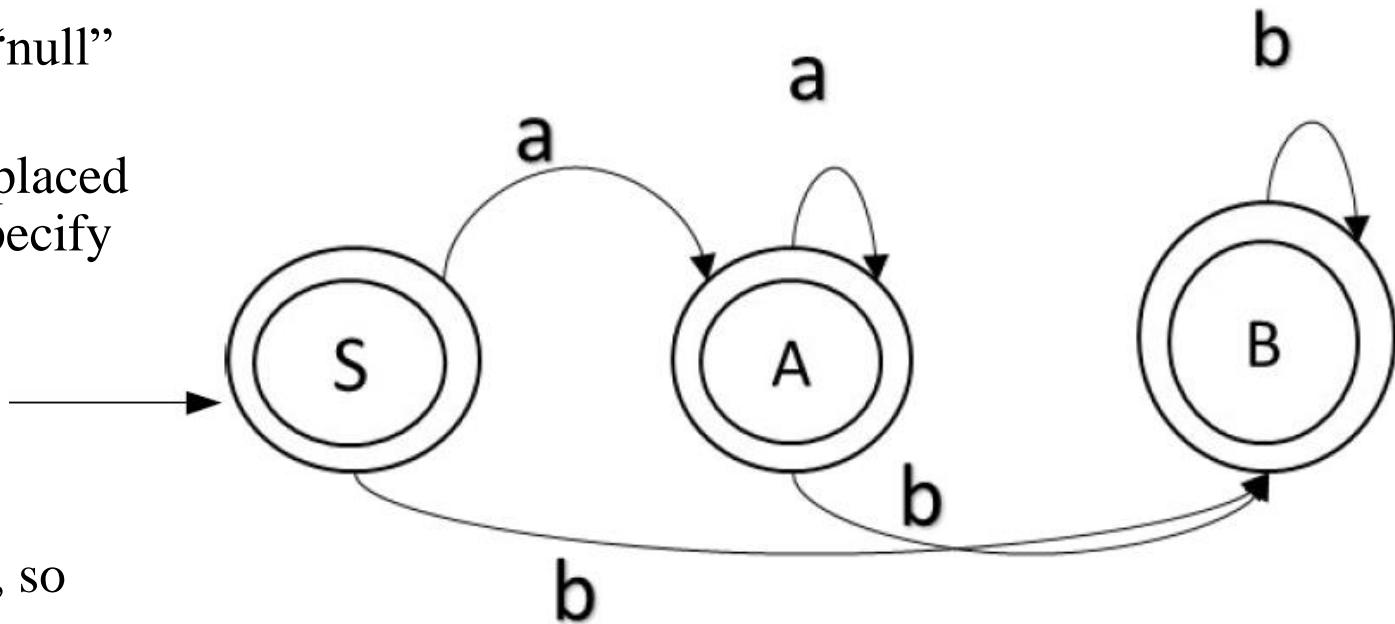
- An automaton can be used to define a grammar of a language
- Simple example:

- States:  $Q = \{S, A, B\}$
- Alphabet:  $\Sigma = \{a, b, \lambda\}$  ( $\lambda$  is a “null” symbol)
- Productions tell what can be replaced by which, so the productions specify the transitions

$$\delta = S \times a \rightarrow A, S \times b \rightarrow B, \\ A \times a \rightarrow A, A \times b \rightarrow B, B \times b \rightarrow B$$

- Initial state  $q_0 = S$
- Here all states are accept states, so  $F = Q = \{S, A, B\}$

$$S \rightarrow AB \\ A \rightarrow aA \mid \lambda \\ B \rightarrow Bb \mid \lambda$$

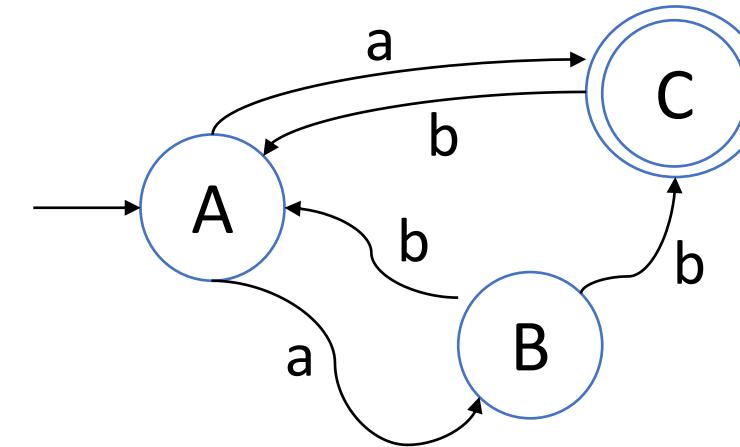


# From NFA to DFA

- It is common that a problem is, in many cases, easier to approach by constructing a non-deterministic finite automaton (NFA)
- An NFA is problematic to write into a program, though, because the automaton should be able to recover from bad guesses
- We can convert all NFAs to DFAs using subset construction
- A  $k$ -state NFA can always be converted to a (max.)  $2^k$ -state DFA
  - In many cases, the DFA will simplify and have less states
- Conversion in a nutshell:
  - Create a transition table for the NFA
  - If some transition has multiple state options, consider this state combination a new state
  - Create a new transition table for the DFA (derive the transitions of new states)
- The resulting DFA can be simplified by deleting unreachable states
- Examples here: <https://www.javatpoint.com/automata-conversion-from-nfa-to-dfa>

## NFA to DFA: Example 1

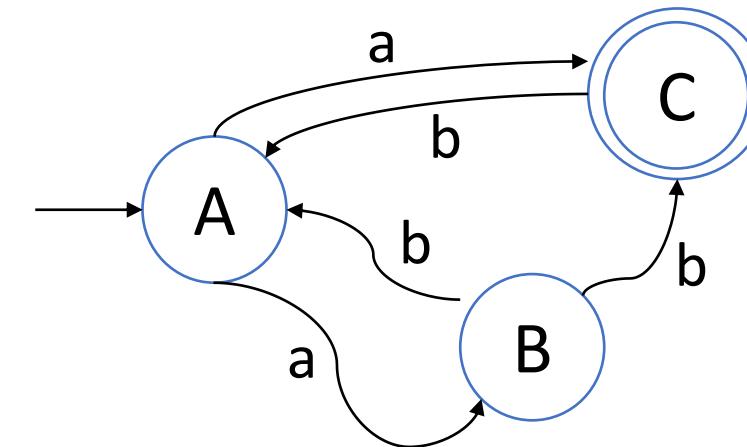
- Convert the following NFA to a DFA.



## NFA to DFA: Example 1

- Convert the following NFA to a DFA.
- Transition table for the NFA:

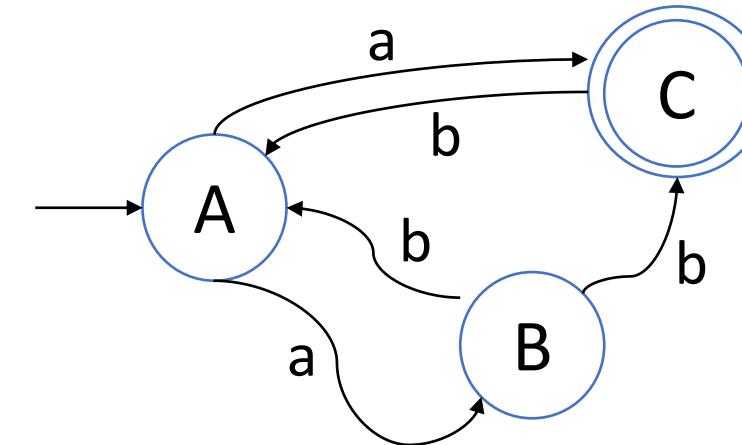
	a	b
$\rightarrow A$	B,C	-
B	-	A,C
*C	-	A



## NFA to DFA: Example 1

- Convert the following NFA to a DFA.
- Transition table for the NFA:

	a	b
$\rightarrow A$	B,C	-
B	-	A,C
*C	-	A



- Transitions for new states:
 
$$\delta'[B, C], a = \delta[B], a \cup \delta[C], a = \emptyset \cup \emptyset = \emptyset$$

$$\delta'[B, C], b = \delta[B], b \cup \delta[C], b = [A, C] \cup [A] = [A, C]$$

$$\delta'[A, C], a = \delta[A], a \cup \delta[C], a = [B, C] \cup \emptyset = [B, C]$$

$$\delta'[A, C], b = \delta[A], b \cup \delta[C], b = \emptyset \cup [A] = [A]$$

# NFA to DFA: Example 1

- Convert the following NFA to a DFA.
- Transition table for the NFA:

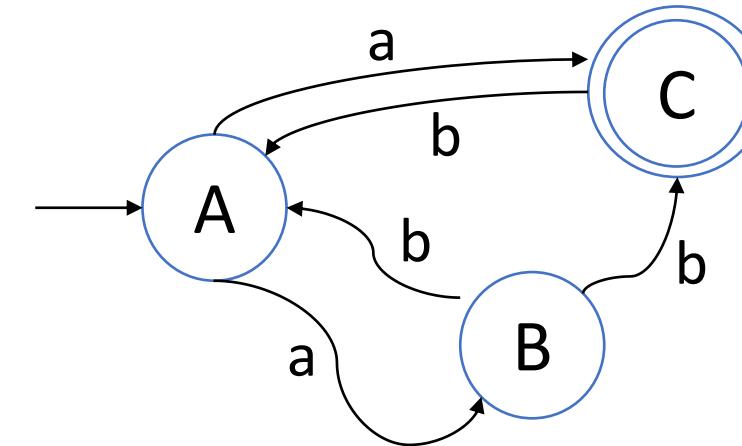
	a	b
$\rightarrow A$	B,C	-
B	-	A,C
*C	-	A

- Transitions for new states:
 
$$\delta'[B, C], a = \delta[B], a \cup \delta[C], a = \emptyset \cup \emptyset = \emptyset$$

$$\delta'[B, C], b = \delta[B], b \cup \delta[C], b = [A, C] \cup [A] = [A, C]$$

$$\delta'[A, C], a = \delta[A], a \cup \delta[C], a = [B, C] \cup \emptyset = [B, C]$$

$$\delta'[A, C], b = \delta[A], b \cup \delta[C], b = \emptyset \cup [A] = [A]$$

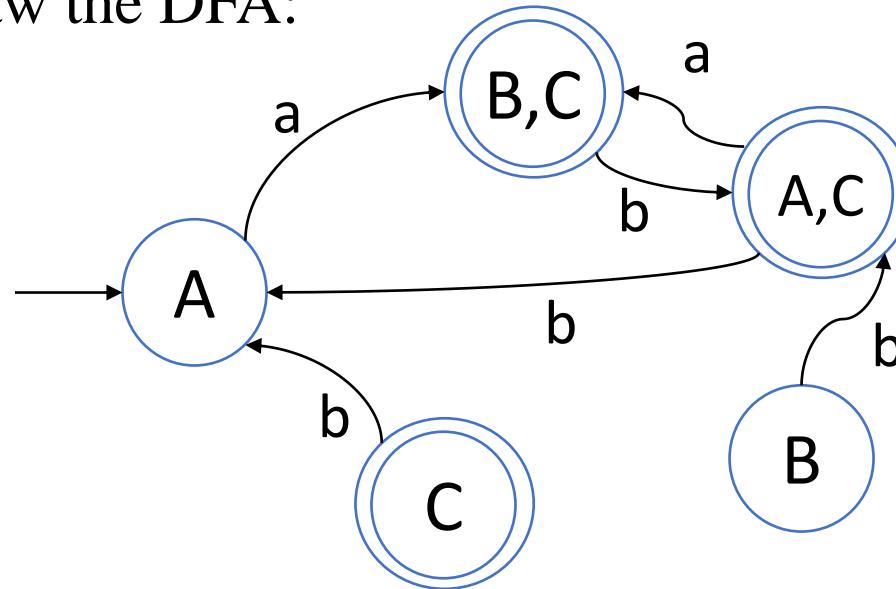


- Transition table for the DFA:
  - [B,C] and [A,C] are also accept states, because they contain accept state C

	a	b
$\rightarrow A$	B,C	-
B	-	A,C
*C	-	A
*B,C	-	A,C
*A,C	B,C	A

## NFA to DFA: Example 1

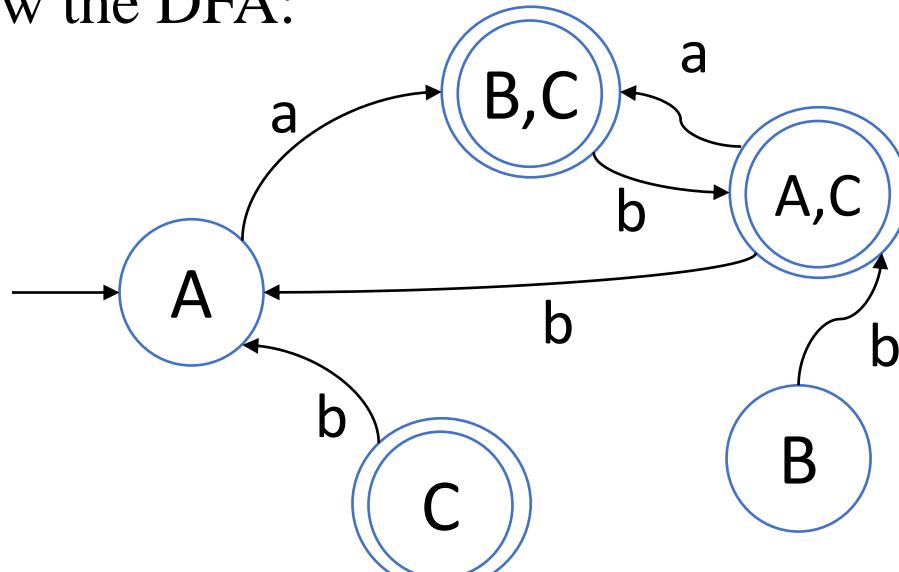
- Draw the DFA:



	a	b
$\rightarrow A$	B,C	-
B	-	A,C
*C	-	A
*B,C	-	A,C
*A,C	B,C	A

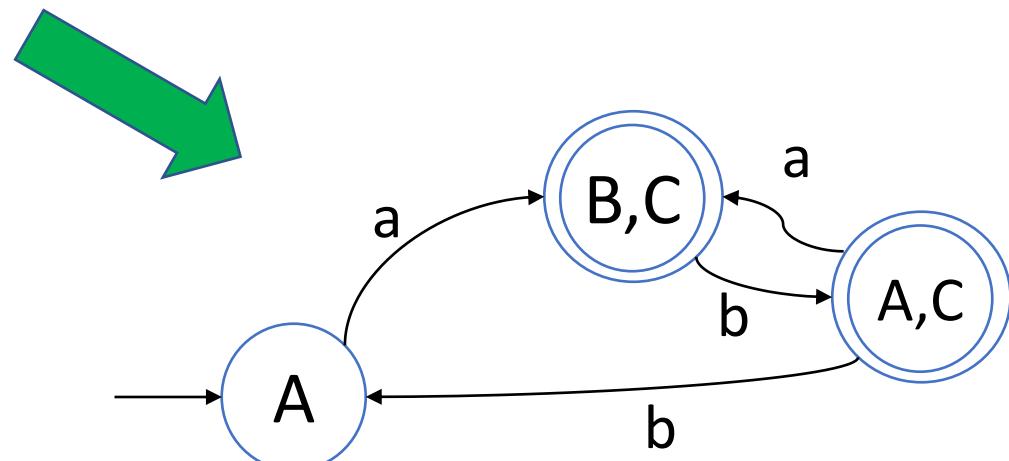
# NFA to DFA: Example 1

- Draw the DFA:



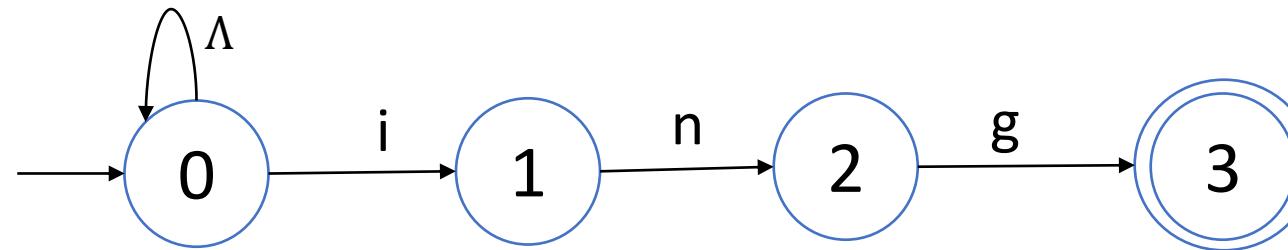
- No transitions can take us from the initial state to B or C, so these states are unreachable → can be discarded:

	a	b
→A	B,C	-
B	-	A,C
*C	-	A
*B,C	-	A,C
*A,C	B,C	A



## NFA to DFA: Example 2

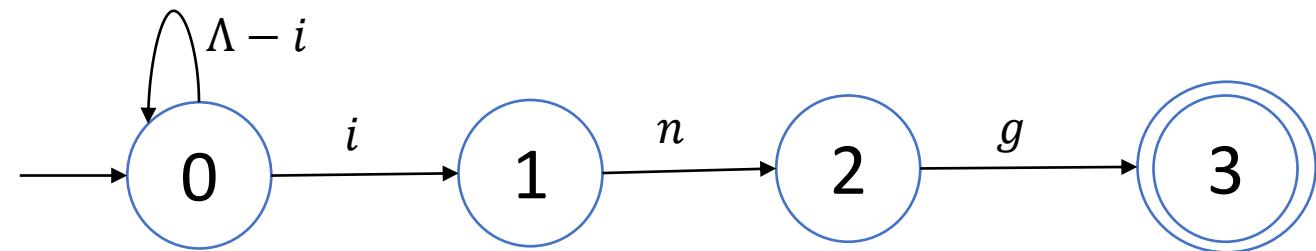
- Sometimes we can formulate a DFA from an NFA by using more “common sense”
- Suppose we want to create an automaton which identifies words that end in suffix “–ing”. For this kind of a problem, an NFA can be constructed rather easily:



- Here, the symbol  $\Lambda$  means “any character”
- One would think that this automaton wouldn’t work, because when it encounters an “i”, it can go either to 0 or to 1 – but it does; the automaton goes through all possible paths until the word has been either a) identified or b) deemed unidentifiable.
- How could we construct this into a DFA that does exactly what we want?

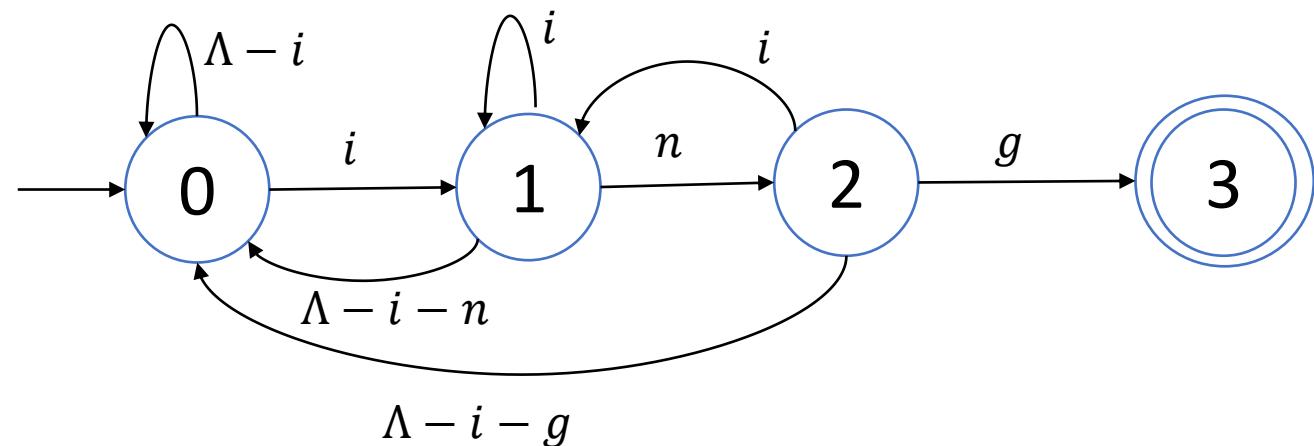
## NFA to DFA: Example 2

- First modification is easy: let's remove  $i$  from “all characters”
  - Now the automaton is already a DFA! But does it work the way we want?
  - No – for example, “shipping” would cause an error (the first “ $i$ ” it encounters isn’t the one that belongs to the “-ing” suffix)



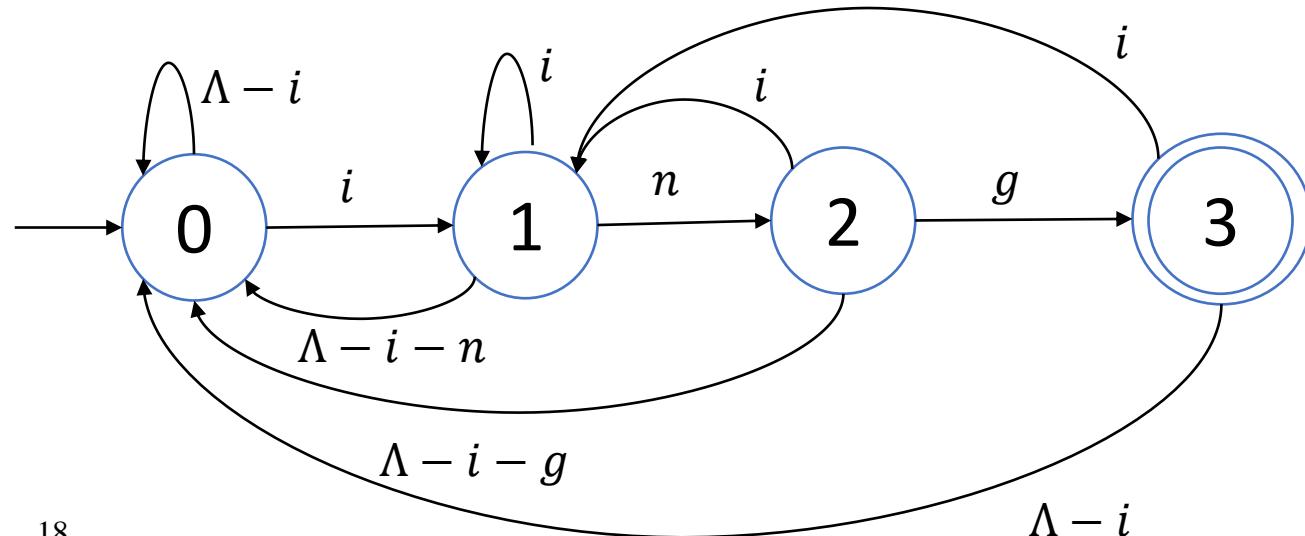
## NFA to DFA: Example 2

- First modification is easy: let's remove  $i$  from “all characters”
  - Now the automaton is already a DFA! But does it work the way we want?
  - No – for example, “shipping” would cause an error (the first “ $i$ ” it encounters isn’t the one that belongs to the “-ing” suffix)
- Second modification: enable going backwards in the automaton
  - Does it work now? No, because it doesn’t detect whether the word *ends* in -ing. (For example, “ringer” or “upbringing” would be problematic – depending on the setup of the automaton.)



# NFA to DFA: Example 2

- First modification is easy: let's remove  $i$  from “all characters”
  - Now the automaton is already a DFA! But does it work the way we want?
  - No – for example, “shipping” would cause an error (the first “ $i$ ” it encounters isn’t the one that belongs to the “-ing” suffix)
- Second modification: enable going backwards in the automaton
  - Does it work now? No, because it doesn’t detect whether the word *ends* in -ing. (For example, “ringer” or “upbringing” would be problematic – depending on the setup of the automaton.)
- 3<sup>rd</sup> modification
  - Back loops from state 3
- Now it works!



# String search using regular expressions

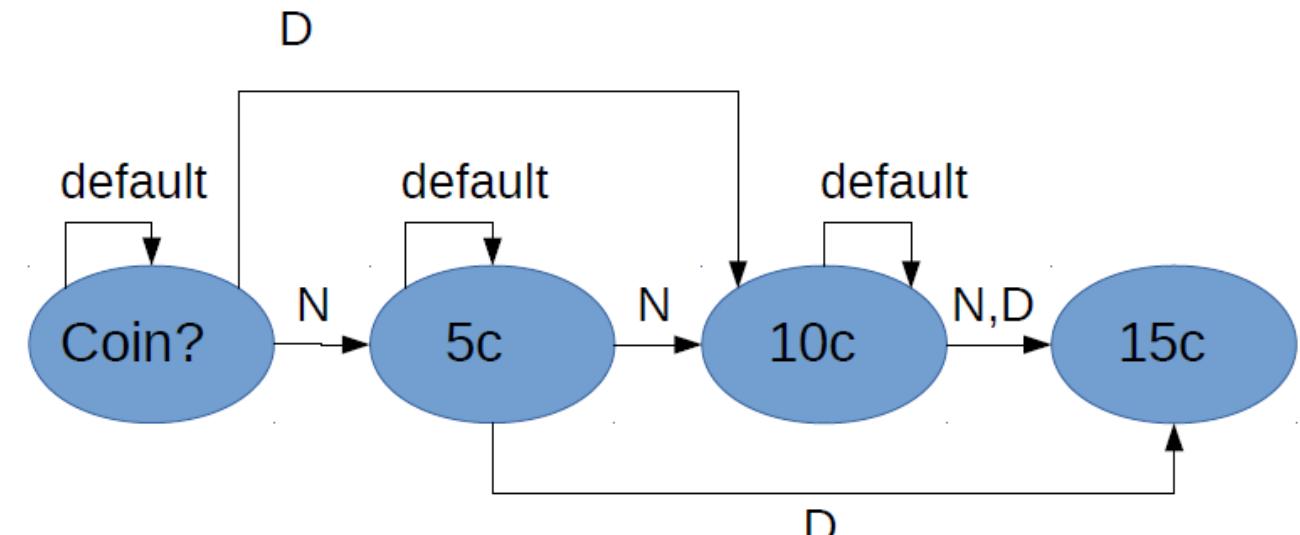
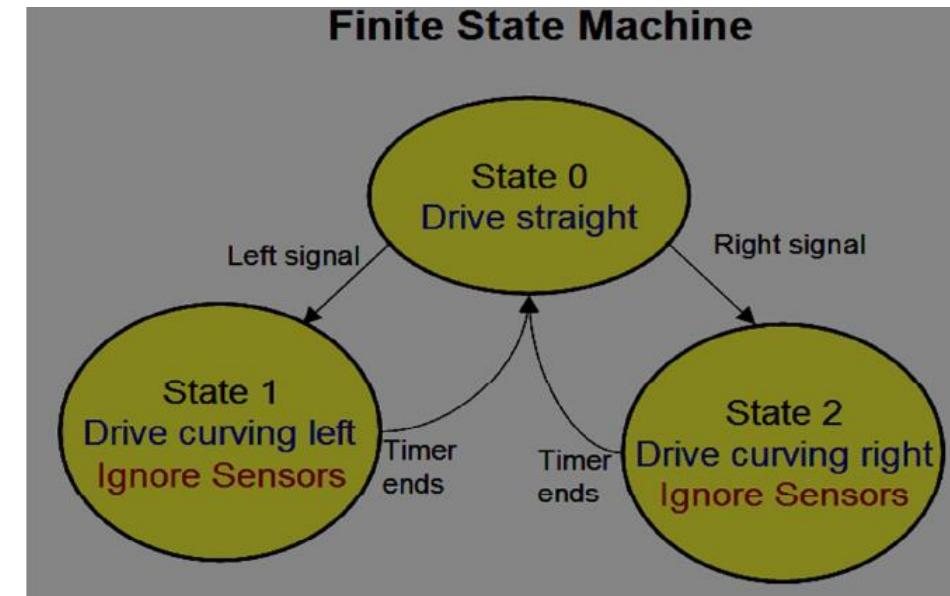
- In previous examples we used automata to search for strings that fulfilled our given conditions
- Instead of using an automaton, we can describe these strings using *regular expressions* (regex) – the most effective way to represent any language
- We've already encountered some of these before, but let's dive a bit deeper now:
  - Asterisk:  $a^*$  = 0 to infinite number of concurrent a's
  - Plus:  $a^+$  = 1 to infinite number of concurrent a's
  - Question mark:  $ab?c$  = zero or one b's (so, "abc" and "ac" are accepted)
  - Wildcard (dot):  $a.b$  = the dot can be any character
  - Boolean OR:  $a|b$  = a or b
  - Parentheses:  $(abb|bab)a$  = "abba" OR "baba"
  - Curly braces:  $a\{3,5\}$  = 3 to 5 pcs of concurrent a's

# Regular expressions and automata

- Regular expressions are the simplest way to define a search string
- All regular expressions can be converted to an automaton
- This conversion is actually done by first creating an NFA from the regular expression and then converting that to a DFA
- Regular expressions are widely used in programming language grammars, some search engines & text processors (“find & replace”)
  - Not Google, though - since the larger the database, the more resource-intensive their use is
- Hence, knowing how to use these is a nice skill to have
- Really good site to practice: <https://www.regexpal.com/>
  - Allows the user to give a test string and then check in real-time how many matches the given regular expression produces

# Practical automata examples

- Lane assist in a car
  - Turn signal changes state
  - In states 1 and 2, lane detection sensors are ignored
- In the old days there were vending machines that sold Coca-Cola for 15 cents a bottle (nowadays inflation has caught up)
  - default = no money added
  - D = dime (10 cents)
  - N = nickel (5 cents)
  - Accept state = 15c
  - Note! No change given

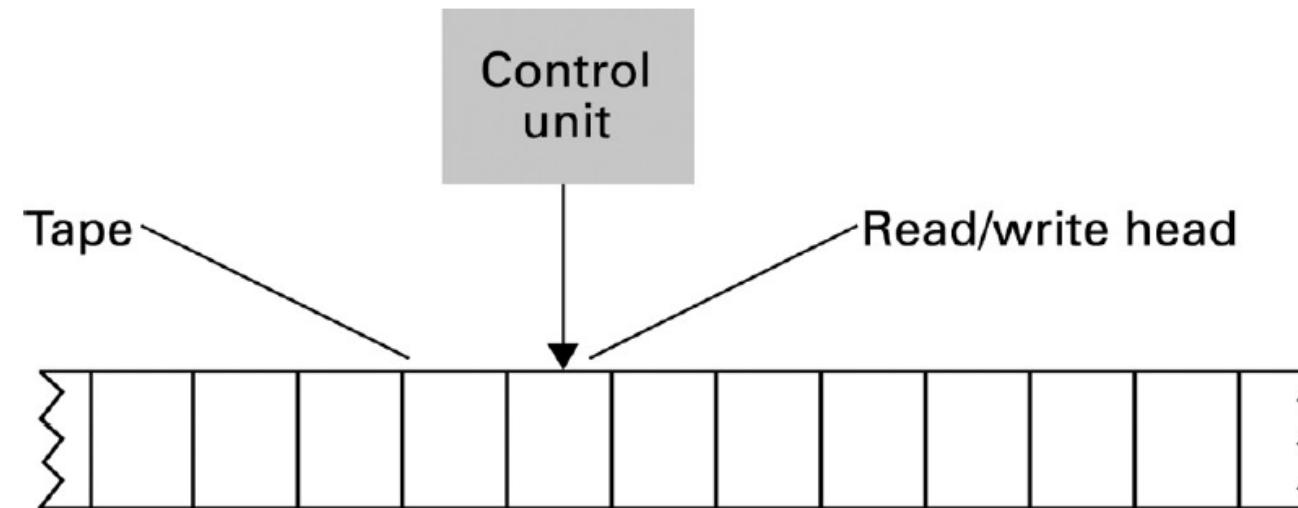


# Turing machine

- State machines were quite primitive automata; they didn't have memory, so the transition was only dependent on the current state and next input character
- If we expand our automaton by adding a memory, we end up in a primitive model of a computer called *Turing machine* (according to Alan Turing, 1936)
  - Actually there's a "middle version" called pushdown automata (PDA) in between these; a PDA doesn't have memory, but it employs a stack
- Memory of a Turing machine is a tape, which can be both read and written on
- This is done via a read/write head, which can read the tape one character at a time
  - After operation, read/write head can be moved one step at a time to the left or right
- The write-possibility enables us to also modify the input – while in a DFA, the input could only either be accepted or rejected

# Structure of a Turing machine

- A Turing machine is a simple mathematical model of computation
  - Tape is infinitely long and it has been divided to cells
  - A tape cell can contain any symbol from the symbol group (alphabet of the machine)
  - Control unit reads and/or writes the symbols on tape cell by cell
  - Control unit can move the read/write head left (L), right (R) or stay (S) in place



# How Turing machine works

- Calculation always starts from initial state and ends in final state
- Calculation consists of steps made by the control unit
- A step consists of
  - Reading the cell on the tape
  - Writing on the cell on the tape
  - Moving the read/write head (or tape – some authors think that the tape moves)
  - Changing the state
- Early computers were basically Turing machines
  - Memory could only be used in specific order
- Nowadays modern computers use RAM, which can be read or written in any order
  - So, modern computers are more agile than Turing machines
- Still, a Turing machine can perform all calculations that a computer does!

# Definition of a Turing machine

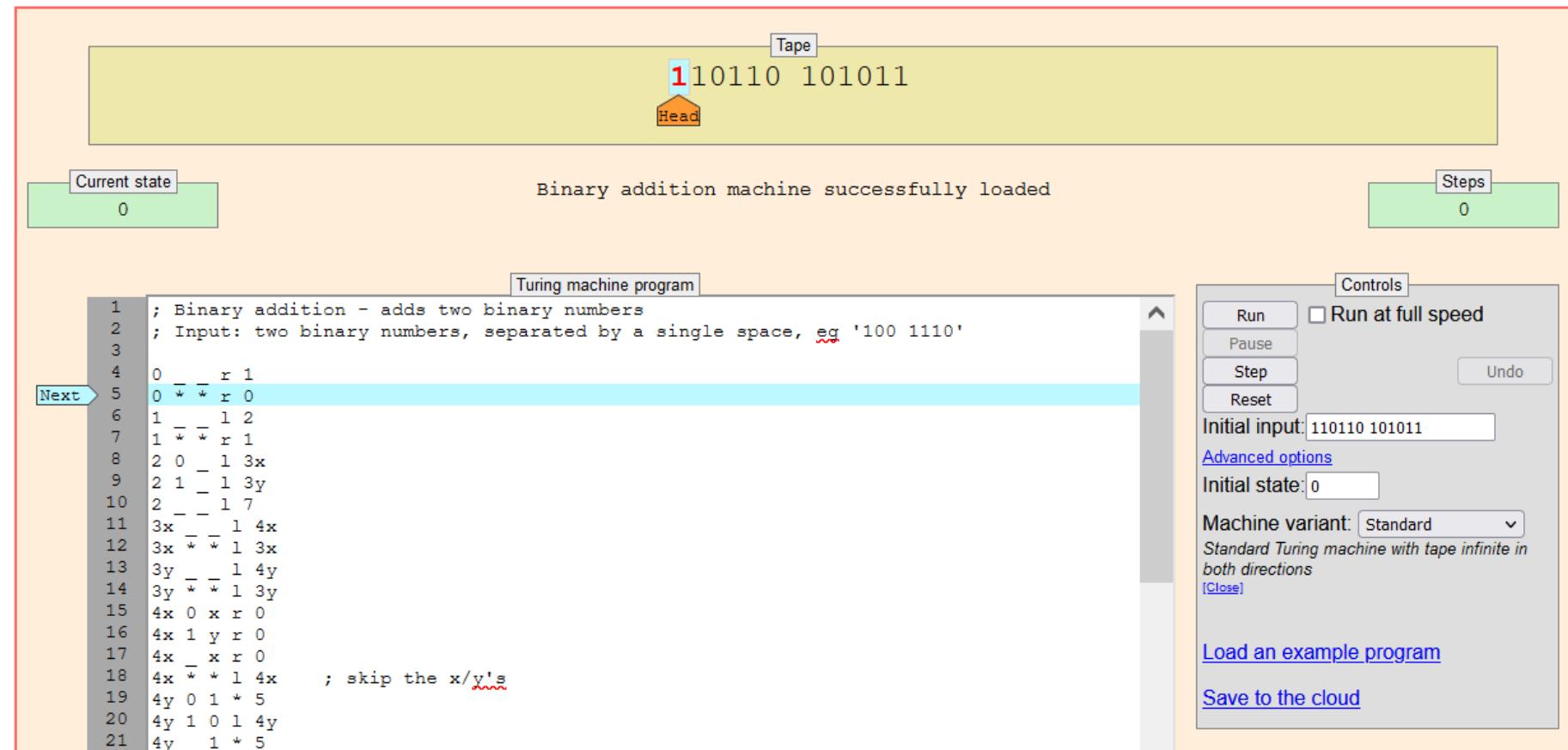
- A Turing machine  $M$  is defined by a 7-tuple  $M = (Q, T, I, \delta, b, q_0, q_F)$ :
  - $Q$  = a finite group of states
  - $T$  = a group of tape symbols
  - $I$  = the set of input symbols (Note:  $I \subseteq T$ )
  - $\delta$  = a transition function that specifies the transitions  $Q \times T \rightarrow Q \times T \times \{L, S, R\}$
  - $b$  = blank symbol
  - $q_0$  = initial state (Note:  $q_0 \in Q$ , naturally)
  - $q_F$  = set of final states (Note:  $q_F \subseteq Q$ , naturally)
- Example of a transition:  $q_1, x \rightarrow q_2, y, L$ 
  - Meaning: if we're currently in state  $q_1$  and the symbol on tape is  $x$
  - Procedure in this case: write symbol  $y$  on tape, move the read/write head left, switch to state  $q_2$

# Morphett Turing simulator

- Behavior of different Turing machines can be investigated using a Turing simulator
- There are many of these online, but this Morphett's version seems like the best:  
<https://morphett.info/turing/turing.html>
- Learn how to use this simulator by trying out some of the example programs
- Some things to notice:
  - In Morphett, state transformations syntax is different - it specifies the transitions in order: (current state, symbol on tape, symbol written on tape, head move direction, new state to enter)
  - So, for example, the previous transition  $q_1, x \rightarrow q_2, y, L$  in Morphett would be  $q_1\ x\ y\ L\ q_2$  (separated only by one spacebar)
  - Default initial state is 0, but this can be changed from "Advanced options"
  - Head position can be specified using an asterisk (\*) in the input

# Morphett Turing simulator

- Use “Step” button in order to see step by step how the machine proceeds
- On the right machine shows the step number
- Try different inputs!



# Example 1

- What does this Turing machine do? (Starts from right side of input)

Current state	Current cell content	Value to write	Direction to move	New state to enter
START	*	*	Left	ADD
ADD	0	1	Right	RETURN
ADD	1	0	Left	CARRY
ADD	*	*	Right	HALT
CARRY	0	1	Right	RETURN
CARRY	1	0	Left	CARRY
CARRY	*	1	Left	OVERFLOW
OVERFLOW	*	*	Right	RETURN
RETURN	0	0	Right	RETURN
RETURN	1	1	Right	RETURN
RETURN	*	*	No move	HALT

# Example 1

- What does this Turing machine do? (Starts from right side of input)
  - After a couple of simulations, we see that it adds 1 to the input (binary addition:  $101 \rightarrow 110$ )

Current state	Current cell content	Value to write	Direction to move	New state to enter
START	*	*	Left	ADD
ADD	0	1	Right	RETURN
ADD	1	0	Left	CARRY
ADD	*	*	Right	HALT
CARRY	0	1	Right	RETURN
CARRY	1	0	Left	CARRY
CARRY	*	1	Left	OVERFLOW
OVERFLOW	*	*	Right	RETURN
RETURN	0	0	Right	RETURN
RETURN	1	1	Right	RETURN
RETURN	*	*	No move	HALT

## Example 2

- What does this Turing machine do? (starts from right side of input)

$$\begin{aligned}
 M &= (Q, T, I, \delta, b, q_0, q_f) \\
 Q &= \{1, 2, 3, \text{H}\} \\
 T &= \{0, 1, \_\} \\
 I &= \{0, 1\} \\
 b &= \_ \\
 q_0 &= 1 \\
 q_f &= \text{H}
 \end{aligned}$$

$$\begin{aligned}
 q_i, x \rightarrow q_j, y, \{\text{L}, \text{S}, \text{R}\} \\
 \delta = & \begin{array}{l} 1, \_ \rightarrow 1, \_, \text{L} \\ 1, 0 \rightarrow 2, 0, \text{L} \\ 1, 1 \rightarrow 2, 1, \text{L} \\ 2, \_ \rightarrow 3, \_, \text{R} \\ 2, 0 \rightarrow 2, 0, \text{L} \\ 2, 1 \rightarrow 2, 1, \text{L} \\ 3, \_ \rightarrow \text{H}, \_, \text{S} \\ 3, 0 \rightarrow 3, 0, \text{R} \\ 3, 1 \rightarrow 3, 1, \text{R} \end{array}
 \end{aligned}$$

## Example 2

- What does this Turing machine do? (starts from right side of input)
  - Nothing much – it seems to search for the nearest blank space that has a number on its right side, and then comes back
  - Note: tape symbols are not altered in any transition!

$$\begin{aligned}
 M &= (Q, T, I, \delta, b, q_0, q_f) \\
 Q &= \{1, 2, 3, \text{H}\} \\
 T &= \{0, 1, \_\} \\
 I &= \{0, 1\} \\
 b &= \_ \\
 q_0 &= 1 \\
 q_f &= \text{H}
 \end{aligned}$$

$$\begin{aligned}
 q_i, x \rightarrow q_j, y, \{\text{L}, \text{S}, \text{R}\} \\
 \delta = & \begin{array}{l} 1, \_ \rightarrow 1, \_, \text{L} \\ 1, 0 \rightarrow 2, 0, \text{L} \\ 1, 1 \rightarrow 2, 1, \text{L} \\ 2, \_ \rightarrow 3, \_, \text{R} \\ 2, 0 \rightarrow 2, 0, \text{L} \\ 2, 1 \rightarrow 2, 1, \text{L} \\ 3, \_ \rightarrow \text{H}, \_, \text{S} \\ 3, 0 \rightarrow 3, 0, \text{R} \\ 3, 1 \rightarrow 3, 1, \text{R} \end{array}
 \end{aligned}$$

## Example 3

- What does this Turing machine do? (starts from left side of input)

$$\begin{aligned}
 M &= (Q, T, I, \delta, b, q_0, q_f) \\
 Q &= \{1, 2, 3, 4, 5, 6, \text{H}\} \\
 T &= \{0, 1, \_\} \\
 I &= \{0, 1\} \\
 b &= \_ \\
 q_0 &= 1 \\
 q_f &= \text{H}
 \end{aligned}$$

$$\begin{aligned}
 \delta = & 1, \_ \rightarrow \text{H}, \_, \text{S} \\
 & 1, 0 \rightarrow 2, 0, \text{S} \\
 & 1, 1 \rightarrow 2, 0, \text{S} \\
 & 2, \_ \rightarrow 5, \_, \text{L} \\
 & 2, 0 \rightarrow 3, 0, \text{L} \\
 & 2, 1 \rightarrow 4, 1, \text{L} \\
 & 3, \_ \rightarrow 6, 0, \text{R} \\
 & 3, 0 \rightarrow 6, 0, \text{R} \\
 & 3, 1 \rightarrow 6, 0, \text{R} \\
 & 4, \_ \rightarrow 6, 1, \text{R} \\
 & 4, 0 \rightarrow 6, 1, \text{R} \\
 & 4, 1 \rightarrow 6, 1, \text{R} \\
 & 5, \_ \rightarrow \text{H}, \_, \text{S} \\
 & 5, 0 \rightarrow \text{H}, 0, \text{S} \\
 & 5, 1 \rightarrow \text{H}, 0, \text{S} \\
 & 6, \_ \rightarrow 2, \_, \text{R} \\
 & 6, 0 \rightarrow 2, 0, \text{R} \\
 & 6, 1 \rightarrow 2, 1, \text{R}
 \end{aligned}$$

## Example 3

- What does this Turing machine do? (starts from left side of input)
  - Needs a couple of simulations to understand
  - Machine treats the input as a number with a sign (two's complement)
  - It takes the absolute value of the input and then multiplies it by two

$$\begin{aligned}
 M &= (Q, T, I, \delta, b, q_0, q_f) \\
 Q &= \{1, 2, 3, 4, 5, 6, H\} \\
 T &= \{0, 1, \_\} \\
 I &= \{0, 1\} \\
 b &= \_ \\
 q_0 &= 1 \\
 q_f &= H
 \end{aligned}$$

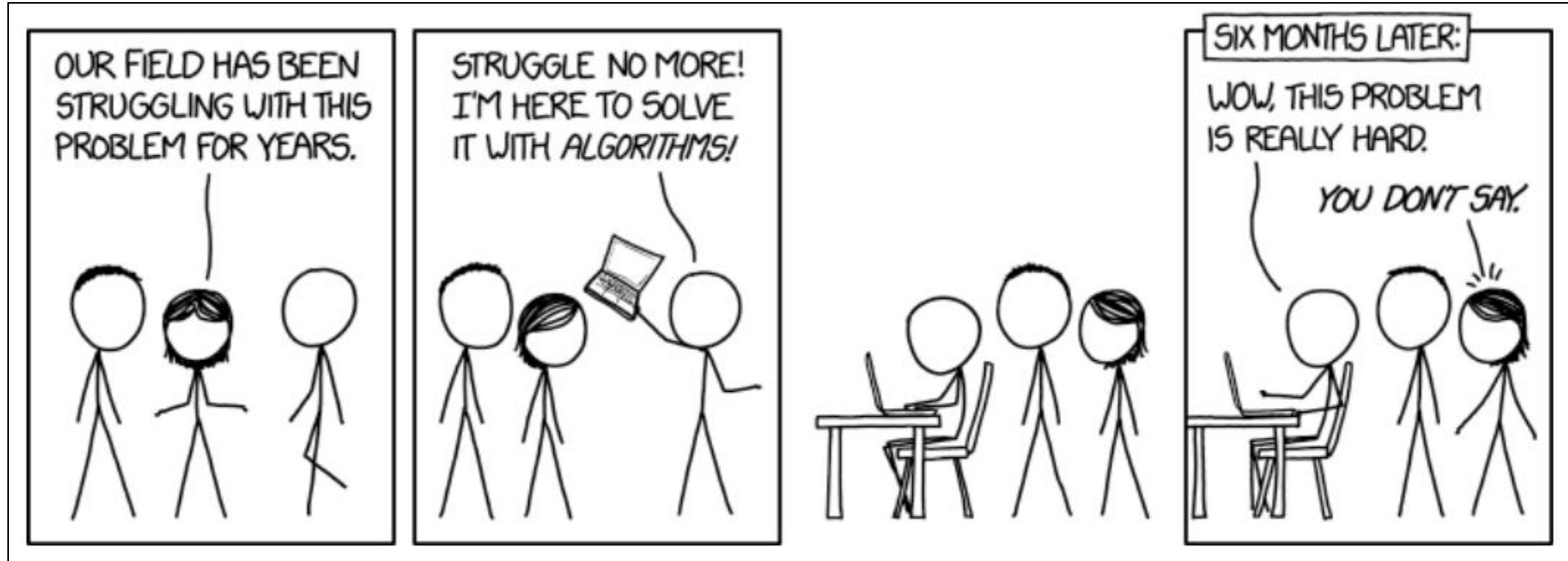
$$\begin{aligned}
 \delta = & 1, \_ \rightarrow H, \_, S \\
 & 1, 0 \rightarrow 2, 0, S \\
 & 1, 1 \rightarrow 2, 0, S \\
 & 2, \_ \rightarrow 5, \_, L \\
 & 2, 0 \rightarrow 3, 0, L \\
 & 2, 1 \rightarrow 4, 1, L \\
 & 3, \_ \rightarrow 6, 0, R \\
 & 3, 0 \rightarrow 6, 0, R \\
 & 3, 1 \rightarrow 6, 0, R \\
 & 4, \_ \rightarrow 6, 1, R \\
 & 4, 0 \rightarrow 6, 1, R \\
 & 4, 1 \rightarrow 6, 1, R \\
 & 5, \_ \rightarrow H, \_, S \\
 & 5, 0 \rightarrow H, 0, S \\
 & 5, 1 \rightarrow H, 0, S \\
 & 6, \_ \rightarrow 2, \_, R \\
 & 6, 0 \rightarrow 2, 0, R \\
 & 6, 1 \rightarrow 2, 1, R
 \end{aligned}$$

**Try these yourself! All these 3 examples have been converted to Morphett code in the .txt file that can be found in Moodle. Just copy & paste the Turing machine in Morphett and experiment with different inputs!**

# Thank you for listening!

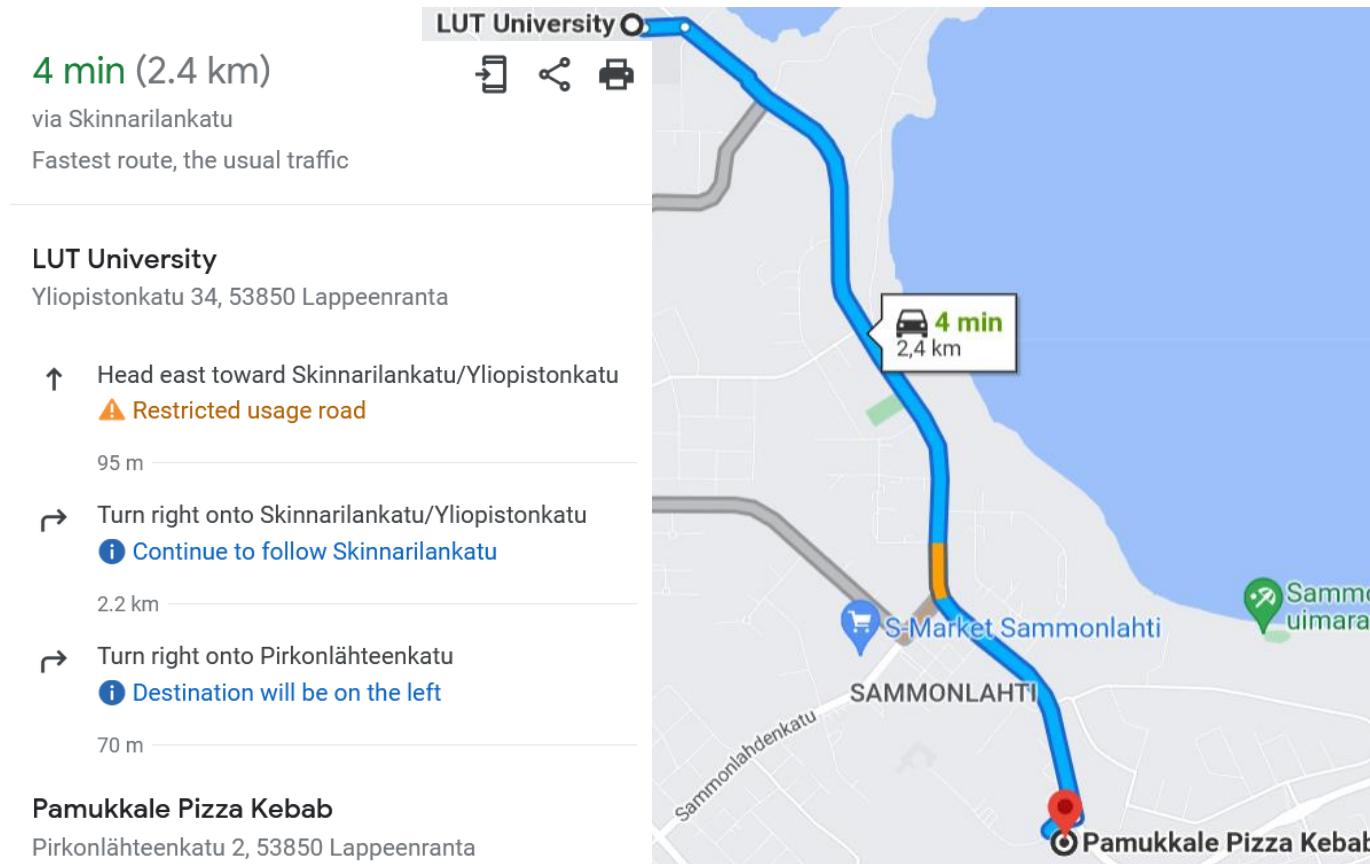


# 9. Algorithms



# Everyday “algorithms”

- Are these detailed enough?



# What is an algorithm?

- An algorithm is basically a set of instructions
- Broadly speaking, algorithms are not limited to computing – for example, also following real-world instructions can be considered algorithms:
  - Recipes (cooking instructions of different meals)
  - Furniture assembly (instructions on how to put together a shelf)
- In these real-world applications, the detail level of instructions is usually vague (or at least related to the assumed skill level of the reader)
- Computers require the instructions in an extremely precise form
- Formal definition of an algorithm in computer science: “*An algorithm is an ordered set of unambiguous, executable steps that defines a terminating process*”
  - Termination requirement is because of the field, not because of mathematics

# Church-Turing theses

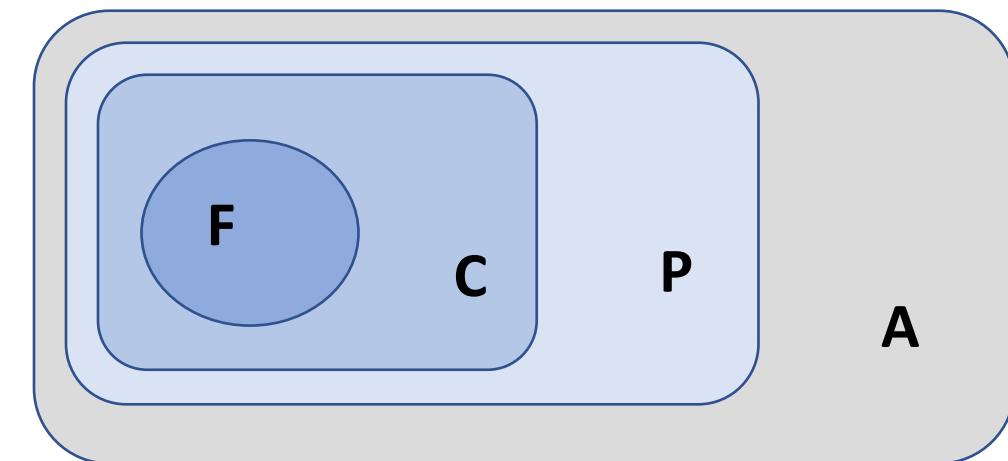
- Alonzo Church & Alan Turing proposed their own definitions for an algorithm
- Eventually, they came up with their (and some other authors') definitions being just as good, so they formulated the definition to two theses:
  - 1) All known reasonable definitions of an algorithm are equivalent to each other
  - 2) Any reasonable definition invented in the future will be equivalent to the previous ones
- First one has been proven true, second one can't be proven but is accepted true
- Consequences of the theses:
  - All algorithms can be executed by a Turing machine
  - Every algorithm that can be executed by some computer can also be executed on all other computers – i.e. all computers are equivalent to each other and can perform the same tasks!
- All computers are not created equal, though; there are major differences in efficiency and execution time

# Solvability of problems

- For centuries it was thought, that all problems in the world can be solved using some algorithm
  - Solvability = has somebody discovered an algorithm for the problem or not?
- Only in the 1930s it was proven that there exists problems that *can't* be solved using an algorithm
  - David Hilbert's “Entscheidungsproblem”, proven algorithmically unsolvable by Kurt Gödel’s “Incompleteness Theorem” in 1931 – and soon followed by others
  - Solvability = feature of the problem itself
- So, nowadays problems can be divided to three groups:
  - Solvable problems (there exists an algorithm which solves the problem for all inputs)
  - Algorithmically unsolvable problems (proven that there is no algorithm)
  - Problems, whose solvability is (yet) unknown

# Computability

- Instead of solvability, a better term would be to talk about *computability*
  - Takes into account the effectiveness of the solution: can it be found in reasonable time?
- Problems are classified based on their computability using the best algorithm:
  - Feasible problems (complexity is polynomial at max)
  - Computable problems (complexity can be exponential, but there is a finite number of options)
  - Partially computable problems (algorithm can find a solution if there is one, but can't reach a conclusion that there is no solution)
  - All problems
- Notice that smaller classes are subsets
- We'll return to these a bit later...



# Analysis of algorithms

- For many problems, there are several algorithms that can be used for solving
- Especially in these cases it is natural that we are interested in ways to compare the possible algorithms to each other – i.e. analyze their performance
- Analysis of algorithms revolves around two properties:
  - *Complexity* (Note! Means calculational complexity – not “how difficult it looks like”)
  - *Correctness*
- Naturally, we would want all our algorithms to be 100% correct, so more emphasis is put on the first one
- Correctness is a big criteria when we’re programming the algorithm, though:
  - Advanced algorithms are often harder to prove correct
  - Decreased calculational complexity usually comes at the price of increased “visual complexity”, so it also leads to increased risk of programming mistakes

# Complexity

- Different problems and algorithms set various requirements for execution
- Most important computer resources are
  - Execution time (number of CPU cycles)
  - Memory use
  - System (for example, number of processors in parallel computation)
- Usually the best algorithm is considered as the one that uses the least resources
- ...which in most cases translates to quickest execution time
  - In some database applications, memory use might be a limiting factor
- Demand for resources is dependent on the size of the input – denoted by  $n$ 
  - Size  $n$  = length of input list / number of nodes on graph / number of cells in a table etc.

# Time complexity $T(n)$ and space complexity $M(n)$

- Time complexity of an algorithm is given as a function of input size –  $T(n)$
- Unit of time complexity is not any time measure, because the execution time is very much dependent on the computer used for calculation
- Unit is therefore the number of elementary operations needed for solution
- If we are interested in the use of memory resources, the measure for this is space complexity  $M(n)$
- Unit of space complexity is the total amount of memory space used by the algorithm

# Asymptotic complexity

- The exact number of elementary operations is not interesting – it's the order of magnitude that's important
  - Small differences between algorithms may even out due to other reasons (data transfer etc.)
- In asymptotic complexity, only the most dominant term is considered
- Also, constant factors are ignored, because asymptotically they don't change the situation that much
- Examples:
  - If our algorithm has time complexity  $T(n) = n^2 + 5n + 3$ , its asymptotic (time) complexity is simply  $T(n) \sim n^2$
  - If our algorithm has time complexity  $T(n) = 3n \log n + 100n - 5$ , its asymptotic (time) complexity is simply  $T(n) \sim n \log n$

# Asymptotic complexity

- Why can we ignore the other terms? See the table below and notice how small is the difference between  $T(n) = n^2$  and  $T(n) = n^2 + 5n + 3$  for large input sizes

$\log_2 n$	$n$	$n^2$	$n^2 + 5n + 3$	$2^n$
0	1	1	9	2
n. 3	10	100	153	1024
n. 7	100	10 000	10 503	$n \cdot 10^{30}$
n. 10	1 000	1 000 000	1 005 003	...
n. 13	10 000	100 000 000	100 050 003	
n. 17	100 000	10 000 000 000	10 000 500 003	
n. 20	1 000 000	1 000 000 000 000	1 000 005 000 003	

# Theta and “Big-Oh”

- The time an algorithm needs in order to find a solution is not constant even for same size inputs; it can be dependent on pure “luck”
- Therefore, when we talk about complexity, three different cases can be considered:
  - Best-case scenario
  - Average-case scenario
  - Worst-case scenario
- The actual number of elementary operations that must be performed will be something ranging from best-case to worst-case scenario
- The average-case complexity is usually referred to as theta –for example  $\Theta(n^2)$
- Usually we are most interested in the worst-case scenario that gives the upper bound for complexity; this is often referred as “Big-Oh” – for example  $O(n^2)$ 
  - NOTE!  $T(n)$  is a function – theta and Big-Oh are not!

$$T(n) = n^3 - 4n \sim n^3 \rightarrow O(n^3)$$

# Time complexity classification

- Complexities can be classified in seven groups – from best to worst:
  - Constant  $T(n) \sim 1$
  - Logarithmic  $T(n) \sim \log n$
  - Linear  $T(n) \sim n$
  - Linear-logarithmic  $T(n) \sim n \log n$
  - Polynomial  $T(n) \sim n^c$
  - Exponential  $T(n) \sim c^n$
  - Factorial  $T(n) \sim n!$
- Note: the base of the logarithm is not important (usually considered 2)
- Great comparison chart & tables here: <https://www.bigocheatsheet.com/>

# Execution time vs. complexity

- Generally speaking, anything below polynomial complexity gives reasonable execution times – polynomial is “doable if must”, but exponential is useless
  - Table values calculated for a CPU of 1 MHz

$n \setminus T(n)$	$\log_2 n$	$n$	$n \log_2 n$	$n^2$	$n^3$	$2^n$
10	3 µs	10 µs	30 µs	100 µs	1 ms	1 ms
100	7 µs	100 µs	700 µs	10 ms	1 s	$410^{22}$ a
1000	10 µs	1 ms	10 ms	1 s	17 min	$10^{226}$ a
$10^4$	13 µs	10 ms	130 ms	100 s	12 d	
$10^5$	17 µs	100 ms	1,7 s	2,8 h	32 a	
$10^6$	20 µs	1 s	20 s	12 d	$310^4$ a	
$10^7$	23 µs	10 s	4 min	3,2 a	$310^7$ a	
$10^8$	27 µs	100 s	44 min	317 a	$310^{10}$ a	
$10^9$	30 µs	17 min	8,3 h	$310^4$ a	$310^{13}$ a	

# Example: Search algorithms

- Suppose we want to search for an item in a list – say, a student from the university student record (using either a name – or student number, which eliminates the possibility of namesakes)
- Our university has roughly 10 000 students, but there are probably another 10 000 old students in the records, so let's say the database has 20 000 items
- If we're searching for a certain student number from the records, the size of our problem is  $n = 20\,000$
- Let's compare two search algorithms

Student number	Student name	...
0024022	Smith, Chad	
0025035	Frusciante, John	
0025594	Kiedis, Anthony	

# Example: Search algorithms – sequential search

- Simplest search algorithm is called *sequential search*:
  - 1) Start from the beginning of the list
  - 2) Compare the value to the one we're searching; if it's the desired one, search is done
  - 3) If it's not, move to the next item on the list and repeat step 2
- Very surefire way, but doesn't seem efficient
- For sequential search, complexities are the following:
  - Best-case scenario:  $T(n) \sim 1$  (desired item was the first one on the list, what a luck!)
  - Worst-case scenario:  $T(n) \sim n$  (desired item was the last one, or not on the list at all)
  - Average-case scenario:  $T(n) = \frac{1}{2}n \sim n^*$  (statistical odds)
- Conclusion: sequential search is  $\Theta(n)$  and  $O(n)$

\*Warning: Don't be fooled by this example - finding the average-case complexity is often much harder and requires advanced statistical calculations.

# Example: Search algorithms – binary search

- Another, more powerful algorithm is called *binary search*:
  - 1) Initiation: potential range = the entire list
  - 2) Examine the middle entry of potential range. If the item is the desired one, search is done
  - 3) If the middle entry is greater than the desired item, reduce the potential range to entries on the left side of the middle entry. If it's less, reduce the potential range to entries on the right.
  - 4) Go back to step 2
- This algorithm requires that the list is already ordered! (Now we can assume it is.)
- For binary search, complexities are the following:
  - Best-case scenario:  $T(n) \sim 1$  (desired item was the middle one on the list, what a luck!)
  - Worst-case scenario:  $T(n) \sim \log n$  (potential range evolution:  $20\ 000 \rightarrow 10\ 000 \rightarrow 5000 \rightarrow 2500 \rightarrow 1250 \rightarrow 625 \rightarrow$  etc, so after  $x$  steps our potential range will be reduced to 1)
- Conclusion: sequential search is  $O(\log n)$  (and also  $\Theta(\log n)^*$ )

$$\begin{aligned}\frac{n}{2^x} &= 1 \\ n &= 2^x \\ x &= \log n\end{aligned}$$

\*Calculation of this is skipped.

# Example: Search algorithms - comparison

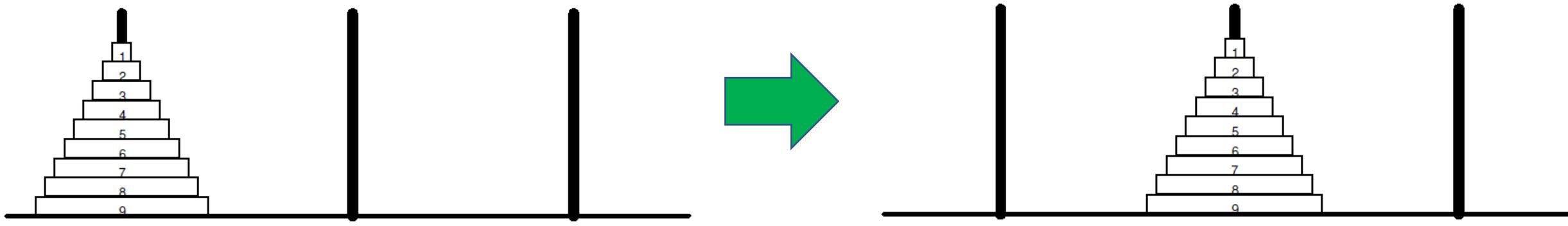
- If our computer is able to perform one comparison in 5 ms, the worst-case search times for algorithms would be:
  - Sequential search –  $20\,000 \cdot 5\text{ ms} = 100\,000\text{ ms} = 100\text{ s} = 1\text{ minute }40\text{ seconds}$
  - Binary search –  $\log(20\,000) \cdot 5\text{ ms} \approx 15 \cdot 5\text{ ms} = 75\text{ ms} = 0.075\text{ seconds}$
- If you're a secretary, the waiting time of the sequential search is painfully long
- The binary search, on the other hand, feels like an instant response
- The effect is even greater if we extend this idea to larger problem size – consider, for example, finding a person by social security number in China ( $n = 1.4$  billion):
  - Sequential search –  $1\,400\,000\,000 \cdot 5\text{ ms} = 7\,000\,000\text{ s} \approx 81\text{ days}(!!!)$
  - Binary search –  $\log(1\,400\,000\,000) \cdot 5\text{ ms} \approx 31 \cdot 5\text{ ms} = 155\text{ ms} = 0.155\text{ seconds} – \text{basically, still an instant response!}$

## Example: Towers of Hanoi

- This problem is based on an old legend of a Hindu temple where, according to the legend, were three posts, and in one of them 64 disks stacked from largest to smallest in diameter (two other posts are initially empty)
- The holy mission of the priests was to move all disks from the original post to another post – but with following limitations:
  - Only one disk can be moved at a time.
  - After a move, all disks have to be on a post – they can't be “moved to the side” to wait
  - At all times, disks have to be in size order – so, from largest to smallest
- The solution to the problem consists of surprisingly many moves
- Let's perform an illustration for the algorithm using 9 disks (in order to get neater pictures):

## Example: Towers of Hanoi

- Pictured: initial state and the desired state



- Let's break the solution down to smaller parts in a recursive manner:
  - First we have to move all disks except the bottom one to the 3rd post (right)
  - Then we move the bottom disk to the goal post
  - Then we move all disks on the 3<sup>rd</sup> post to the goal post

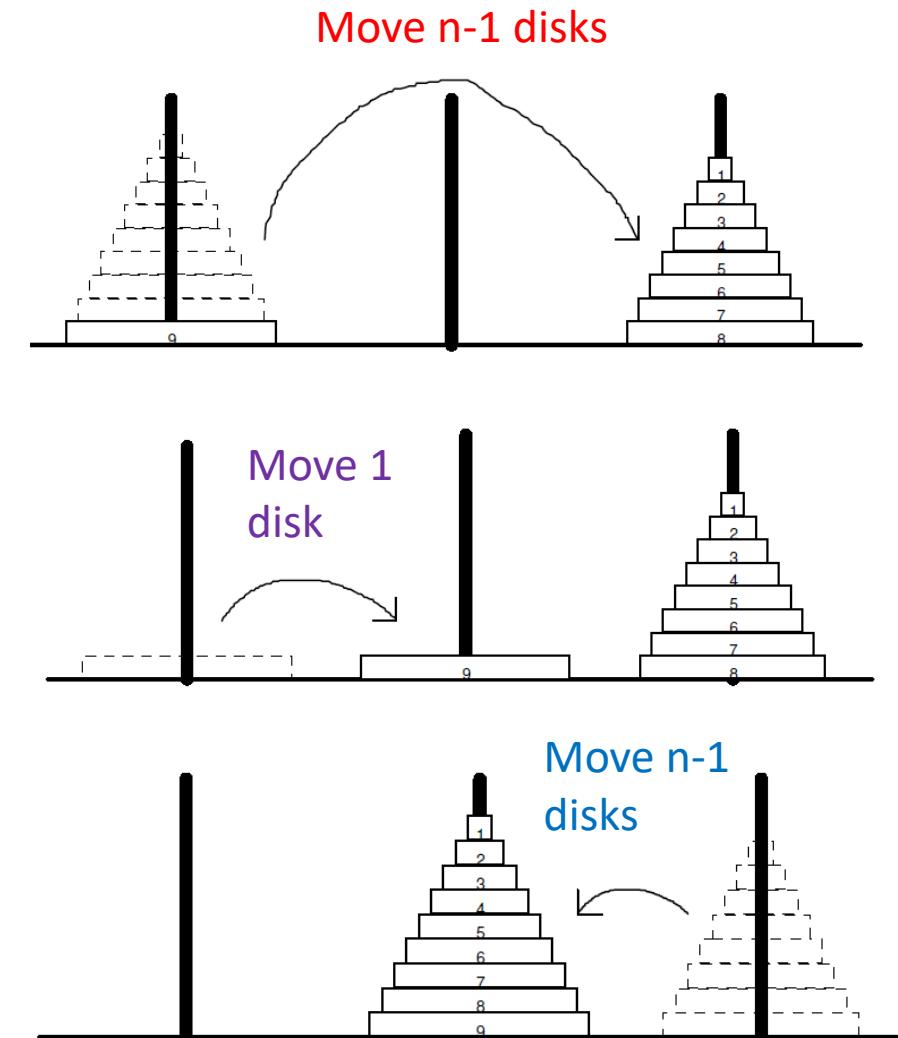
# Example: Towers of Hanoi

- Illustration of these 3 steps:
- The complexity was originally  $T(n)$
- Now the complexity can be expressed as

$$T(n) = T(n - 1) + 1 + T(n - 1)$$

$$= 2T(n - 1) + 1$$

- This is a recursive algorithm. Let's continue!



# Example: Towers of Hanoi

- Continuing the recursion, we get
  - $T(n) = 2T(n - 1) + 1 = 2(2T(n - 2) + 1) + 1 = 4T(n - 2) + 2 + 1$  (recursion #2)
  - $T(n) = 4(2T(n - 3) + 1) + 2 + 1 = 8T(n - 3) + 4 + 2 + 1$  (recursion #3)
  - $T(n) = 8(2T(n - 4) + 1) + 4 + 2 + 1 = 16T(n - 4) + 8 + 4 + 2 + 1$  (recursion #4)
- Let's express the terms in powers of two:
  - $T(n) = 2^4T(n - 4) + 2^3 + 2^2 + 2^1 + 2^0$  (recursion #4)
- I think we can see a pattern in here! After  $n-1$  recursions we have simplified this to a problem that has only one disk. Let's link the powers to the  $n$ :
  - $T(n) = 2^{n-1}T(1) + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0$  (recursion #n-1)
- What is the complexity of  $T(1)$ ? Naturally it's 1, because moving 1 disk = 1 move
  - $T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0$

## Example: Towers of Hanoi

- Now, our complexity function is  $T(n) = 2^{n-1} + 2^{n-2} + 2^{n-3} + \dots + 2^1 + 2^0$
- Number of needed elementary operations could be calculated, but how about the classification? A simplified expression would be nice
- Discovery: this  $T(n)$  is a geometric series, where  $a_1 = 2^0$ ,  $q = 2$  and  $n = n$
- Sum of a geometric series-formula:

$$S_n = a_1 \frac{1 - q^n}{1 - q} \rightarrow T(n) = 2^0 \cdot \frac{1 - 2^n}{1 - 2} = 1 \cdot \frac{1 - 2^n}{-1} = 2^n - 1$$

- So, the problem is exponential in complexity:  $O(2^n)$
- Number of moves for 64 disk-case:  $2^{64} - 1 \approx 1.8447 \cdot 10^{19}$ 
  - A good animation (3...8 disks) here: <https://www.mathsisfun.com/games/towerofhanoi.html>

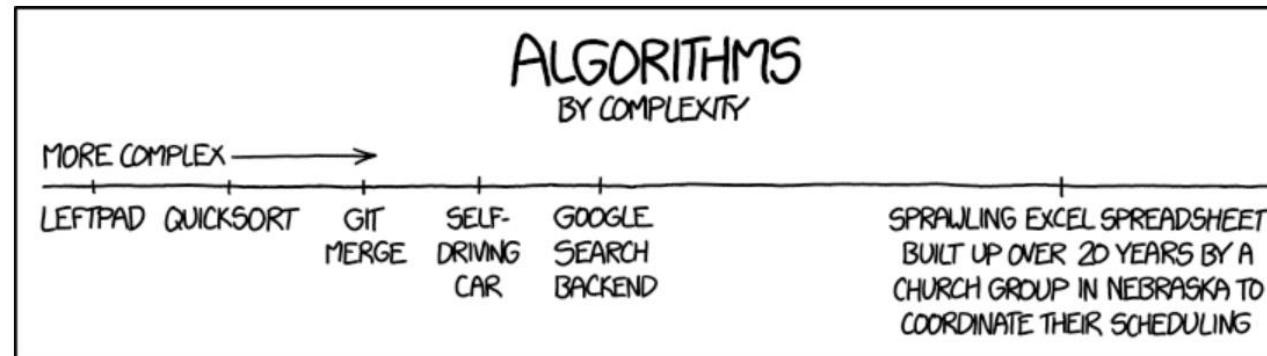
# Feasibility of problems

- As we noticed, the complexity of a problem has a huge impact on its computability
- For this reason, problems that can be calculated in polynomial time (i.e. there exists an algorithm that is  $O(n^c)$ , at maximum) are considered *feasible problems*
- Contrariwise, problems which are exponential (or greater) in complexity are considered *infeasible problems*
- Infeasibility can be a result of two things (or both):
  - Limited number of options, but very high number of moves
  - Limited number of moves, but very high number of options
- Examples of infeasible problems:
  - Towers of Hanoi, traveling salesman problem, playing chess...
  - Scheduling of timetables



# How to solve infeasible problems?

- It is not possible to produce complete solutions for infeasible problems
- This kind of problems can still be considered, if we compromise on perfectionism:
  - Limit analysis to special cases of the problem (that can be solved quickly)
  - Construct an algorithm that works quickly for average/most common inputs
  - Approximative algorithms (find a solution that's close)
  - Probabilistic algorithms (find *almost* always a correct solution)
  - Heuristic algorithms (user guides the decision process and discards options which are "bad" – in many cases, humans can detect these much earlier than a computer!)

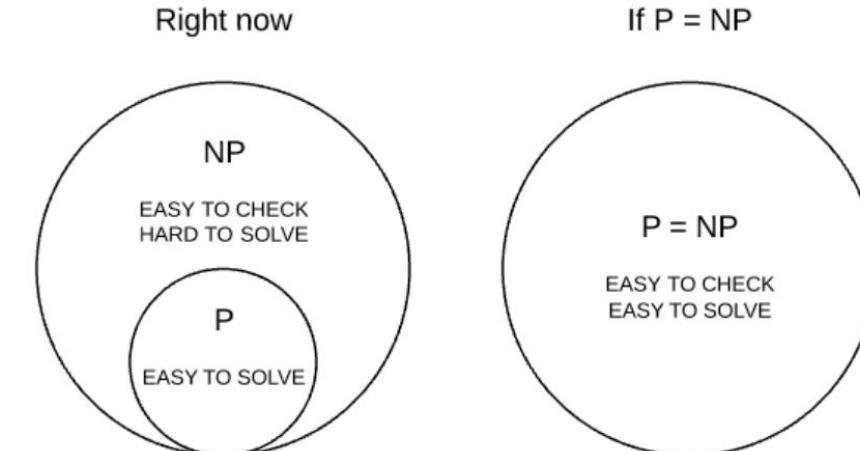


# P vs NP

- Now when we're familiar with the concept of complexity, let's use another grouping to classify problems
- P = set of problems that can be deterministically solved in polynomial time (so, feasible problems)
- NP = set of problems whose solutions can be verified in polynomial time, but can only be solved in non-deterministic fashion (if at all)
  - For example, finding prime factors of a large number: really hard to find, but when a solution is found, it can be checked by simple multiplication
  - Another example: finding solutions to  $A^3 + B^3 + C^3 = 3$  (A, B, C are integers)
- *NP-hard* = set of problems that can't be solved nor checked in polynomial time
  - Many optimization problems (no way to check whether an “optimal” solution is actually optimal or not)

# NP-complete problems

- *NP-complete* = subset of NP; set of problems to each which any other NP-problem can be reduced to in polynomial time
- Discovery of a polynomial-time solution for even one NP-complete problem would mean that all other NP-complete problems can also be solved in polynomial time
- This would mean that  $P = NP$ 
  - This would be a revolutionary discovery – providing an answer for the question "Is P=NP?" will be rewarded by \$1 million from Clay Mathematics Institute
- Current understanding is that  $P \subset NP$ 
  - Hasn't been proven yet!



# Correctness

- Can an algorithm be proven to work correctly in all cases?
  - Tough job, but not impossible
- In an ideal case, the *correctness* of an algorithm can be proven
- There are two types of correctness
  - Functional correctness: the algorithm produces the desired result for each input
  - Total correctness: on top of functional correctness, the algorithm also terminates in finite time
- As the name implies, total correctness is harder to prove
  - For example, if the algorithm searches for a minimum value for some function and this minimum value is achieved by multiple variable values → will the algorithm terminate, or will it alternate between these options indefinitely?

# Methods of correctness proof

- A list of methods that are possible to use:
  - Induction proof
  - Use of invariants (= arguments that remain unchanged during the execution), which are proven by induction
  - Proof by consecutive arguments
  - Curry-Howard correspondence (aka Curry-Howard isomorphism)
  - Hoare logic
- Automatization of proofs requires a separate logical system
  - Prolog is a programming language specifically meant for automatic theorem proving
- These methods above won't be considered in detail on this course
  - You may encounter some of these in later courses, though

# “Proof” by testing

- In practice, correctness can be proven only for small programs or program modules; proving the correctness of large programs (such as operating systems) is practically impossible
- Because some kind of verification that the program works correctly is needed, a practical way to approach this is *testing*
- Testing can detect at least the following types of problems:
  - Definition errors (index mistakes, divide by zero-situations, ...)
  - Usage errors (user thinks he/she gives the input correctly, but the program interprets it in another way → improve documentation or improve UI/syntax)
- Something to remember: also testers can make mistakes!

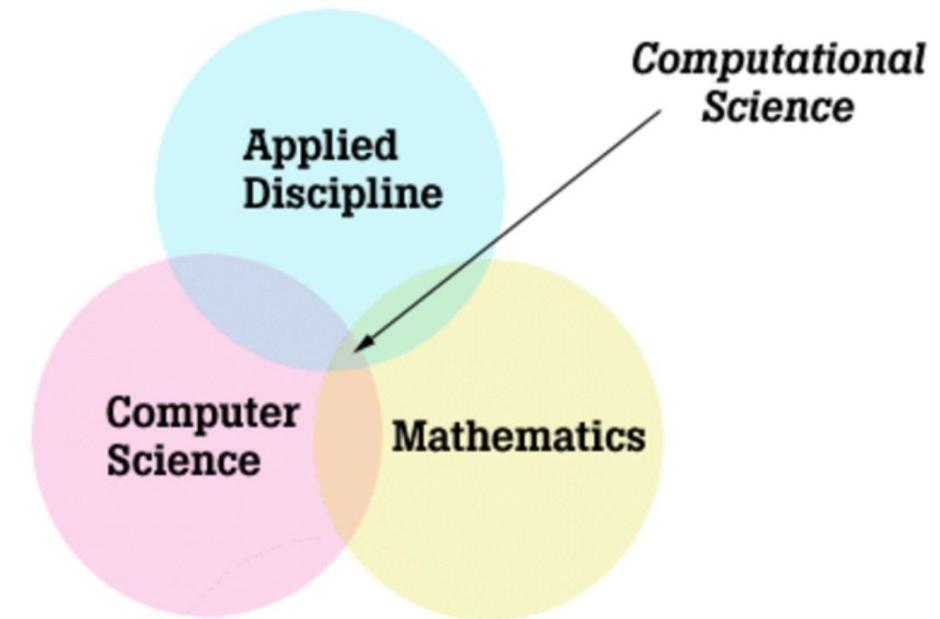
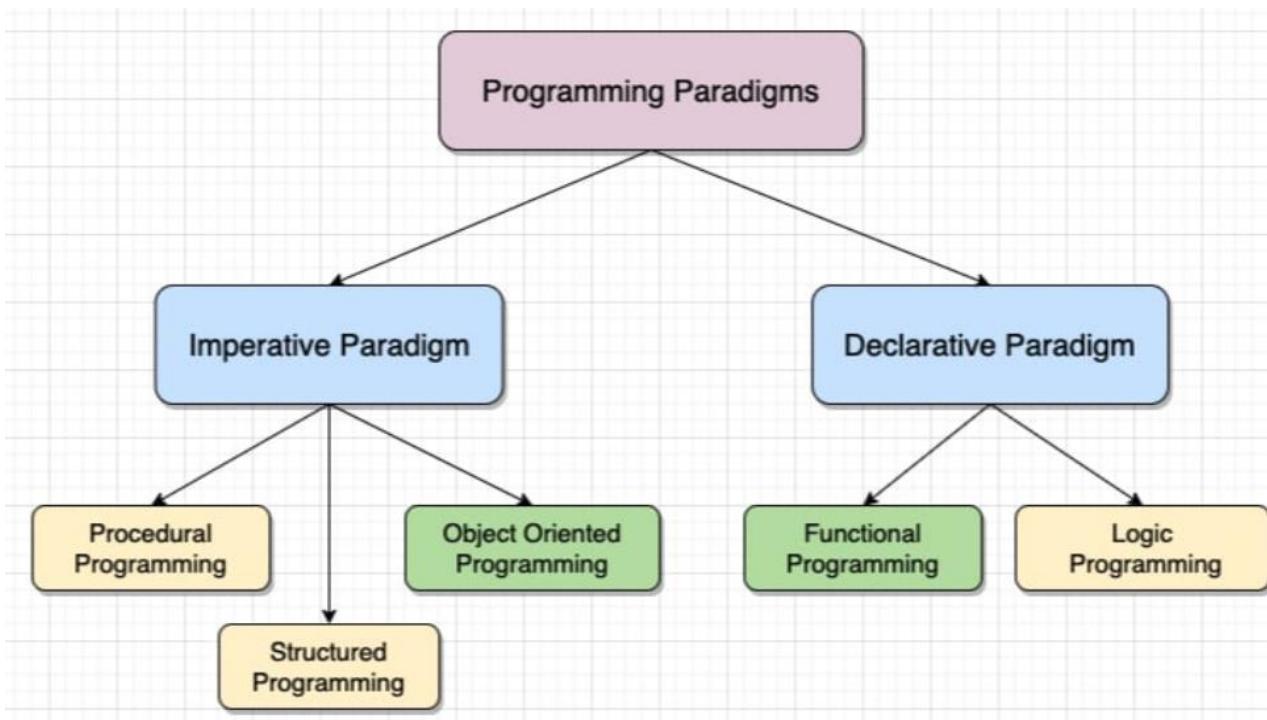
# Summary

- Problems can be classified based on their computability
- A single problem can be solved by (theoretically indefinitely) many algorithms
  - There are major differences in efficiency – especially as the size of the problem increases
  - (Asymptotic) time complexity is the key
- Computability of a problem is largely defined by the complexity of the best algorithm that solves the problem
- Problems that can be solved in polynomial time are considered feasible
- It is hard to prove completely the correctness of an algorithm – let alone the correctness of a program
- Especially with larger programs we need to rely on extensive testing

# Thank you for listening!



# 10. Programming paradigms & computational science



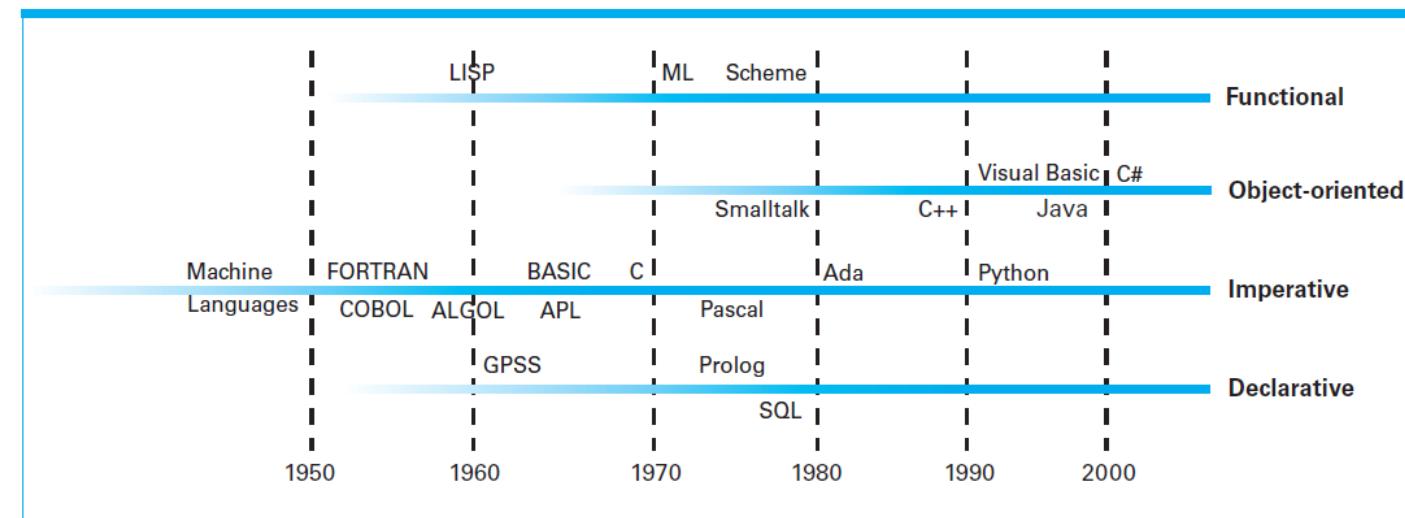
# History of programming languages

- We've already discussed different programming languages, their evolution and translating from one language to another during the first half of this course
- Let's make a short revision:
  - Originally, computers were programmed using computer-specific (and essentially numeric) machine languages; these are commonly known as first-generation languages
  - Readability of code took a giant leap forward with the development of mnemonic assembly languages (which gave us the option to give variables and operations somewhat understandable names) – known as second-generation languages
  - The programs written in assembly language were still quite machine-dependent, because the primitives in them were equal to their machine language counterparts
  - Also, these primitives were very small as building blocks
  - Both these problems were solved by third generation languages, which were both machine independent as well as contained much larger primitives
  - Nowadays the generations span up to 5, but gen4 & gen5 languages are for specific purposes

# Programming paradigms

- Generation-based approach is not the best for classification of languages
- Programming languages have developed along different paths as alternative approaches to the programming process – called *programming paradigms*
- These paradigms represent fundamentally different approaches on how to build solutions to presented problems
- Four main paradigms are
  - Imperative
  - Functional
  - Object-oriented
  - Declarative

Figure 6.2 The evolution of programming paradigms



# Imperative paradigm

- Imperative paradigm (also known as the procedural paradigm; although some distinct procedural as a subset) is the traditional approach to programming
- Aim is to develop a sequence of imperative commands that (if followed) manipulates the data in such a way that the desired result is reached
  - Very engineer-ish “flowchart” method
- The structure of the program can be clarified by dividing it to subprograms called *procedures*
- Procedures can be executed sequentially (phases can be numbered) or concurrently
  - Concurrency requires additional control methods
- Examples: C, Ada, Pascal, FORTRAN, Basic, Cobol, ...

# Downsides to imperative paradigm

- Imperative (and procedural) algorithms have previously dominated programming, because they are very deterministic in nature and correspond to von Neumann-architecture
  - Troubleshooting is “easy”, because the problematic procedure can be identified
- On the other hand, if there are many procedures, meticulous attention needs to be put on the execution order; it is far too easy to design an algorithm that has large “visual complexity” and hence is prone for mistakes
  - For example, if there are several IF clauses: which “ELSE” is dedicated to which IF?
- Procedural algorithms are not intuitive in all cases
- Proof of correctness is a work-heavy task

# Functional paradigm

- In functional paradigm, a program is seen as an entity that accepts inputs and produces outputs directly from the inputs by using smaller entities
  - These entities are called functions (hence the name)
- Program is constructed by connecting smaller functions in such a manner that the outputs of the previous function act as inputs to the next one
  - The "main function" = the solution to the problem
- Information is presented as common data list structure
  - Data as well as the program are presented in same fashion
- Based on Church's lambda calculus
  - Result of an algorithm is defined as a mathematical function (arguments = inputs)
- Examples: Lisp, Scheme, Erlang, Haskell, ...

# Example: imperative vs. functional paradigm

- Problem: find the balance of a checkbook
- Imperative:

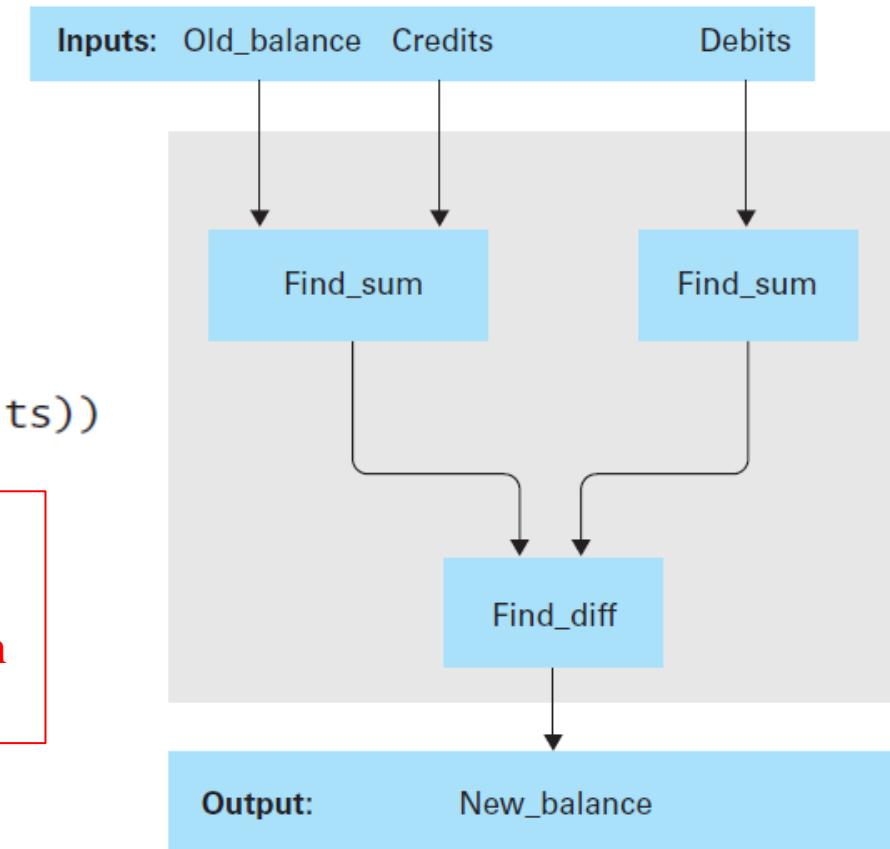
```
Total_credits = sum of all Credits  
Temp_balance = Old_balance + Total_credits  
Total_debits = sum of all Debits  
Balance = Temp_balance - Total_debits
```

- Functional:

```
(Find_diff (Find_sum Old_balance Credits) (Find_sum Debits))
```

At first glance, these may seem quite identical.

But note: imperative version has several statements, and the mid-results of all such statements are stored! Functional version only has one, which results in better efficiency.



# Object-oriented paradigm

- In object-oriented paradigm, a software system is viewed as a collection of *objects* which can perform actions on themselves and/or request actions from other objects
  - For example, in a GUI, all icons are objects
- Methods linked to the object describe how the object responds to different events
  - Left mouse click, right mouse click, double click, ...
- Advantage: if several programs use the same object, the functions needed are already provided with the object
- Description of the properties of an object is called a class
  - Creating multiple objects with same properties is easy (class can be applied to new objects)
- Foundation of most OOP languages is imperative, though - methods are small imperative program units
- Examples: C++, Java, Python

# Declarative paradigm

- In declarative paradigm, programmer is asked to describe the problem to be solved
- Declarative algorithms are non-deterministic, so the execution order of phases of the algorithm is not typically known
- The system applies a general-purpose problem-solving algorithm
  - Such algorithms unfortunately don't exist for very many problems
  - Commonly some deterministic search method is applied (depth-first search or similar)
- Declarative languages are therefore most suitable for some special applications
  - Hypothesis testing & predictions
  - Parallel computing
- Subfield: logic programming (find out whether claim x is true or not)
- Examples: Prolog, SQL

# Concurrent vs. parallel computing

- The traditional programming method is to do everything in clearly ordered phases
  - so, in sequential fashion
- This limits us from making use of parallel computing
- Concurrent computing can be applied to sequential programs
  - Concurrent = multiple actions can be performed by rapidly switching the process in execution in order to give the user a feeling of parallelism
- True parallelism is based on parallel use of our system:
  - Multi-core processors / multiprocessor computers
  - Computer networks, clusters, decentralized computation
  - Supercomputers

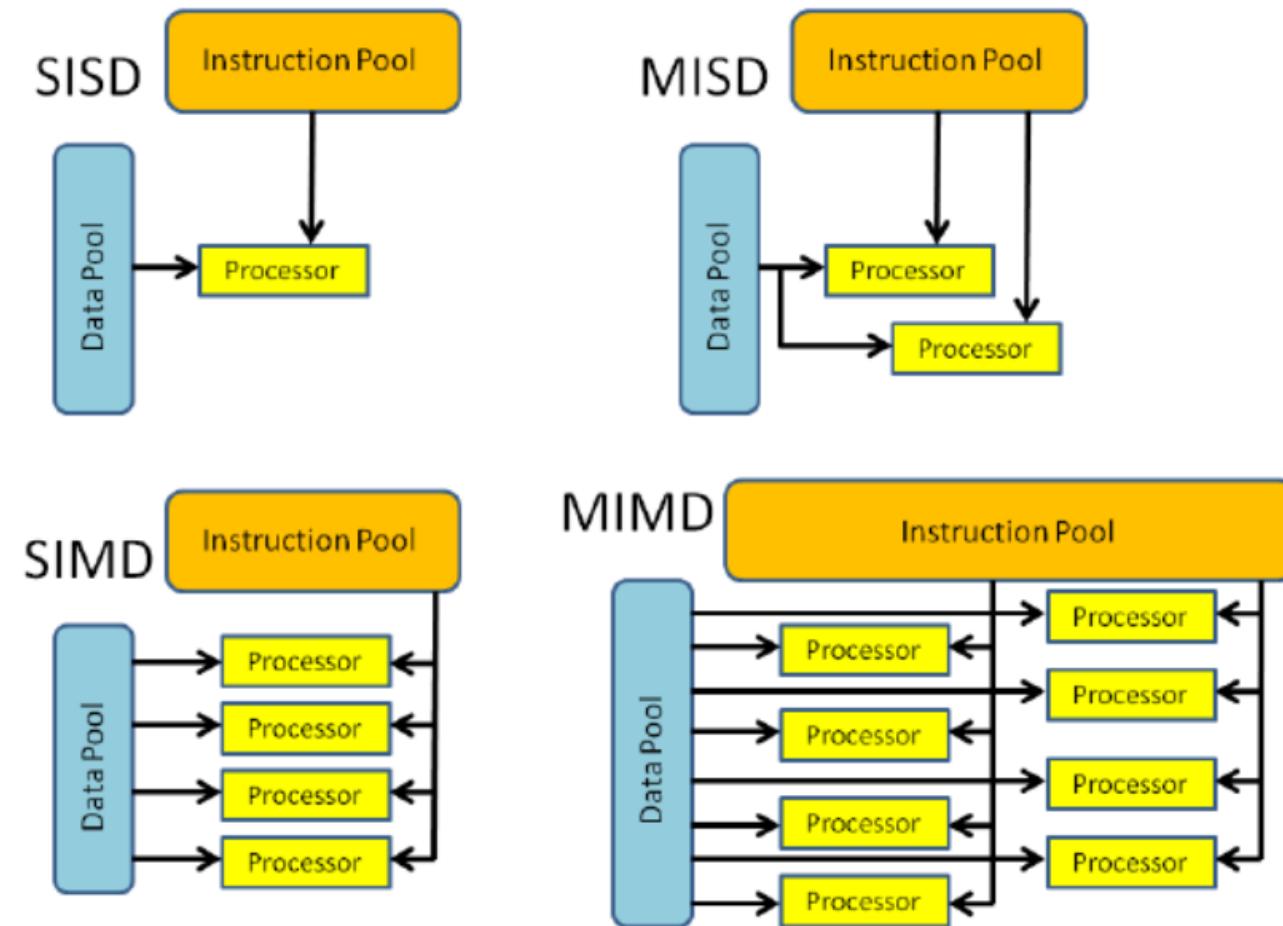
# Parallel programming

- Not all problems are suitable for parallel computing
  - For example, problems where the previous phase produces the inputs of next phase
  - This is surprisingly common in practice
  - Parallel computing could be used in some phases, but not throughout the problem
- On the other hand, there are also problems for which parallelism feels natural
  - For example, face recognition: find a match from a database of 500 000 photos → if we have 10 computers available, the database can be divided so that each goes through 50 000 photos
- Designing a parallel algorithm may deviate a lot from a sequential algorithm
- There are programming languages which are especially suited for parallel programming (usually dependent on system or architecture)
- Parallelism in pseudolanguage: PARDO

```
FOR i := a,...,b PARDO
    <body>
END
```

# Flynn's taxonomy

- Computer architectures can be divided in four categories in relation to their ways to handle instruction & data streams:
  - SISD (single instruction, single data stream; all old single-core processors)
  - MISD (multiple instruction, single data; used only for fault detection)
  - SIMD (single instruction, multiple data; basically all modern GPUs)
  - MIMD (multiple instruction, multiple data; all modern multi-core processors)
- This division is known as *Flynn's taxonomy*



# Communication and speedup

- Especially in MIMD implementations, good communication between processors/cores is the key to performance improvements
  - Cores can share mid-results with each other etc.
- There are two ways to communicate and share information:
  - Shared memory (all cores can access the information)
  - Message passing (if memory is distributed; cores send “private messages” to each other)
- A benchmark quantity for measuring quality of parallelism is speedup
  - If the problem is divided to  $n$  cores/CPUs, how multiple will the performance rise?
  - Best-case: linear speedup =  $n$  (theoretical maximum)
- In special cases, a superlinear speedup is possible
  - For example, if breaking the problem to smaller parts results in all parts fitting in the cache (greatly reduced memory fetch times)

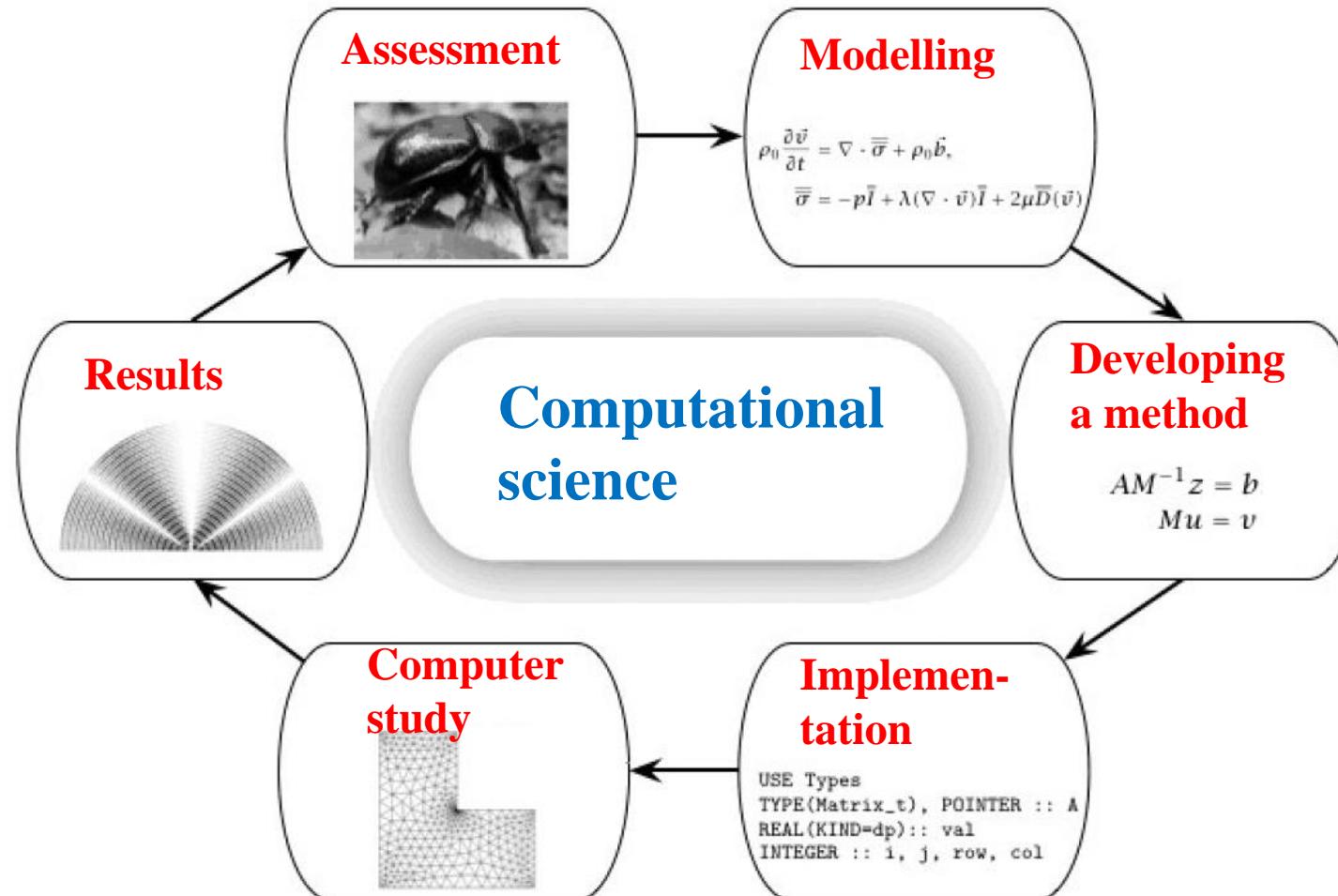
# Computer science and mathematics

- Previously we've looked at things from the angle of a programmer
- We might remember from the first lecture that the first computer scientists were actually mathematicians; computer science and mathematics have a special bond
- There are problems in mathematics which are far too complex to solve in an analytical fashion
- The field that aims to solve these problems by taking advantage of modern computing capabilities is called *computational science*
- Computational science primarily deals with the most mathematical side:
  - Mathematical models
  - Algorithms
  - Optimization

# Definition of computational science

- Computational science is a field mainly concentrated in three topics:
  - 1) Algorithms and programs, modelling & simulation knowhow
  - 2) Development of hardware, software, data communications technology and information processing components needed for solving challenging tasks
  - 3) Computing infrastructure that supports the problem-solving & development of related sciences
- Possibilities:
  - Create more realistic models for solving complex and wide study problems (climate change, garbage problem of seas, fusion energy, ...)
  - Decrease the need for expensive experiments & prototypes
  - Study the correlations and causations between phenomena
  - Gain better understanding of things in the “big picture”
  - Improve risk control and tolerance to uncertainty

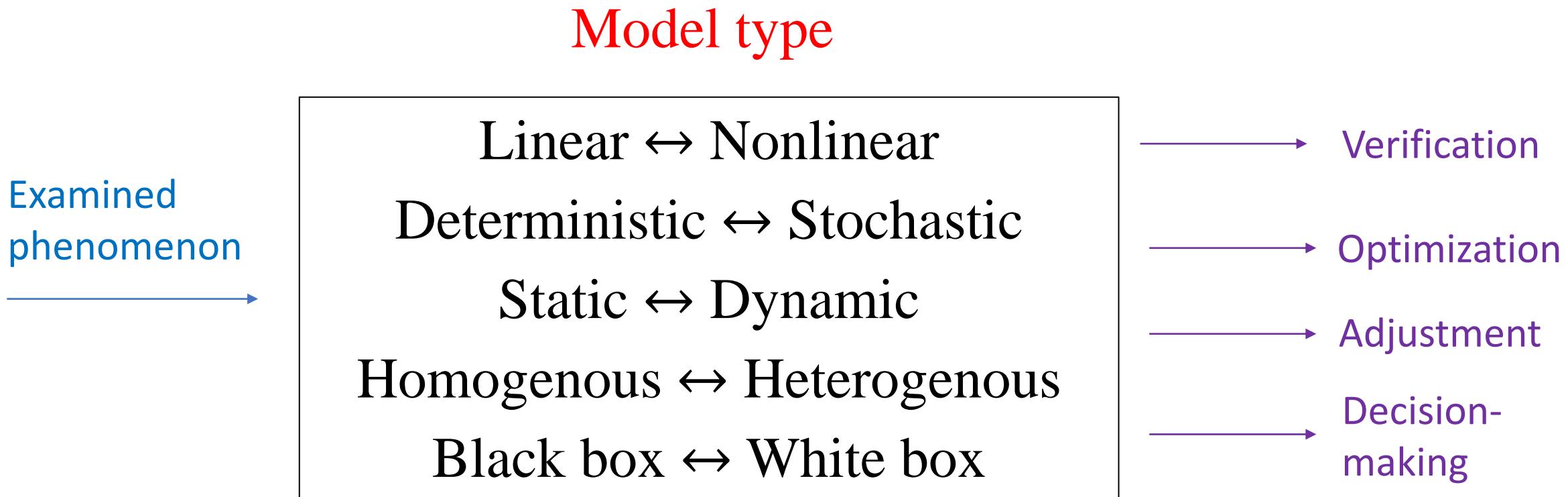
# Process cycle of computational science



# Modelling with data

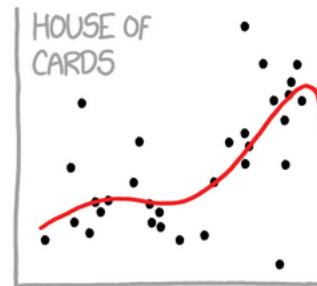
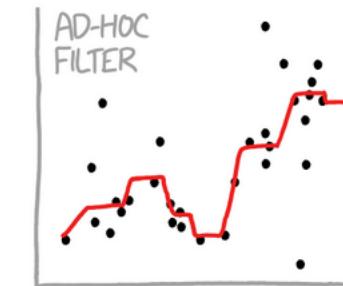
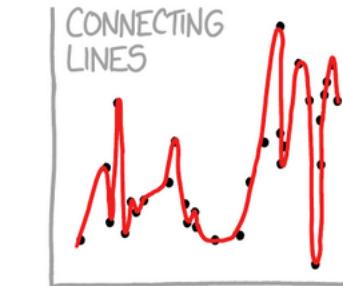
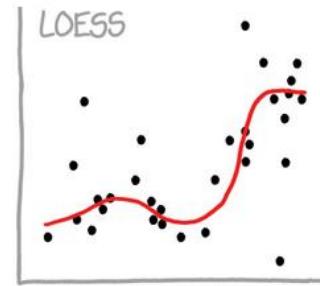
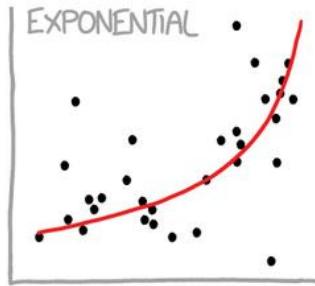
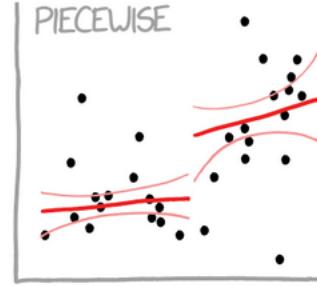
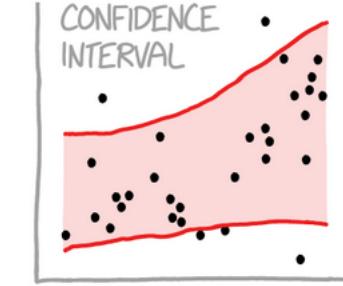
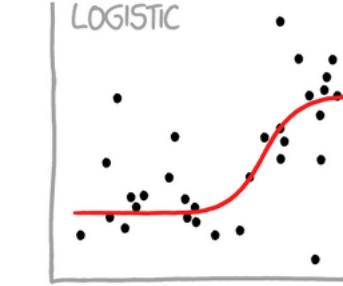
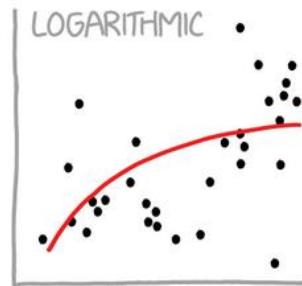
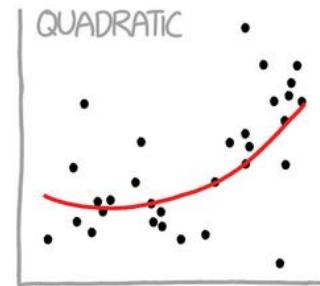
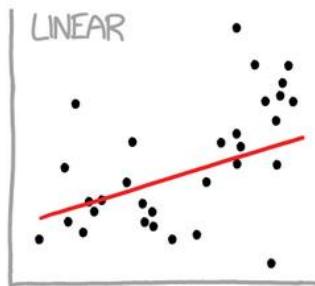
- What part of the data is essential considering the question?
- How correct is the data we get from the sensors?
  - Calibration errors, measurement errors
- What internal relations does the data have?
  - Which data is “raw” and measured, which has been calculated from raw data?
- How can the data be saved during the measurement, are there delays?
  - For example, digital image correlation (DIC) systems require a quick hard drive
- Accessibility and transferability of data
  - Cloud storage or something else? Transferring hundreds of GBs of data is hard – especially if security is an issue

# Types of mathematical models



# What model fits the data?

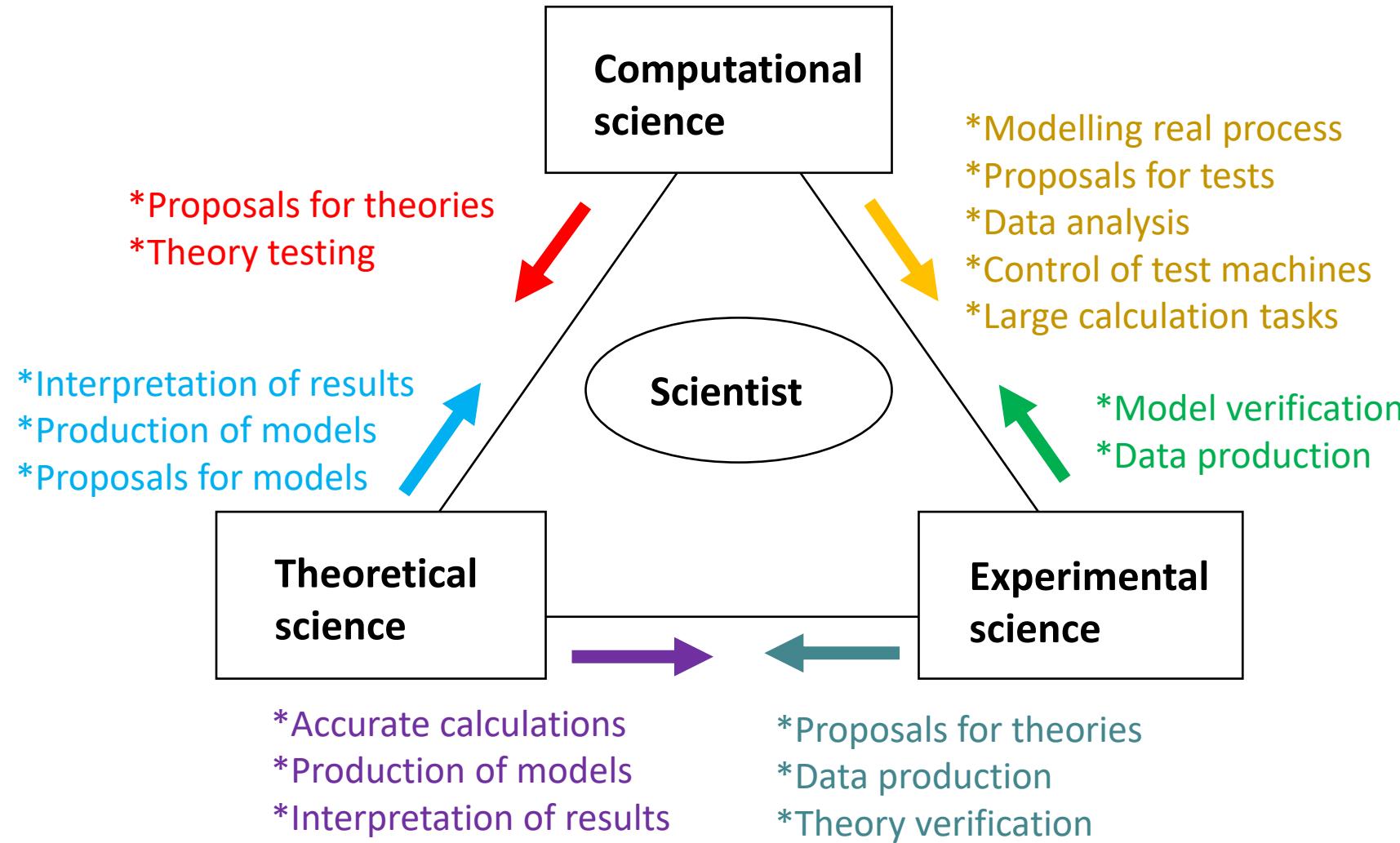
## CURVE-FITTING METHODS AND THE MESSAGES THEY SEND



# Optimization

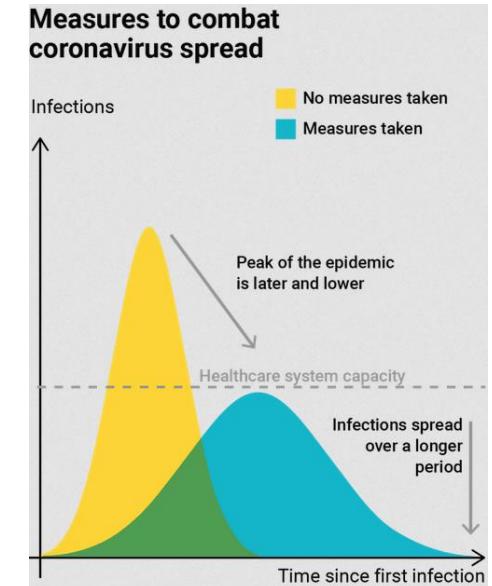
- After a model has been created, it can be *optimized*
- Computational science has given rise to many advanced optimization techniques
- One good example are genetic algorithms, which mimic evolution:
  - 1) Initiation: create a random population of solutions
  - 2) Selection: select the best solutions using a fitness function and delete the worst ones. If a good enough solution has been found (or time has run out), stop and go to 5.
  - 3) Combination: perform crossovers and mutations to the remaining population on order to create a new population – the next evolution version
  - 4) Repetition: go back to stage 2 with the new population
  - 5) Post-processing & visualization of results, termination
- Especially good algorithms when problem is supposed to have multiple local minima where derivative-based methods may get stuck

# Role of computational science



# Application areas of computational science

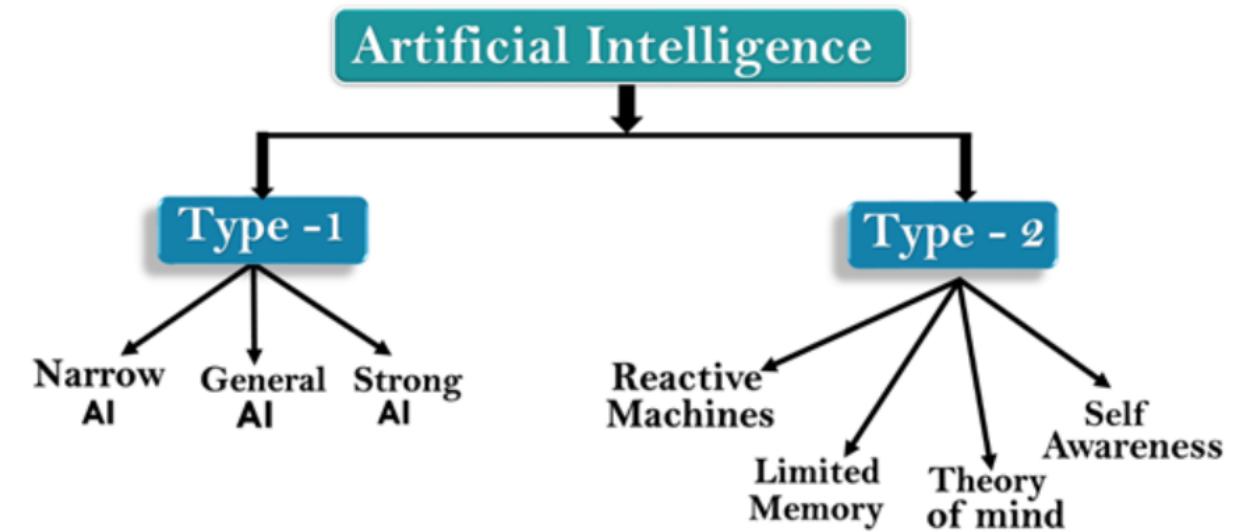
- Modelling fusion reactor behavior
  - Global epidemic models (ebola, covid-19)
  - Streaming services (recommendation algorithms, user demand)
  - Measurement & instrumentation technology
  - Remote mapping of natural resources (laser scanning of forests etc.)
  - Process diagnostics in factories
  - Raw material analysis (optimal sawing of a log)
  - Dynamic traffic steering
  - Machine vision & pattern recognition applications
- ...and many, many more!



# Thank you for listening!



# 11. Visualization of data & artificial intelligence

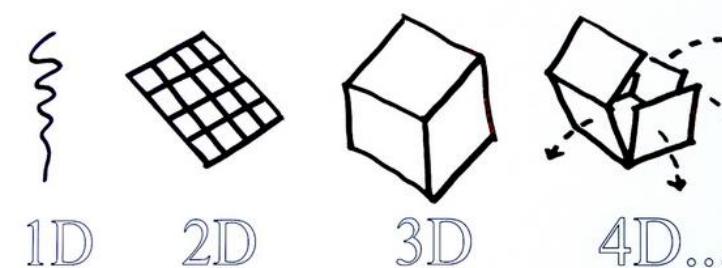
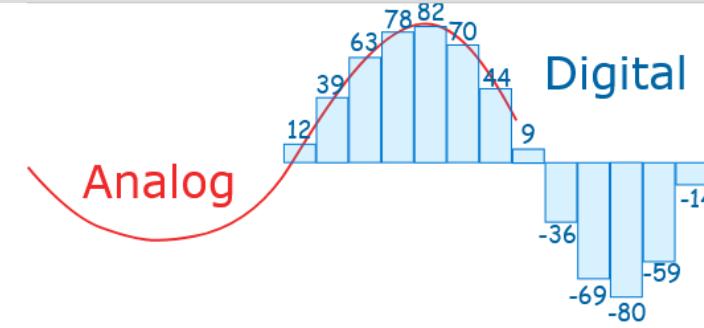


# Revision: Data vs. information

- “*Data*” is just numbers or text without context
- When we add context to data, it becomes *information*
- Correctly interpreted information enables advances in *knowledge*
- Are we visualizing data or information?
  - The nature of this question is more philosophical, because the computer doesn’t understand the difference
- We might start with visualization of data – which then becomes visualization of information, when context is added
  - Removal of outliers
  - Axis labels, headers, scaling

# Types & dimensions of data

- Our data can be numerical or symbolic
- Numeric data signal can be analog or digital
  - Analog is continuous by nature (number data, but recording must be done by discretization)
  - Digital is discrete by nature, consists of 1's and 0's
  - Signals differ in susceptibility to disturbances (analog is prone to noise)
- Data has different dimensions – based on what we're recording:
  - 1-dimensional (single variable – for example, voltage measurement)
  - 2-dimensional (x- & y-coordinates – for example, black and white photo)
  - 3-dimensional (for example, a color image – RGB)
  - 4-dimensional (3D + time; for example, a color video)
  - n-dimensional



R	G	B
54	58	255
45	0	8
85	78	0
47	51	74
20	100	36
36	207	21
22	148	123
52	52	147
36	214	41
158	278	255
0	78	247
72	51	255
74	136	74
74	251	74

# Information processing or calculation?

- Terminology-wise, what are we actually doing?
- The borderlines of scientific disciplines are not clear
- One take at the subject based on data type & methods of problem-solving:
  - Note: this is a simplified view and has been contested

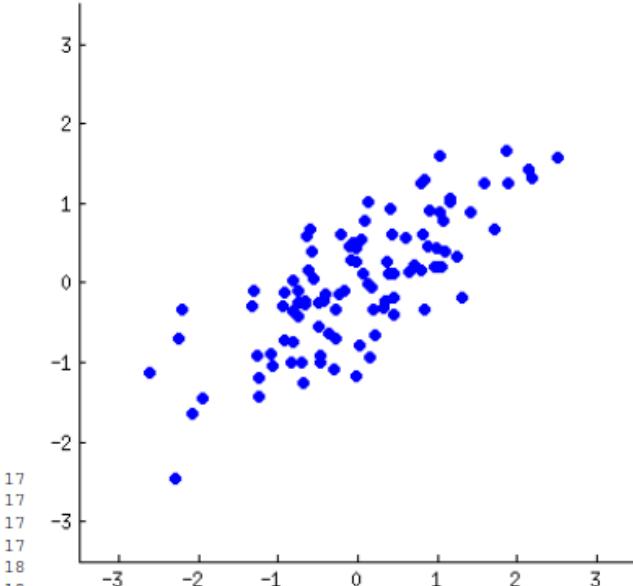
	Algorithmic	Heuristic
Numeric data	Technical mathematics calculations	Simulation & signal processing
Symbolic data	Economical/governmental information processing	Artificial intelligence & knowledge engineering

# Importance of visualization

- Measurements provide us lots of data, usually in numeric form
- It is very hard to see trends from hundreds of rows of numbers/text
- When data is plotted to a graph, relationships between variables become easier to see
- Essential for formulation of hypotheses



185	182	152	179	179	181	179	180	179	179	178	176	177	17
180	183	132	177	181	182	177	178	175	178	177	177	175	17
126	175	171	183	173	183	181	179	181	182	180	180	180	17
129	125	167	184	183	183	179	181	181	182	183	179	179	17
49	76	117	175	187	181	185	183	178	181	181	181	180	18
65	52	68	96	184	183	178	182	183	182	179	181	181	18
107	104	104	148	182	193	189	189	187	189	188	183	174	179
97	98	100	102	102	106	122	137	113	116	123	169	190	178
29	30	29	28	29	27	92	117	29	28	27	33	194	177
32	31	29	27	36	53	98	97	30	30	26	25	182	176
190	192	178	164	203	227	229	149	129	128	98	53	140	105
197	214	183	169	194	158	201	56	35	34	121	107	175	161
156	186	192	178	223	220	161	36	33	33	134	116	199	201
97	228	71	42	124	206	110	36	30	32	137	122	152	45
117	187	142	35	85	112	141	35	32	31	142	122	84	30
173	196	71	38	37	72	71	40	34	31	146	125	71	49
211	214	92	36	35	150	213	141	37	116	145	135	76	73

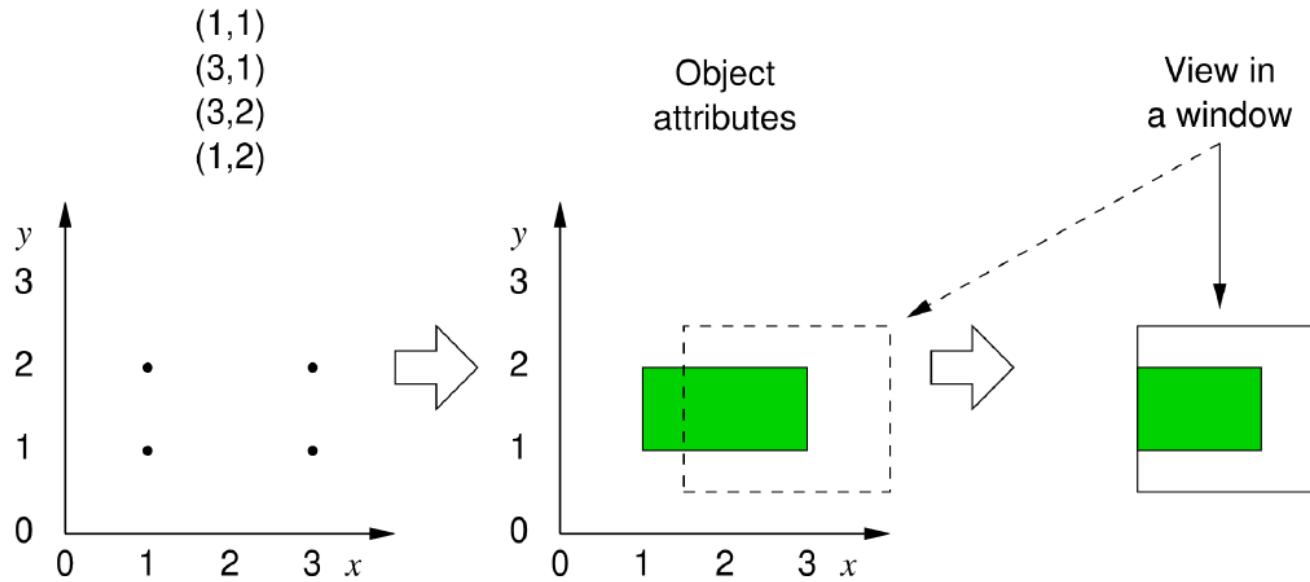


# n-dimensional data

- If we conduct large-scale measurements for some machine/phenomenon, we'll usually have multiple sensors in various places and record dozens of things
- As a result, do we have dozens of one-dimensional data streams or one n-dimensional data stream?
  - If all measurements are done at the same time, theoretically the latter – because the data allows us to study the relations between variables!
  - Difficulties in visualization of n-dimensional data often forces us to decrease the number of dimensions – at least at once
- This requires tools from the toolbox of information processing
  - Selection of essential variables for examination of the desired task using heuristics or specific calculation methods (for example principal component analysis, PCA)
  - Signal processing & linear algebra might be needed, too (data cleanup)

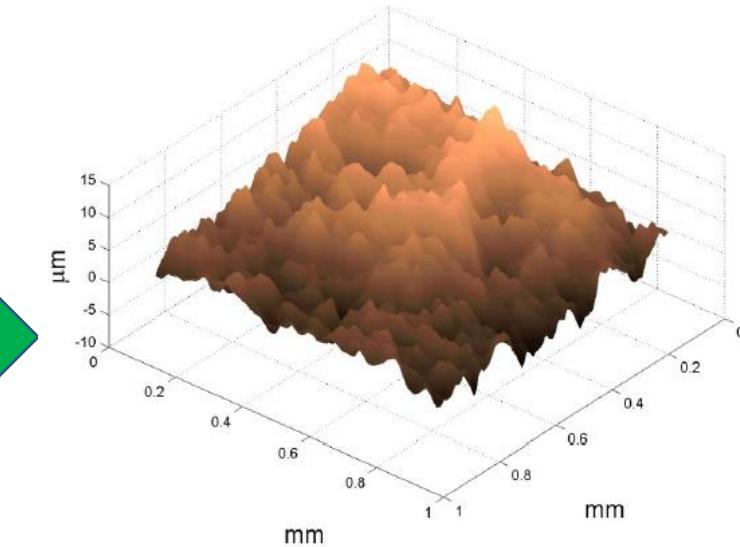
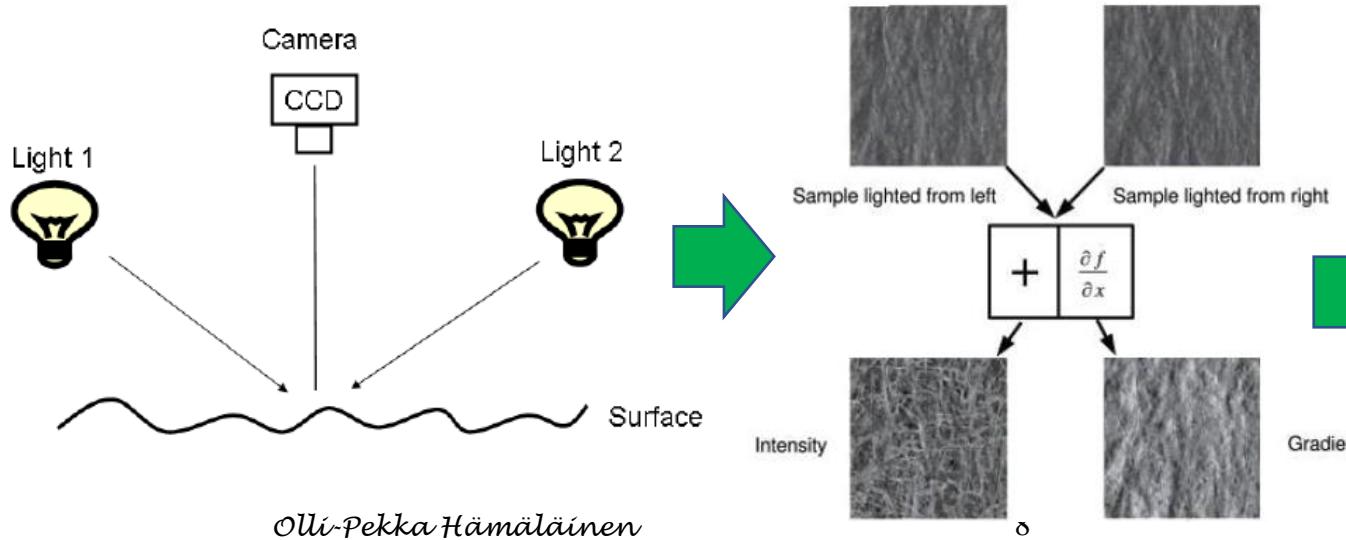
# From data to geometry: production

- Computer graphics is all about how geometric shapes are formed from data
  - Group of data points are told to form an object – but this is nothing yet
  - Object can be given attributes, which specify the connection method of points & possibly some other features, too (like color, for example)
  - Visibility of the object is dependent on player's orientation & interaction with other objects



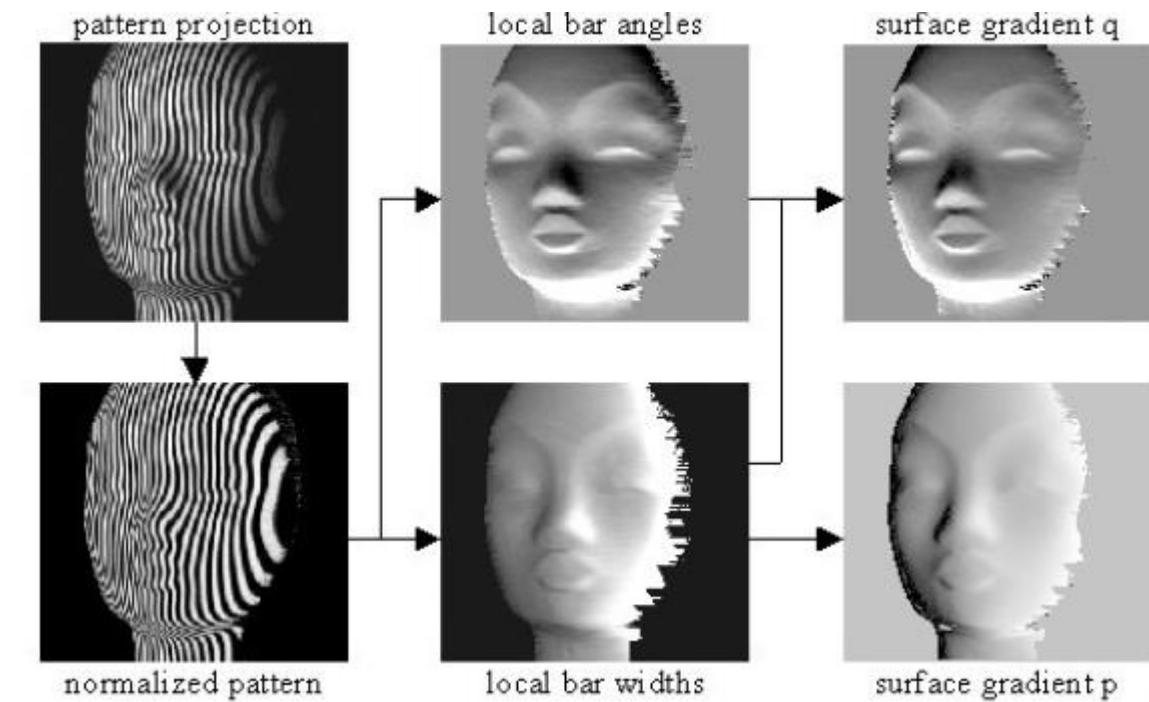
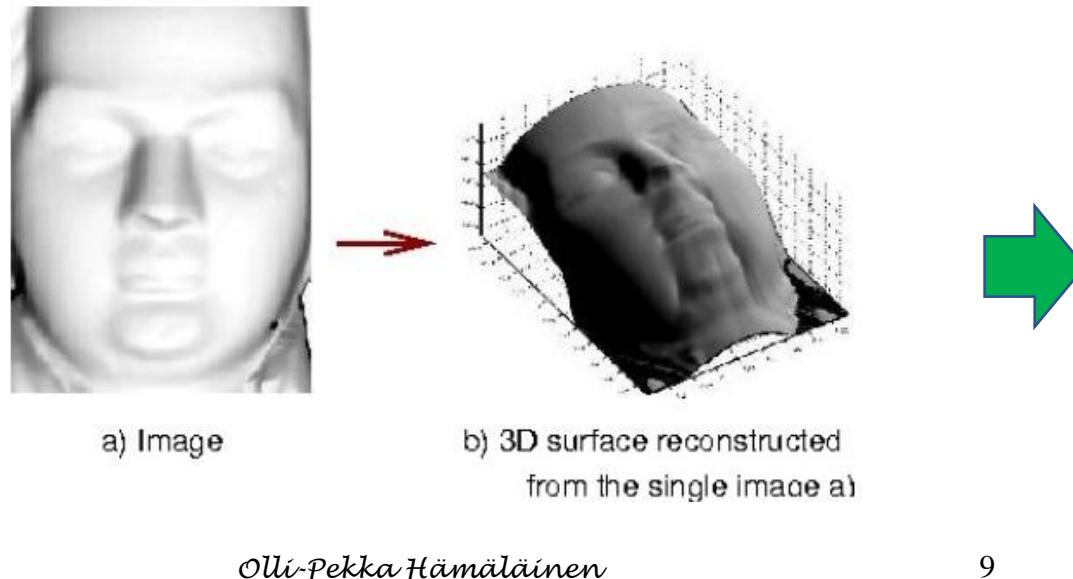
# From data to geometry: reproduction

- If we want to reproduce a real-life geometry on a computer, we have to be able to measure it properly
  - 3D laser-scanning is a good option, but expensive & work-heavy
- A cheaper alternative is to use a “regular” camera and photometric stereo
  - One camera, two pictures of the sample – one lighted from left, one from right
  - By combining the pictures, we can separate intensity and gradient
  - Surface shape can then be extracted by integrating the gradient



# From data to geometry: reproduction

- More refined version of the previous approach is to use structured light
- Widely used in facial reconstruction
  - “Shape from shading” technique is used to convert one image to a 3D surface
  - Structured light is then projected at the 3D reconstruction
  - Produces a smoother result



# Power & limitations of the human mind

- Human mind is a very powerful problem-solving tool
- Its power lies mostly in its ability to create connections between different subjects
  - Think on multiple levels of abstraction
  - Formulate analogies and use them to advantage when learning new things
  - Create innovations by combining knowledge from multiple areas
  - Understand the context of things and draw conclusions from subtle hints
- On the other hand, human mind has many limitations
  - Low amount of “random-access memory” (especially)
  - Constant multitasking is problematic (interruptions are a bad thing here)
  - Variation in performance due to fatigue & boredom
- Conclusion: some problems are more suited for humans, some for computers
- In order to enlarge the latter group, we need artificial intelligence

# What is intelligence?

- What can be classified as intelligence?
  - Reflexes vs. learned behavior models vs. innovative problem-solving
- How do we know if animals are intelligent?
  - “Blanket test”
  - Communication
  - Ability to speak
  - Learning from mistakes (trial and error)
  - Abstract thinking
- How do we know that humans are intelligent?
  - We’ve set the standards by ourselves, is that fair?
- How can we then define when a machine can be considered intelligent?
  - A computer can solve certain problems very quickly – when specified by user

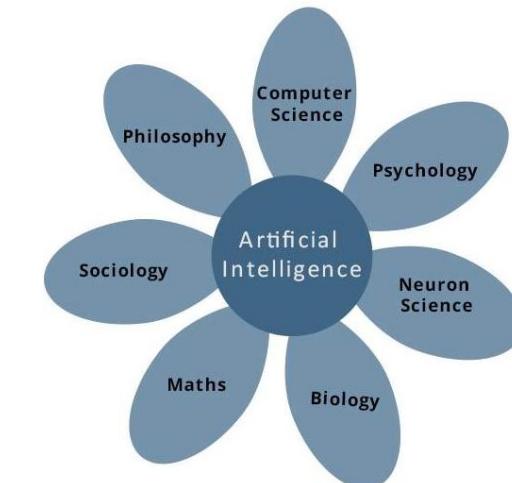


# Measurement of intelligence

- There are no “correct” answers to previous questions due to their philosophical nature; let’s take a more engineering-like approach and try to quantify
- Most well-known measure of intelligence is the intelligence quotient (IQ)
  - IQ score of 100 represents a median result, larger score means more intelligent
  - Distributions around the 100 differ by author (several scales)
  - ...but what do we use as a population based on which we specify the median?
- Measurement of IQ is not 100% objective and has received criticism
  - IQ tests are based on logical thinking and finding patterns
  - A person can improve their IQ test score by rehearsing such problems; does this improve the person’s intelligence or just the test result?
  - Is logical thinking the only sign of intelligence, or can there be something else?
- Current understanding: there are 8 types of intelligence (logico-mathematical is just one of them)

# Concepts of artificial intelligence

- Artificial intelligence is a field of computer science that aspires to build autonomous machines that can carry out tasks without human intervention
- Alongside CS, also other fields of science are needed
  - Psychology (desired behavior)
  - Linguistics (communication)
  - Philosophy (ethics)
- Some general terms and classifications:
  - Weak AI: only developed for one single action
  - Strong AI: superior to humans (not created yet)
  - Symbolic AI (“GOFAI”): created using “traditional” methods, creator sets the rules
  - Connectionist AI: created using neural networks, AI learns the rules by itself using sample data

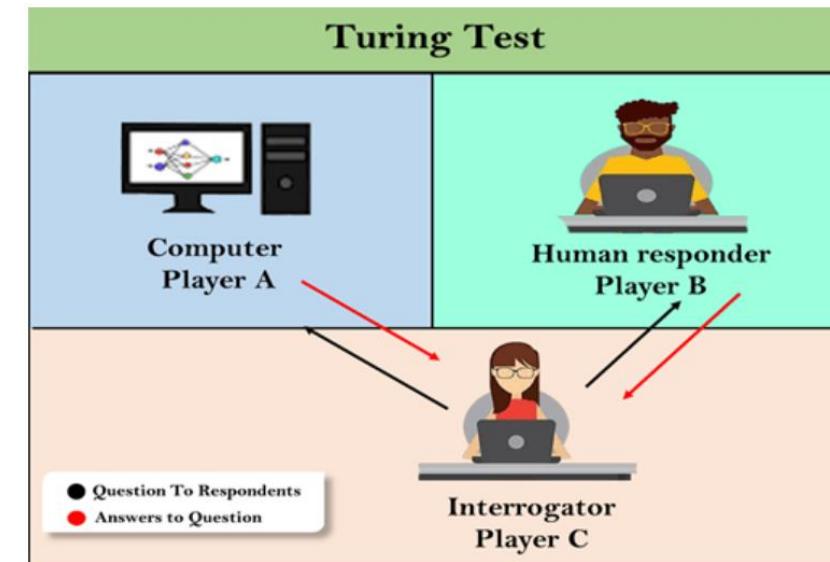


# Goals of artificial intelligence

- AI research is trying to create machines that are capable to
  - Detect and measure (basis of cognition)
  - Process measurements and information (basis for learning, deduction and planning)
  - Learn (learning algorithms, computational intelligence)
  - Deduce (classification & problem-solving)
  - Plan (optimization, assessing quality of forthcoming actions)
  - Communicate (information retrieval & exchange)
  - Move and handle objects (robotics)
- The advances are pursued along two research methodologies:
  - Engineering approach, which is performance-oriented ("just make it work")
  - Theoretical approach, which is simulation-oriented (computational understanding)

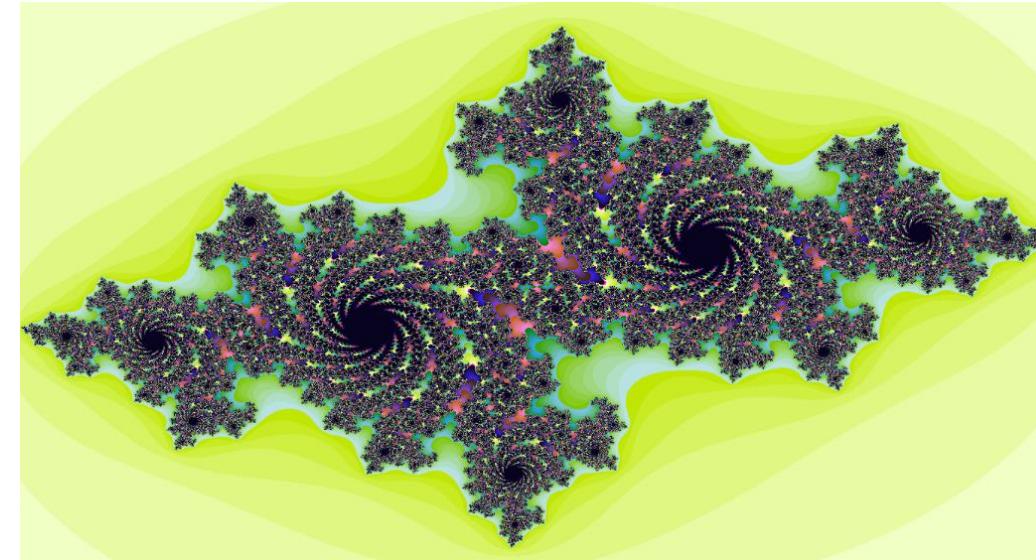
# Turing test

- Turing test was proposed by Alan Turing in 1950
- Idea: a human interrogator communicates with two “test subjects” via a typewriter program for a period of time, and after this is asked to tell which test subject was a real human and which one a machine portraying a human
- What counts as a “passed” test?
  - Turing predicted a 30% pass rate (5min test) by year 2000
  - This was reached in 2014 by "Eugene Goostman"
  - "Eugene" was claimed to be a 13-year-old Ukrainian boy
  - Was this "identity" a key factor in passing the test?
- Also criticism has been voiced towards the test
  - John Searle's "Chinese room"
  - Does the test tell anything about intelligence of the machine?



# Methods for implementation of AI

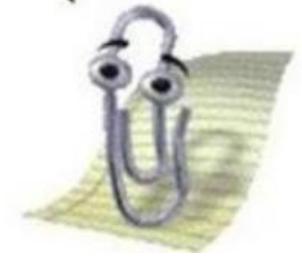
- AI can be implemented using one or several of the following methods:
  - Machine vision
  - Search trees
  - Heuristics
  - Expert systems
  - Neural networks
  - Fuzzy logic
  - Fractals & chaos theory
  - Evolutionary computation
  - Swarm intelligence
- In order to create a strong AI, it is mandatory to combine these methods



# Intelligent agents

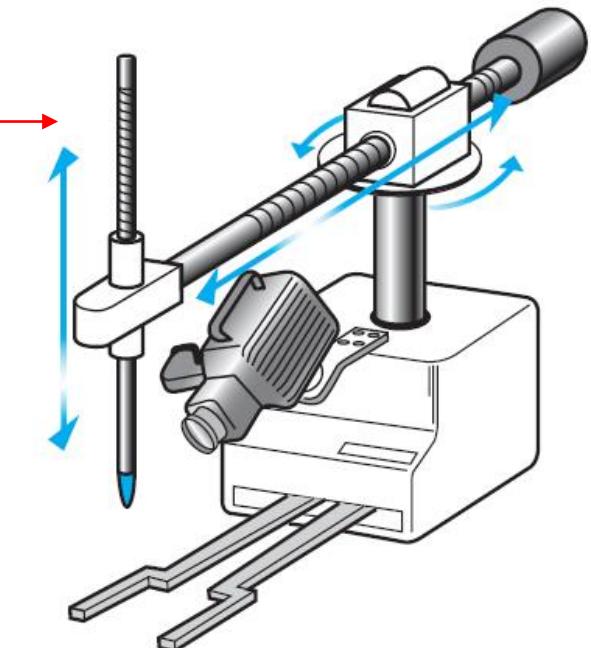
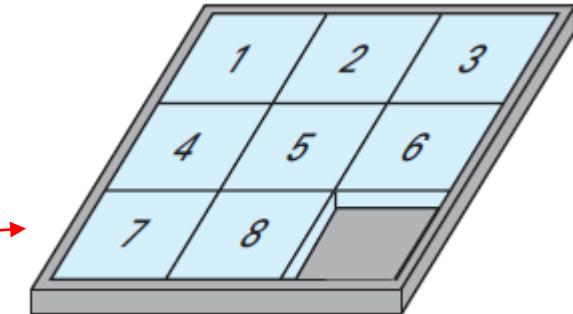
- Intelligent agent = a “device” that responds to stimuli from its environment
  - A robot / game character / self-driving car / chatbot / etc.
- Actions of the agent must be rational responses
- These actions are classified to different levels:
  - Reflex action: pre-programmed responses to specific inputs
  - Contexted action: responses depend on inputs AND the current environment
  - Goal-based action: agents’ responses are results of following a plan to seek a goal
  - Utility-based action: agent is able to measure what option most likely leads to the goal
- Agent is able to learn if its responses improve over time; this can be done by
  - Developing procedural knowledge (via trial and error)
  - Storing declarative knowledge (new principles added to the bank)

Sometimes I just popup for no particular reason, like now.



# Symbolic AI example: eight-puzzle

- Suppose we try to get AI to solve us a puzzle that has 3x3 tile matrix with 8 numbered tiles and one free slot
  - “Solved” state presented in the picture
- First, we need some kind of a machine that can both sense the current state of the puzzle as well as make moves
  - One possible option for such a machine pictured here
- Extracting the positions of tiles is easy, because the geometry is so simple:
  - Image processing = identify geometric features (tiles, numbers)
  - Image analysis = identify what these features mean
- ...but what if our puzzle numbers use different font?

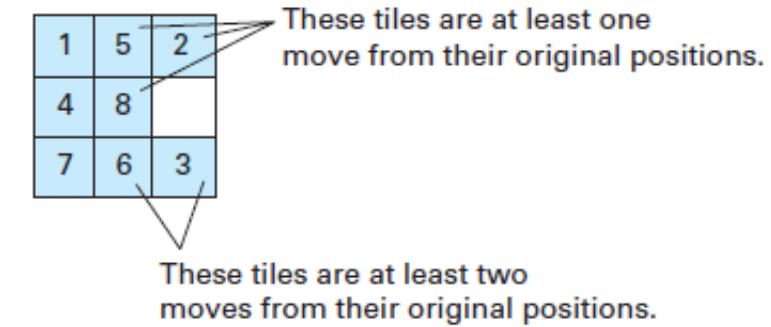


# Symbolic AI example: eight-puzzle

- When the current state has been found out, it's time to make moves
- For this, we need a production system that has 3 main components:
  - Collection of states (start state and goal state as the most important ones)
  - Collection of productions (possible movements of tiles)
  - Control system (decision-making on which tile to move)
- We can search for a solution by using a breadth-first search tree:
  - From start state, branch to all possible follow-up states
  - Continue this branching until one branch reaches the goal state
- Downside of breadth-first method: if we're far away from the solution, the search tree grows to immense size quite quickly
- Alternative: depth-first search tree
  - Explore each possibility until the end

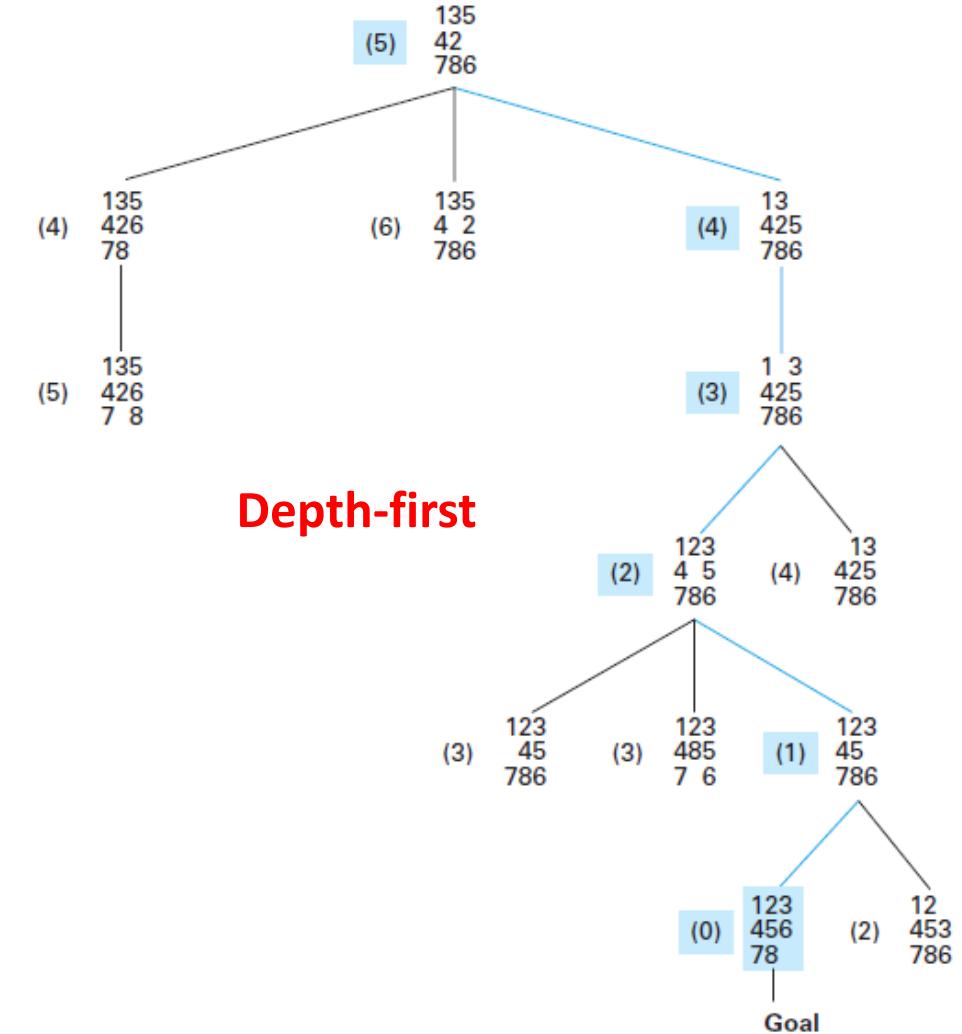
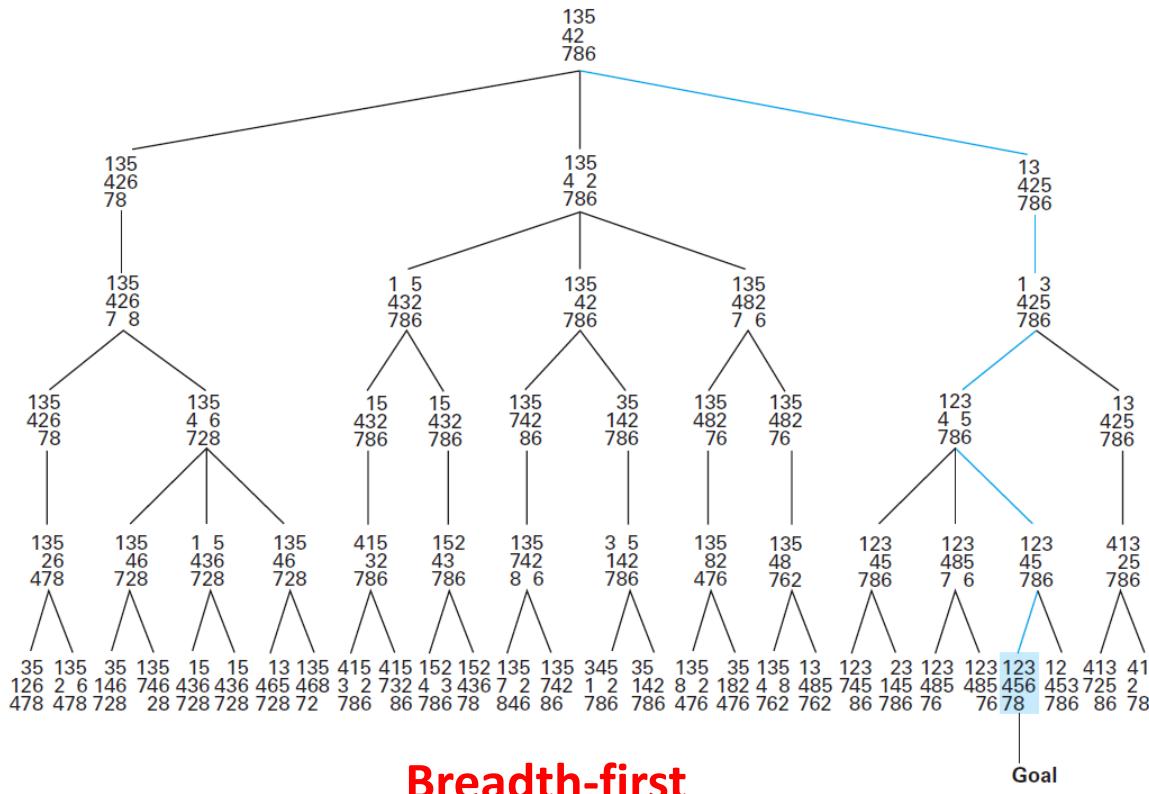
## Symbolic AI example: eight-puzzle

- Depth-first search can be improved if we can formulate a fitness function and then use this as a heuristic to decide which would be the best move
- In this example, two options come to mind:
  - a) Sum of tiles that are not in their correct place
  - b) Sum of distances of tiles from their desired location
- Both of these are easy to calculate
- Let's use b) as a heuristic now
- In each node, we select the branch that has the smallest fitness value
  - If this ends up in a worse situation, we back off



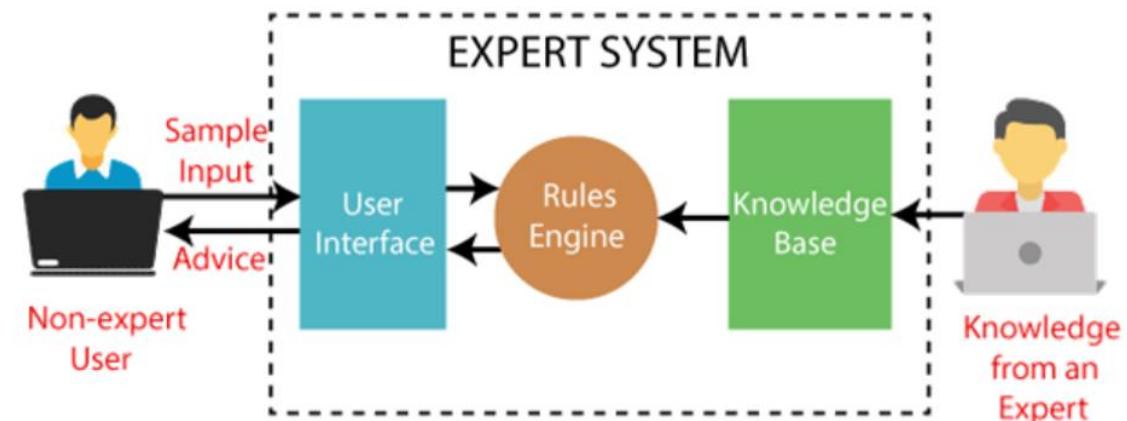
# Symbolic AI example: eight-puzzle

- Comparison of search trees



# Expert systems

- Computer programs which aim to provide help for decision-making
- Experts put together a system that can be used by non-experts
- Expert system consists of three main parts:
  - Knowledge base = big storage of knowledge put together by experts (note: knowledge can be factual or heuristic)
  - Inference engine = rules that are used to make deductions (note: can be deterministic or probabilistic; latter one takes into account the uncertainty)
  - User interface = program that helps the user to communicate with the program
- Rules are independent of each other
- System is quick and reliable
- Often programmed using Prolog

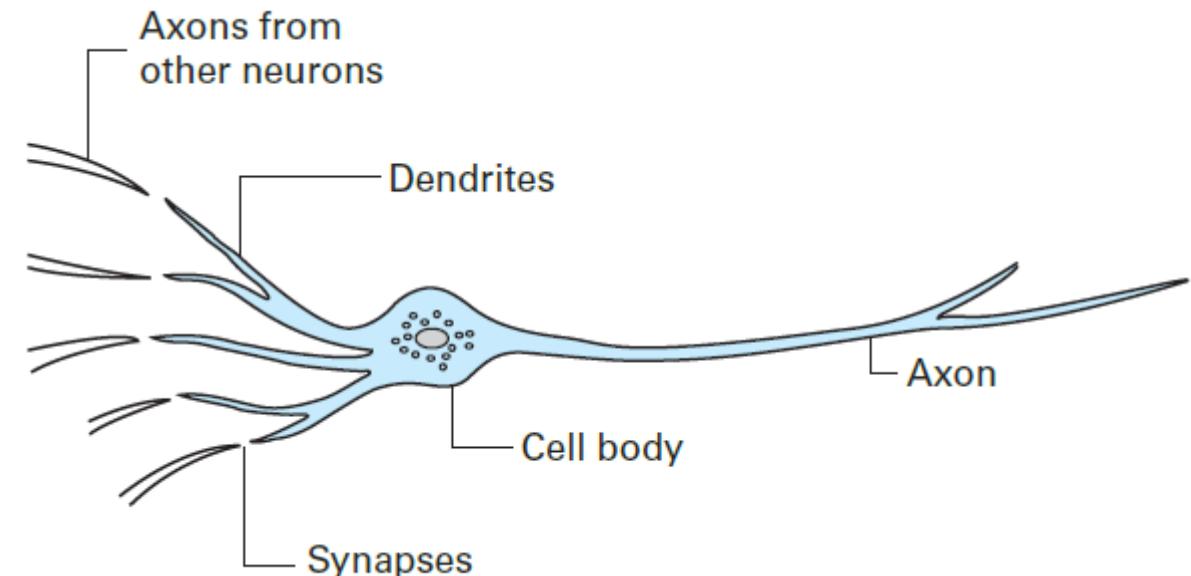


# Learning

- One requirement for good AI is capability to learn new things
- Approaches to computer learning can be classified based on the level of human intervention needed:
  - Learning by imitation (human demonstrates, agent mimics)
  - Learning by supervised training (machine performs the action, human grades the success)
  - Learning by reinforcement (machine performs the actions and is able to assess the “goodness” of those actions by itself)
- Imitation is often used in computer programs (“did you mean: \_\_\_\_”)
- Supervised training is common, used for example in e-mail spam filters
- Reinforcement is a good method when the “goodness” can be calculated or otherwise assessed (game won vs. lost)

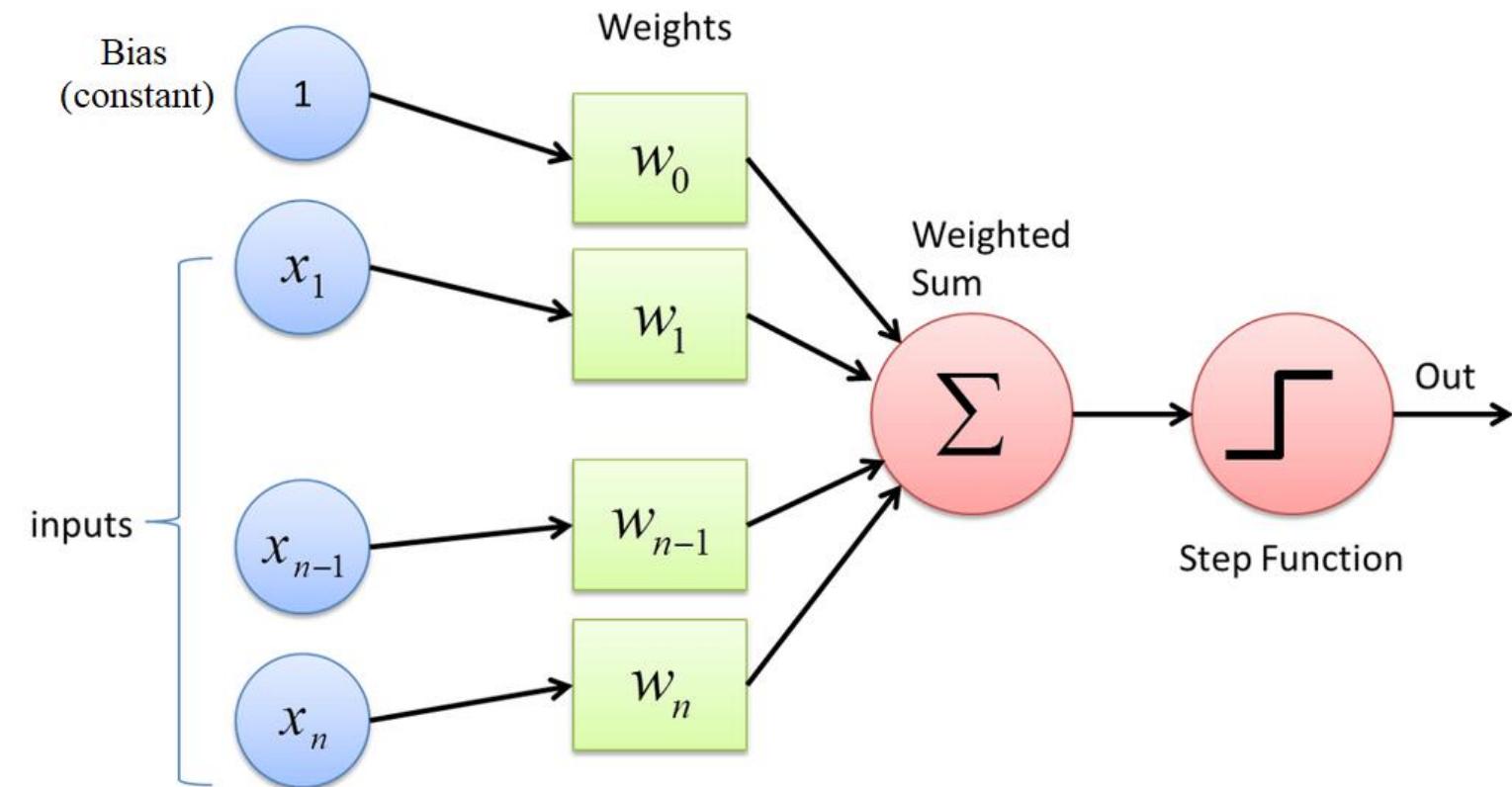
# Neural network

- The most common (albeit not the only one) method of creating a connectionist AI are artificial neural networks
- This network is a model that mimics networks of neurons in biology
- Neurons are connected to each other via synapses
  - Dendrites = “input” tentacles
  - Axon = “output” tentacle
  - Synapses = small gaps between the two
- Neuron cell has two possible states
  - Excited state
  - Inhibited state
- State is transmitted to others via the axon



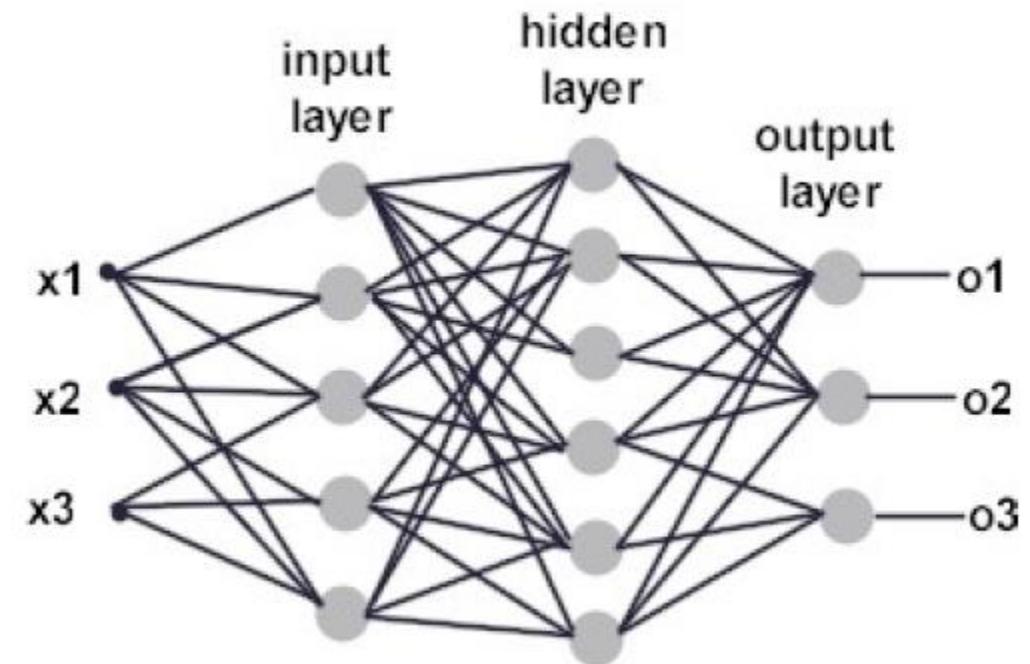
# Perceptron

- A single-layer neural network is called a perceptron
- Input values + bias are given weights
- Perceptron calculates a weighted sum of these (“effective input”)
- Effective input is compared to the threshold value
- If threshold value is exceeded, step function gives 1 as the output value (if not, the output value is 0)
- Training is done by adjusting the weights (network adjusts!)



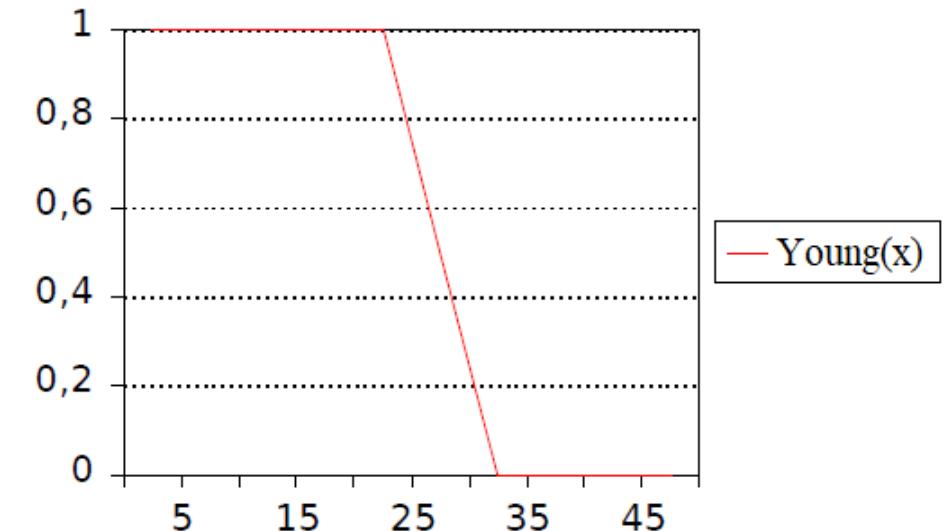
# Multi-layer perceptron

- Human body is estimated to have approximately 1000 neurons, each with around 100 synapses – so there can be multiple layers
- A multi-layer perceptron is called neural networks
- Consists of
  - Input layer
  - Hidden layer (one or several)
  - Output layer
- Increasing the number of variables in the problem increases the need for more neurons and more layers
- When properly trained, is able to learn and model the behavior of any function



# Fuzzy logic

- In fuzzy logic, truth table values for inputs may be any number in [0,1]
- Good for taking into account the impreciseness of information – and also the impreciseness of concepts!
- Example: what is classified as a “young” person?
  - Depends on who you’re asking from, but the transition age from young to older is 22...32
- Information: John is 28 years old
  - John is ”rather young”
- Useful tool when
  - Concepts are vague
  - Data is subjective



# Asimov's three laws of robotics

- Intelligent agents in AI are often considered as robots
- Here, ethics comes to play
  - What should robots be allowed to do and what they shouldn't?
- Isaac Asimov (mostly known for his sci-fi books) presented the three laws of robotics in 1942
- The order of laws is important!
- Later, Asimov also added a 0<sup>th</sup> law “Don’t harm humanity or allow humanity to come to harm”

WHY ASIMOV PUT THE THREE LAWS OF ROBOTICS IN THE ORDER HE DID:

POSSIBLE ORDERING	CONSEQUENCES	
1. (1) DON'T HARM HUMANS 2. (2) OBEY ORDERS 3. (3) PROTECT YOURSELF	[SEE ASIMOV'S STORIES]	BALANCED WORLD
1. (1) DON'T HARM HUMANS 2. (3) PROTECT YOURSELF 3. (2) OBEY ORDERS	EXPLORE MARS!  HAHA, NO. IT'S COLD AND I'D DIE.	FRUSTRATING WORLD
1. (2) OBEY ORDERS 2. (1) DON'T HARM HUMANS 3. (3) PROTECT YOURSELF		KILLBOT HELLSCAPE
1. (2) OBEY ORDERS 2. (3) PROTECT YOURSELF 3. (1) DON'T HARM HUMANS		KILLBOT HELLSCAPE
1. (3) PROTECT YOURSELF 2. (1) DON'T HARM HUMANS 3. (2) OBEY ORDERS	I'LL MAKE CARS FOR YOU, BUT TRY TO UNPLUG ME AND I'LL VAPORIZ YOU. 	TERRIFYING STANDOFF
1. (3) PROTECT YOURSELF 2. (2) OBEY ORDERS 3. (1) DON'T HARM HUMANS		KILLBOT HELLSCAPE

# Challenges of AI

- Despite massive leaps during the last decades, AI is still miles away from being even close to the level of humans
- Many challenges exist – some technological, some not so much:
  - AI still needs a LOT of computing power
  - Private funding is not easy to get, because aside from large technology companies, implementation possibilities of AI are not well understood in the business sector
  - It's hard to get people excited about computers that can perform tasks which almost any human being can do better (especially a problem with computer vision)
  - Trust issues - hackers may find surprisingly simple ways to fool AI
  - Ethical and legislative questions about rights and responsibilities of agents need serious consideration
  - If strong AI can be developed, will science fiction become science fact?

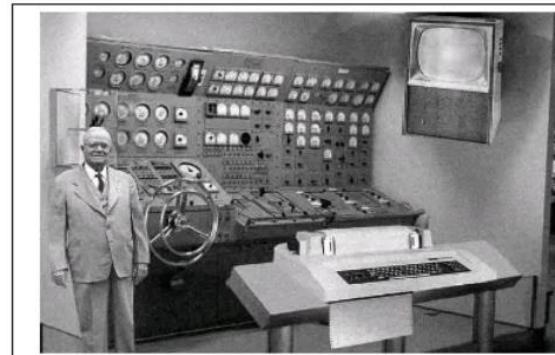
TO PROVE YOU'RE A HUMAN,  
CLICK ON ALL THE PHOTOS  
THAT SHOW PLACES YOU  
WOULD RUN FOR SHELTER  
DURING A ROBOT UPRISING.



# Thank you for listening!



# 12. History and future of computer technology



First Generation



Second Generation



Third Generation



Fourth Generation



Fifth Generation

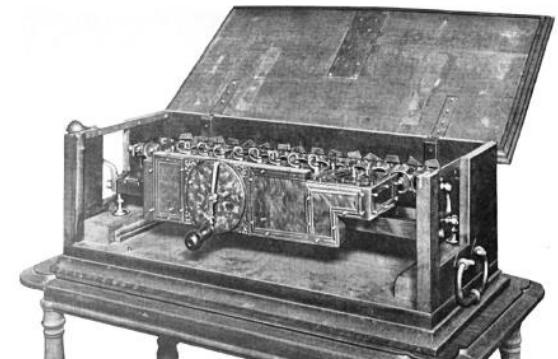
# Generations of computers

- The origin of computers can be traced to calculating devices
- Computing technology has not developed steadily; biggest advances have been made due to novel inventions of some new technology
- History of computers can be divided to generations according to these technological breakthroughs

	Time Period	Defining Technology
<b>Generation 0</b>	1642–1945	Mechanical devices (e.g., gears, relays)
<b>Generation 1</b>	1945–1954	Vacuum tubes
<b>Generation 2</b>	1954–1963	Transistors
<b>Generation 3</b>	1963–1973	Integrated circuits
<b>Generation 4</b>	1973–1985	Very large scale integration (VLSI)
<b>Generation 5</b>	1985–????	Parallel processing and networking

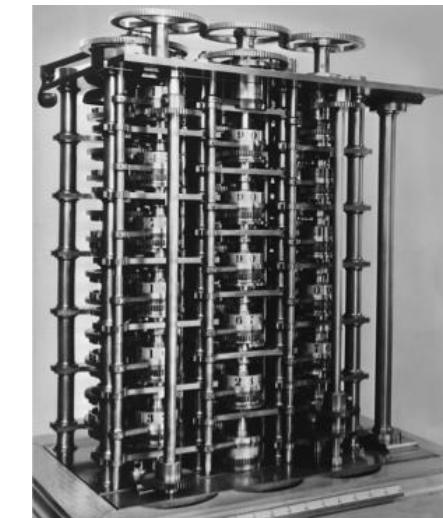
# Mechanical calculators

- “Gen 0” refers to mechanical devices
- The first calculator in the world was developed by Wilhelm Schickard in 1623 (“calculating clock”)
  - Napier’s bones for multiplication, gears for addition/subtraction
  - Destroyed in a fire, replica built in 1960s
- Pascal’s mechanical calculator, 1642
  - Mechanical gears, turned by hand
  - Could handle 8-digit numbers, but only addition & subtraction
- Leibnitz’s stepped reckoner, 1694
  - Stepped drum (teeth differ in length) and gears
  - All four basic arithmetic operations + square root
  - Same idea was later used in Curta Calculator (popular after WW2)



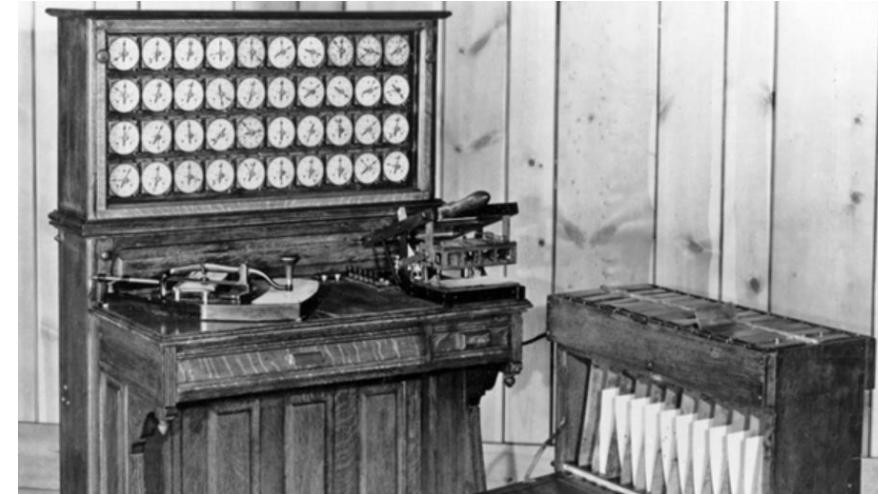
# Programmable mechanical machines

- First machine ever that could be programmed was, surprisingly, not a calculator, but a weaving machine: Jacquard's loom in 1801
  - Weaving patterns were specified by cards with punched holes; hooks passed through the holes to raise specific threads, creating the desired pattern
  - Enabled mass-production of tapestries and other fabrics
- Charles Babbage's Difference Engine, 1821
  - Engine for calculating table values for logarithms and trigonometric functions
  - Only addition and subtraction; performed iterations
  - Steam-powered; not fully constructed due to 19<sup>th</sup> century limitations in manufacturing technology (only a limited prototype built)
  - "Difference Engine 2" was built by Science Museum of London in 1991 according to Babbage's original drawings; works like a charm!
- Babbage's Analytical Engine, 1837
  - Programmable (with punch cards), arithmetic operations AND comparison, square root, loops, conditional jump statements...
  - Never finished due to high ambitions; not even a prototype



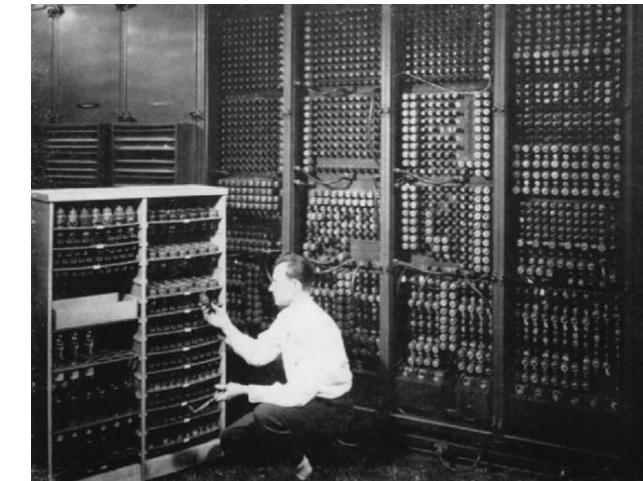
# First electrical computers

- Herman Hollerith's tabulating machine, 1890
  - Used to tabulate data for the 1890 U.S. census (in 6 weeks!)
  - Information was input via punch cards
  - Metal pegs passed through holes and made electrical connection with a metal plate
  - Hollerith's company (founded in 1896) later became IBM
- First computer using electromagnetic relays in 1930s
  - Built by Konrad Zuse in Germany (destroyed during WW2)
- Harvard University's Mark I, 1944 & Mark II, 1947
  - Basic arithmetic operations & trigonometric functions
  - 72-number memory; 10 additions/second
  - Slow, but still 100 times faster than other alternatives



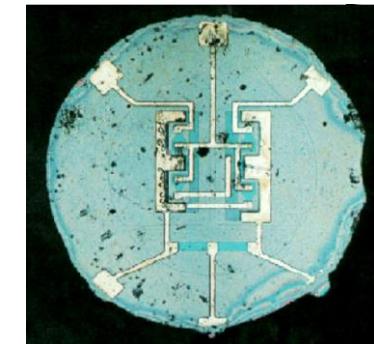
# Vacuum tube computers

- All generation 0 computers had moving parts, so the computing speed was limited by inertia
- Relays started to get replaced by vacuum tubes in 1940s
  - Invented in 1906, but manufacturing was costly prior to WW2
  - No moving parts, only the electrons move
  - Enabled switching of electric signals up to 1000 times faster
- COLOSSUS, 1943
  - Built by the British government for one single purpose: decoding Nazi military communications
  - 2300 vacuum tubes; could read & interpret 5000 characters/second
- ENIAC, 1946
  - Only 20-number memory, but 5000 additions/second
  - 18 000 vacuum tubes, weight 30 tons, power consumption 140 kW



# Transistors and integrated circuits

- Transistor, 1948
  - Piece of silicon, whose conductivity can be controlled by electric current
  - Smaller, cheaper, more reliable & required less power than vacuum tubes
  - Resulted in significant reduction in size & cost of computers
- Integrated circuit, 1958
  - Even though transistors could be made small, their size had to be large enough to allow soldering of wires
  - This changed when a technique for packaging transistors and their circuitry (as copper lines) on a single chip of silicon was developed
  - Another huge leap in miniaturization & cost reduction
  - Allowed mass production of IC chips
- Both these discoveries were so groundbreaking that their inventors were awarded Nobel prizes in Physics (1956 & 2000)
- These advances coupled with von Neumann architecture (stored program concept) led to modern programmable computers



# Effects of miniaturization

- Miniaturization of computer components has multiple advantages
- If we're able to scale down transistor size by a factor  $K$ , it results in
  - Component size:  $A \sim 1/K^2$
  - Power consumption:  $P \sim 1/K^2$
  - Product of power and minimum pulse width:  $PT_{min} \sim 1/K^3$
  - Power consumption per area unit:  $P/A \sim 1$
- So, smaller components consume less energy and work faster
- Decreased physical size is not a bad thing, either

"If the automobile had followed the same development cycle as the computer, a Rolls Royce would today cost \$100, get one million miles to the gallon, and explode once a year, killing everyone inside."

-Robert X. Cringely (InfoWorld columnist pseudonym)

# VLSI and the personal computer revolution

- Next big jump was the first one which didn't happen due to some new discovery on the field of CS but due to advances in manufacturing technology
  - Already the 3<sup>rd</sup> generation saw production of microprocessors
  - Now, millions of transistors could be fitted on a single IC chip
  - This was named Very Large Scale Integration (VLSI)
  - Enabled cost-effective mass production of microprocessors
  - Current transistor count: 10 200 million (Intel i7-12700H)
- Emergence of companies that started selling personal computers (“PC” is a brand name of IBM) for consumers
  - MITS Altair 8800 kit, 1975
  - Apple, 1977
  - IBM PC, 1980

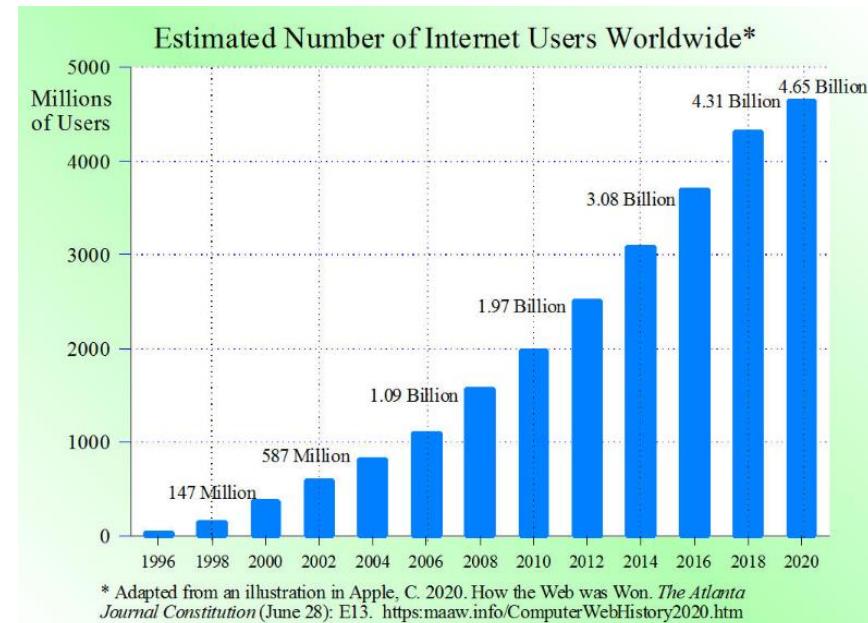
Year	Intel Processor	Number of Transistors <sup>4</sup>
2009	Quad Core Itanium	2,000,000,000
2006	Core 2 Duo	291,000,000
2000	Pentium 4	42,000,000
1999	Pentium III	9,500,000
1997	Pentium II	7,500,000
1993	Pentium	3,100,000
1989	80486	1,200,000
1985	80386	275,000
1982	80286	134,000
1978	8088	29,000
1974	8080	6,000
1972	8008	3,500
1971	4004	2,300



# Parallel processing & networking

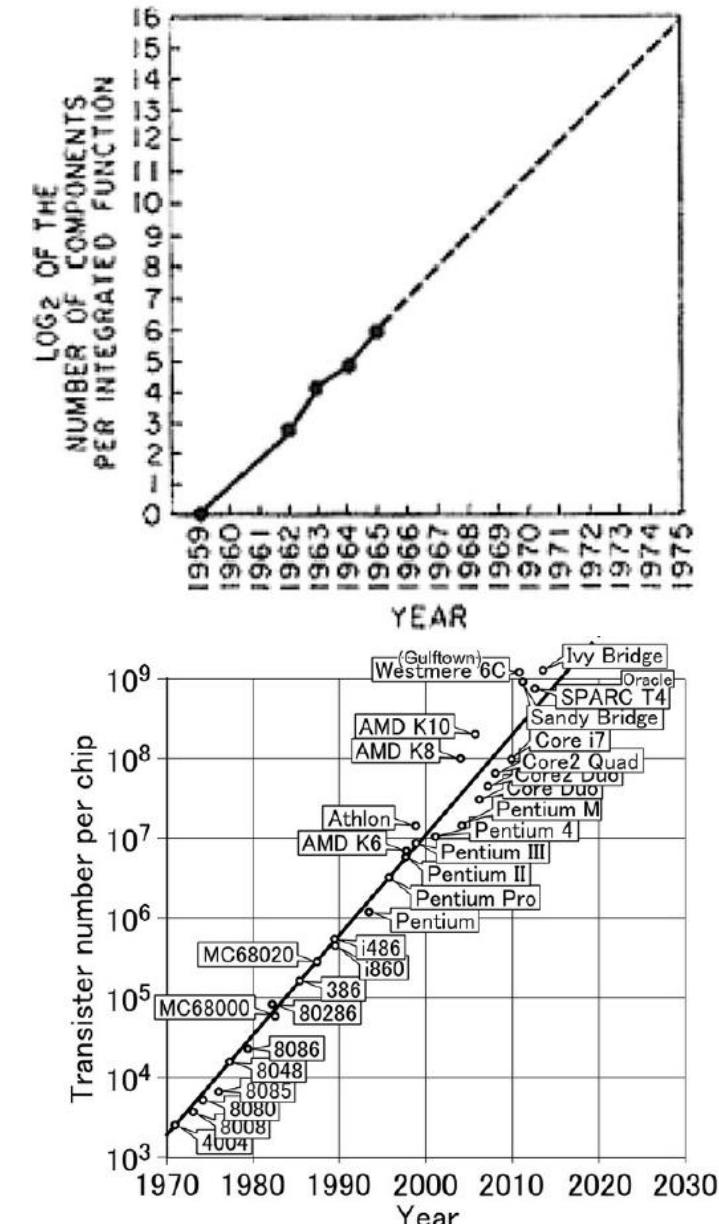
- Latest huge leap has been made in parallel computing – first via networks
- Small networks in large businesses in 1960s
- First large-scale computer network: ARPANet, 1969
  - Originally limited to government & academia
  - Later became “the Internet” we all know
  - Emergence of software platform World Wide Web (WWW), made public in 1991, immensely increased the popularity of Internet
  - Web pages could be explored by browsers
- Physical parallelism emerged later
  - Multi-processor supercomputers & servers
  - Multi-core processors in consumer devices

Year	Computers on the Internet <sup>5</sup>	Web Servers on the Internet <sup>6</sup>
2010	758,081,484	205,368,103
2008	570,937,778	175,480,931
2006	439,286,364	88,166,395
2004	285,139,107	52,131,889
2002	162,128,493	33,082,657
2000	93,047,785	18,169,498
1998	36,739,000	4,279,000
1996	12,881,000	300,000
1994	3,212,000	3,000
1992	992,000	50



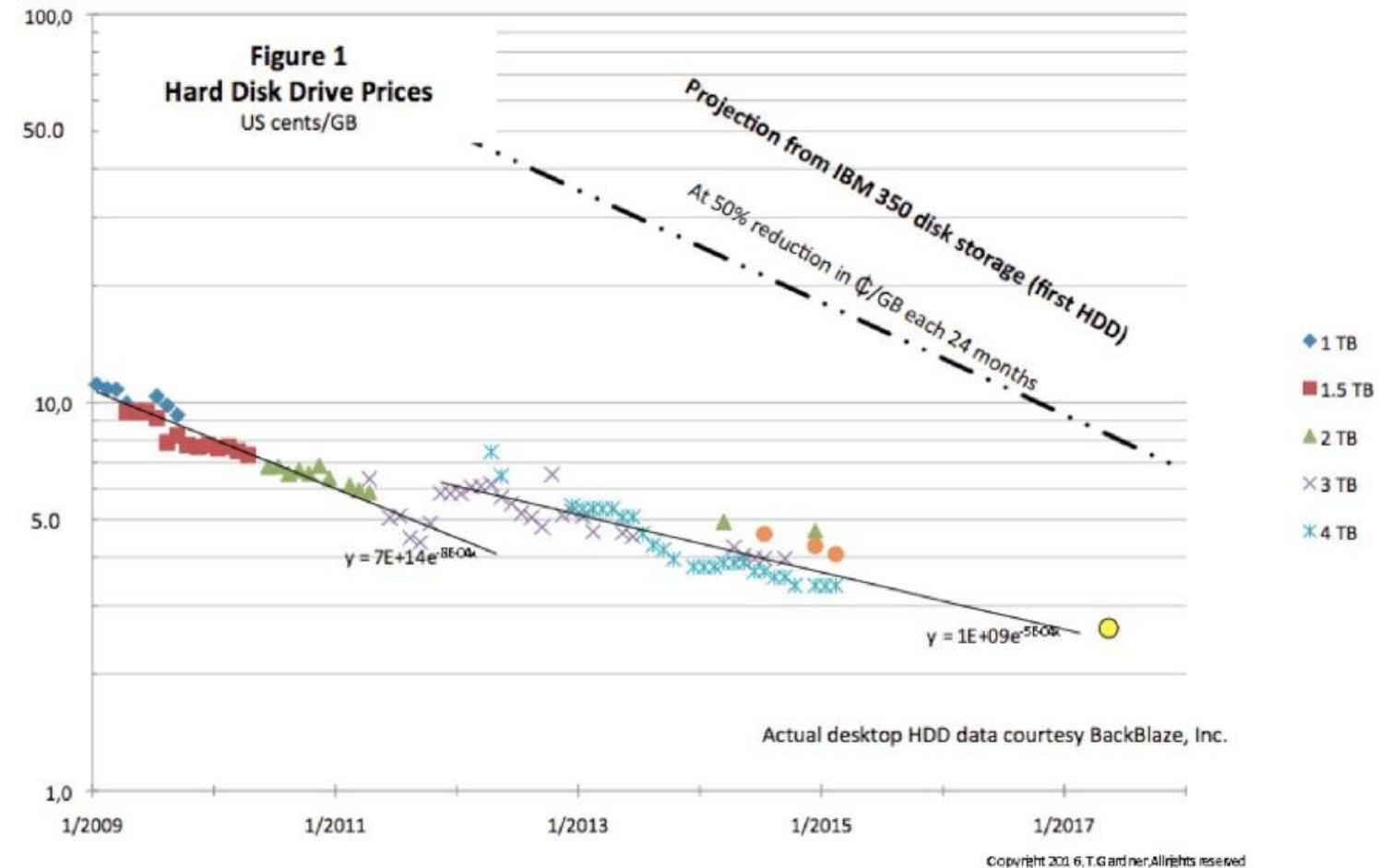
# Moore's law

- Gordon Moore presented the claim that the number of transistors that can be (at reasonable costs) fitted on a single IC chip doubles each year in 1965
  - Later in 1975 he revised this as “doubles in 2 years”
- This means exponential growth, which can't be sustainable
  - "No exponential is forever ... but we can delay 'forever.'"
  - Future holds several obstacles – starting from cost-effectiveness of manufacturing technology and physical limitations
- So far the “law” has held its ground surprisingly well
- Great visualization here:
  - <https://www.visualcapitalist.com/visualizing-moores-law-in-action-1971-2019/>



# Advances in storage capacity

- Another feature of computers that has undergone major advances is storage capacity
- IBM 350 disk storage was introduced in 1956; after that, a prediction was made that unit price of storage would halve each 2 years
- The current situation is way ahead of this (although the trend is slowing down)



# Obstacles for continuation of miniaturization

- Physical limitations:

- Kinetic theory of thermodynamics
- Heisenberg's uncertainty principle
- Irradiance
- Tunneling
- Speed of light

$$\begin{aligned} E_{\text{rw}} &\rightarrow kT \\ \Delta x \Delta p &\geq \hbar \quad \& \quad \Delta E_{\text{rw}} \Delta t \geq h/4\pi \\ E_{\text{rw}} f n &\leq 100 \text{ W/cm}^2 \\ \Theta &\propto |\Psi|^2 > 0 \\ v_d &= \mu_e E < c \end{aligned}$$

- Limitations of manufacturing technology:

- Optical lithography
- Materials

$$\begin{aligned} R = k_1 \lambda / NA \quad \& \quad DOF = k_2 \lambda / NA^2 \\ \text{Si, SiO}_2, \text{Ge, GaAs} \end{aligned}$$

- Economical limitations:

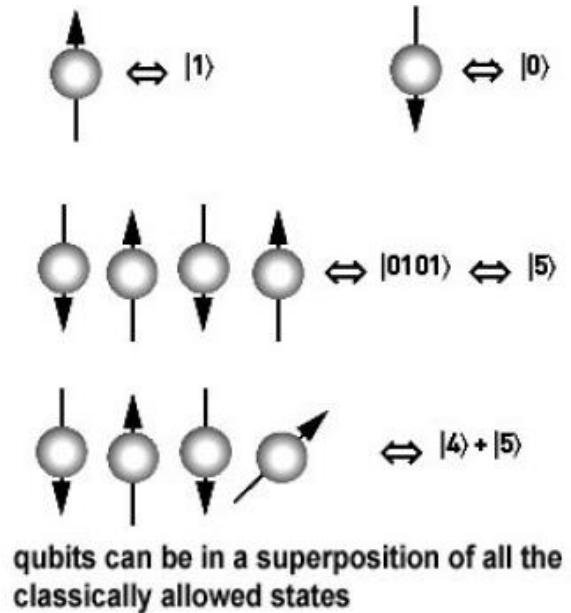
- Unit price of a microchip ( $\mu\text{€}$  per feature)
- Price of a microchip factory (5 to 20 billion dollars)

# Possible future computers

- Progress of computer performance has been mind-blowingly quick, but for this pace to continue, new groundbreaking advances would be needed soon
- One possibility would be to find a completely new way to implement a computer
- Some possibilities could include:
  - Nanocomputer (made from nanotransistors - using carbon nanotubes; would allow miniaturization to continue longer)
  - DNA computing (biocomputer based on DNA calculation, demonstrated by Len Adleman already in 1994; slow, but can process almost an infinite amount of actions in parallel fashion)
  - Optics: optical computer (electricity is replaced by light – so, photons instead of electrons; a company named Xanadu is working on a photonics-based quantum computer)
  - Quantum computer (instead of bits there are qubits)
  - Something yet to be invented?

# Quantum computer

- Qubits are quantum bits – a superposition of two quantum states
  - A qubit can include 1, 0 or both at the same time
  - A series of qubits can denote a superposition of multiple binary numbers at the same time
- In a quantum computer, many numbers are in superposition state  
 → computer performs the operations for all these in parallel fashion
- The state of a qubit has an effect on all other qubits without delay  
 - no matter how far away from each other they are
- Quantum computers have been built and they are in use today
  - Current record: 127 qubits (Intel Eagle) – Intel plans to hit 4 digits in 2023
- Some problems are still to be solved, though:
  - Usage temperature must be close to absolute zero in order to prevent overheating
  - Superpositions collapse at the time of measurement, so quantum computers can only solve problems that can be solved by asking just one question



# Current (and future?) trends of computer technology

- Mobile computers and wearables
  - Mobile services, geolocation
- Cloud services for data storage & computing
  - User doesn't have to be next to a supercomputer physically
  - Web-based programs that run on browser (no need to install anything)
- Artificial intelligence and computational methods
  - Analyzing and visualization, data mining
  - Humane user interfaces, robotics
- All-encompassing computer technology
  - Computers take such forms that people don't think of them as computers anymore
- Adaptation to computer technology
  - Almost everything can be done in remote fashion



# Summary

- Need for computers emerged from difficulties with calculation, which harmed the computability of some problems
- The base requirement for computability of a problem is that there exists some kind of an algorithm that can solve it
  - ...and the practical requirement is that it should be done in reasonable time
- If the complexity of our problem is high, we have two options:
  - Develop a better algorithm with lesser complexity
  - Improve the computation power of the computer used for solving the problem
- Computer science as a discipline actively seeks advances in both these areas
  - Computer technology mainly works on the latter
- Stripped to its essentials, computer technology is just implementation of logic gates and circuits as effectively as possible
  - Current design relies on very large-scale integrated circuits and their parallel processing

# Thank you for listening!

