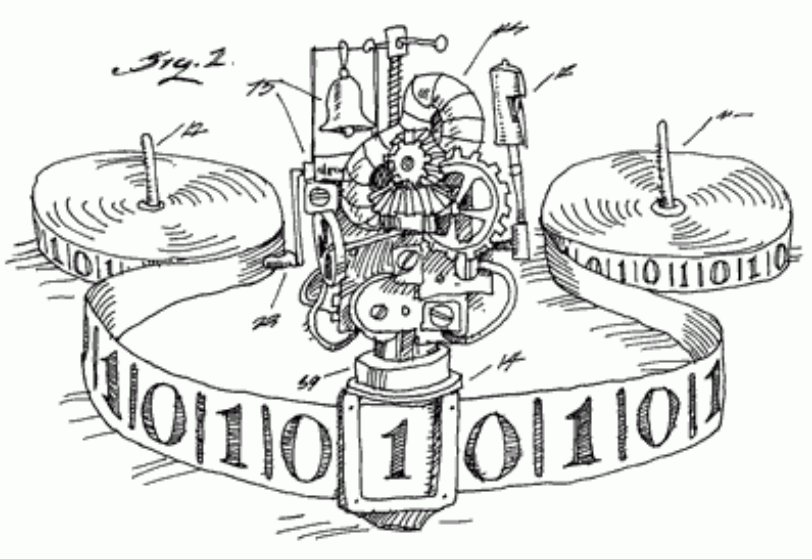
INFO 101 – Introduction to Computing and Security

[2020 - Week 1 / 2]



Prof. Dr. Rui Abreu

University of Porto, Portugal

rui@computer.org

@rmaranhao

Von Neuman Computer Architecture

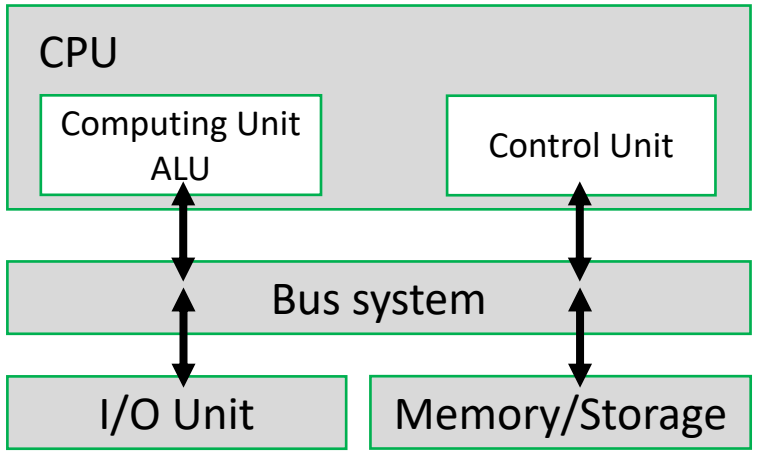
Or how current computers are organized internally

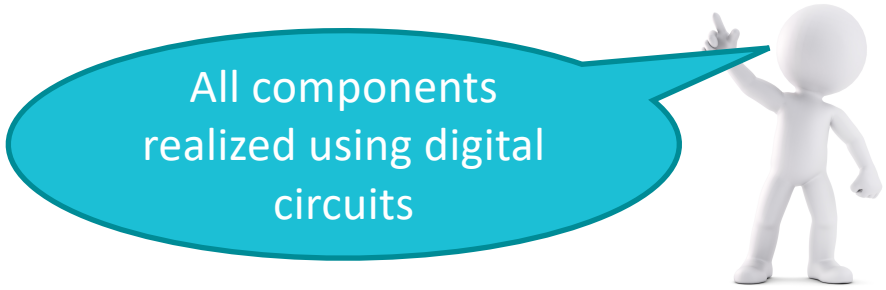
COMPUTER HARDWARE

- The hardware of a computer comprises:
  - The computer itself
  - Peripheral devices:
    - Input devices like keyboards, and computer mouse
    - Output devices: computer screen, printer
    - Other equipment like SSD, hard disk, DVD drive, touch screen
- John von Neumann
  - 1945 architecture behind a general computer system
  - Innovation: programs are stored in the memory

VON-NEUMANN's architecture components

- Central Processing Unit (CPU)
  - Arithmetic-Logic Unit (ALU)
  - Control Unit
- Memory
- Input/Output (I/O) Unit
- Bus system



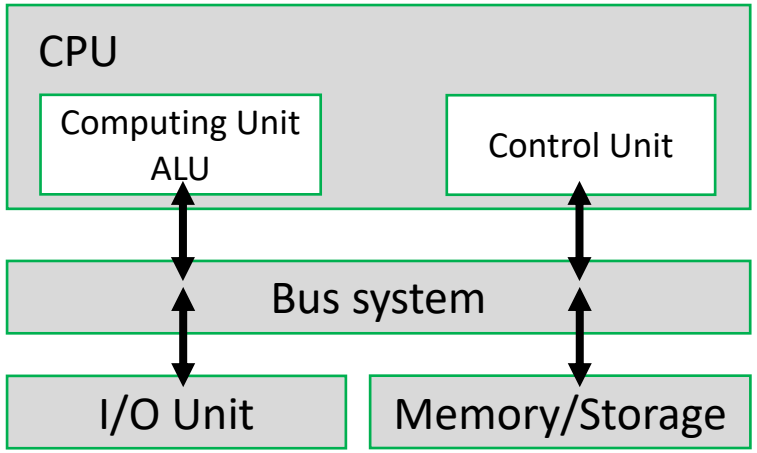


Properties of VON-NEUMANN architecture

- Structure is independent of the underlying problem to be solved
- Memory comprises cells of equal size that are numbered successively
  - Using the memory address we are able to access its content
- Data and programs are stored in the same memory
- A program is a sequence of statements that are
  - Executed sequentially
  - Placed in the same order in the memory
- Deviations from the memory location are possible using branches
- The computer works using machine code

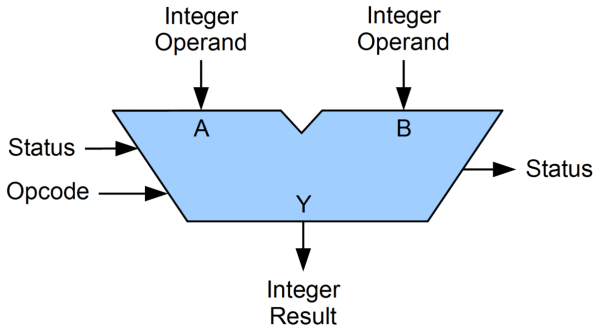
Address	Data
0	00101110
1	00011111
2	10000100
3	00000000
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000011
15	00001110

CPU



- Processor (kernel)
- Objective: Execute programs
  - Program is a sequence of machine code instructions
    - E.g. Add two numbers
  - The processor controls the execution of instructions in the expected sequence
- CPUs comprise:
  - Control Unit: Reads instructions sequentially and executes them using an instruction table.
  - **ALU**: Executes logic and arithmetic operators / instructions
  - Register:
    - In CPU integrated memory (fast)

Arithmetic-Logic Unit



- Two inputs: Integer operands (e.g. each 8 Bit)
  - Operation code (opcode;e.g. 4 Bit):
    - Semantics, e.g.: 1000 = add, 1100 = subtract
    - Determines the calculation that has to be performed
- Output = Integer (e.g. 8 Bit)
- General Purpose Register
  - Special memory inside a CPU, to handle temporary operands and the results of operations
  - Operations can be performend – in many cases – only on register values
- Status register (status flags) comprises results of comparison operations, and the status of the calculation (overflow bit, zero bit, signature bit)
    - Overflow: 11 + 01 = 100 (but if we have only 2 bits 100 cannot be represented!)

Control Unit

- Coordinates data exchange with main memory
- Fetches commands from main memory and controls program execution
- Parts of a control unit:
  - The program counter holds the address of the next command to be fetched from the main memory
  - The instruction register holds the current command/statement to be executed
  - Clock determines the speed of calculation and command fetching

CPU

- The whole computation requires the ALU and the control unit!
- **Example**: Add two values stored in the main memory.
  - Fetch the content of the first memory cell from the main memory and store it in one register.
  - Fetch the content of the second memory cell from the main memory and store it in another different register.
  - Activate the ALU, taking the two registers as inputs and store the result in another register.
  - Store the register value as a final result in another memory cell.

Main memory

- Main Memory (e.g. RAM):
  - Stores programs and data
  - Is a set of addressable memory cells (of e.g. 8 bit)
- Every memory cell has a unique address
- Usually main memory is volatile, i.e., when turning off the power supply all data are gone.
  - There is also non-volatile or persistent memory: harddiscs, CDs, flash-memory, or NVRAM
- From a technical point of view: Main memory is a collection of switches to store 0 and 1.
  - Multiplex function comprising flip flops circuits

BUS SYSTEM

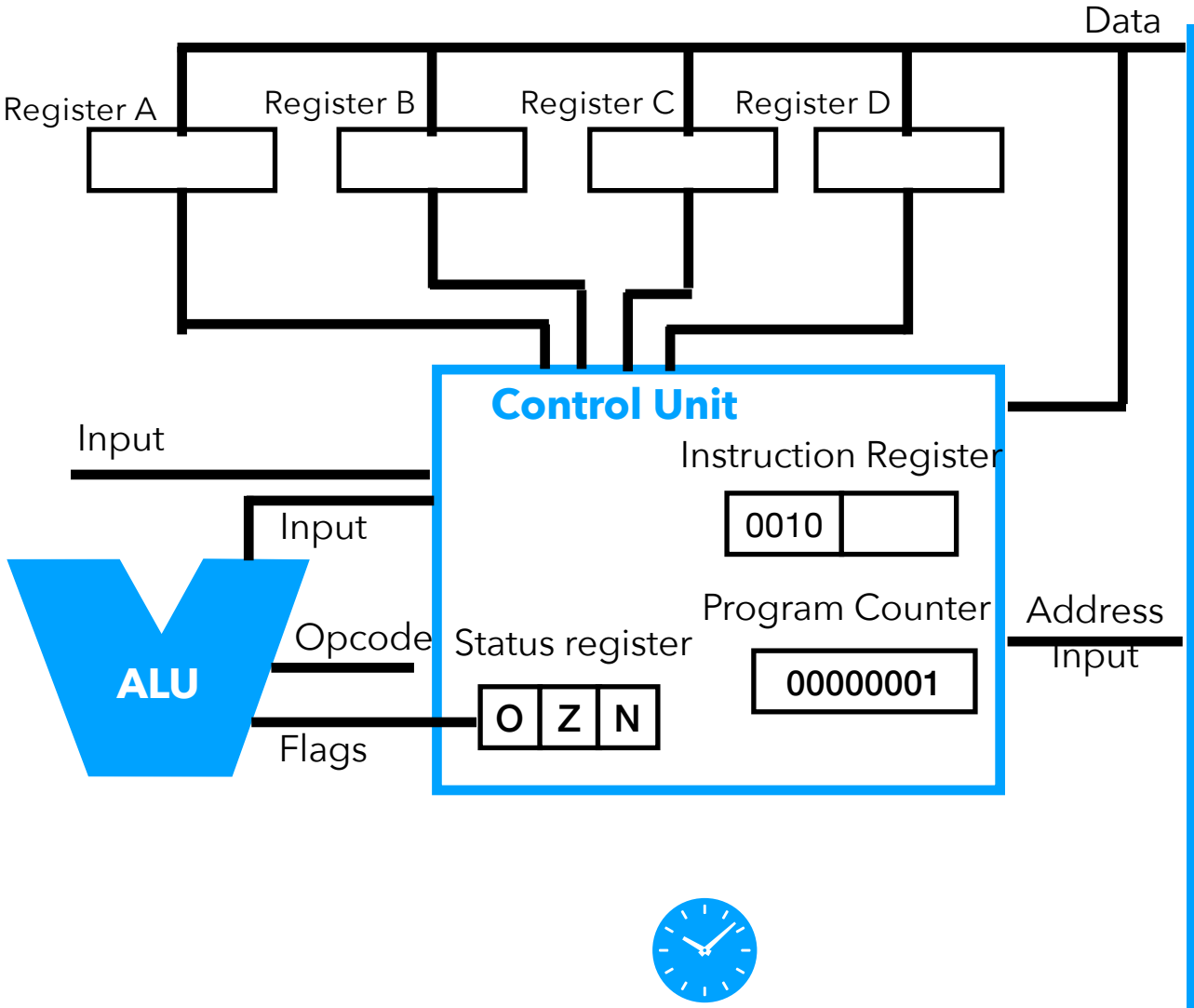
- Components of a computer are communicating using a bus system
- The bus is used to transport data units (of size of typical 8, 16, 32, 64 bit)
- The bus can usually be accessed only exclusively from one component at a specific time. This is handled using a bus protocol
- A bus comprises the following components:
  - Data bus: communication of data to and from the main memory
  - Address bus: Used to specify the addresses of the memory cells
  - Control bus: Used to specify control information
    - Which functional unit has to be accessd? Read or write access? Etc.

I/O Unit

- Communication between the computer/CPU and its peripherial devices like printers etc.
  - Usually makes use of I/O controllers
- Controller:
  - Provide the interface for communication between different I/O devices
  - Are designed for a specific purpose
  - Are usually connected with the bus system to communicate with the CPU and memory

Example: Transmit data from the main memory to the CPU

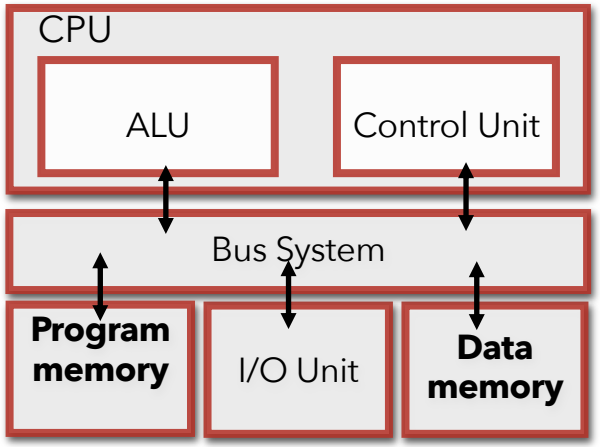
- CPU puts the memory address on the address bus.
- CPU puts the signal „READ“ on the control bus.
- CPU puts the signal „ADDRESS VALID“ on the control bus.
- Main memory in passive mode monitors the control and address bus and detects a valid address.
- Main memory checks the direction of transport on the control.
- Main memory puts the content of the memory cell of the given address on the data bus.
- Main memor puts the signal „DATA READY“ on the control bus.
- CPU detects „ DATA READY“, fetches the data from the data bus, and stores them in a register.
- CPU puts the signal „DATA FETCHED“ on the control bus.
- Main memory deactivates the „DATA READY“ signal.
- CPU deactivates the used address and control bus.



Memory

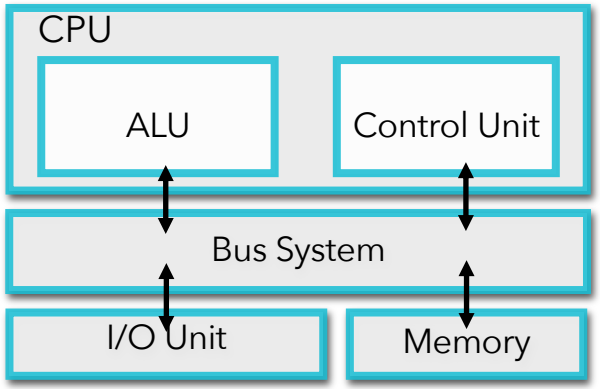
Adress	Data
0	00101110
1	00011111
2	10000100
3	00000000
4	00000000
5	00000000
6	00000000
7	00000000
8	00000000
9	00000000
10	00000000
11	00000000
12	00000000
13	00000000
14	00000011
15	00001110

Harvard-Architecture



- Physical separation of data and program memory
- Commands and data can be fetched and stored at the same time
- Data size and command size are independent
- Separation of memory access leads to secured memory access

VON-NEUMANN-ARCHITECTURE

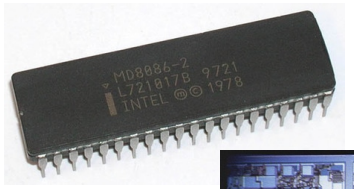


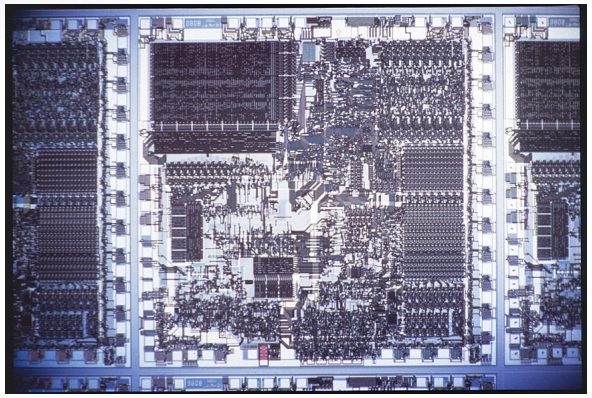
- Memory comprises both data and commands
  - Same storage place for data and programs
  - Data and commands cannot be fetched at the same time
  - Storing data can delete program commands

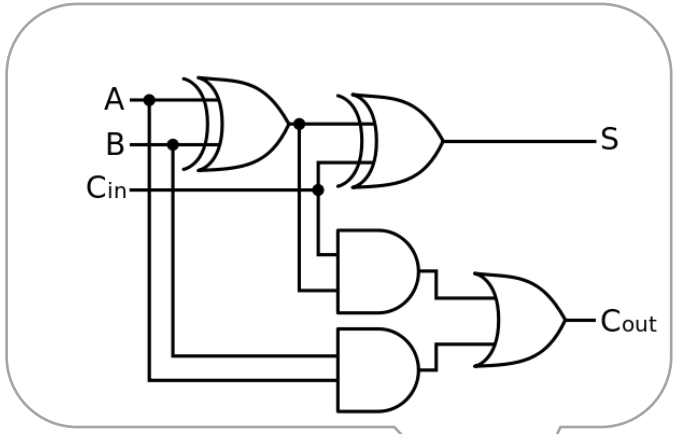
Hardware

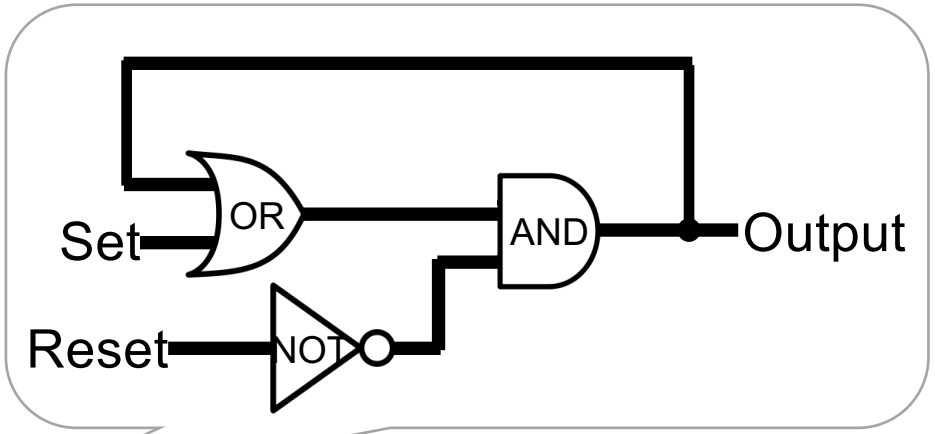
Motivation

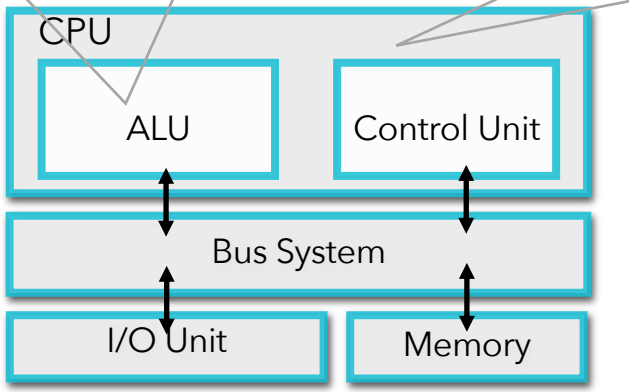
- CPUs and other Hardware implemented using digital gates!









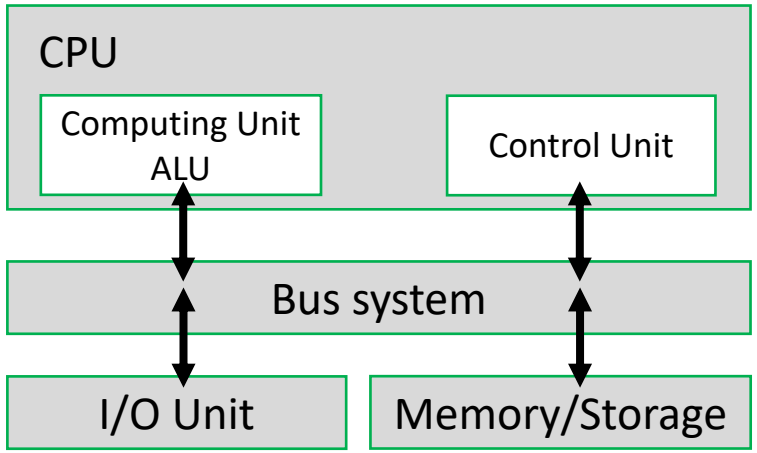


Computer Hardware

- Motherboard
  - Components
    - **CPU** (System clock, clock speed, machine cycle, machine instructions, machine language, instruction set, RISC or CISC, cache, serial processing, pipelining, parallel processing, multiprocessing, registers, word size,...)
    - Bus
    - USB (universal serial bus)
    - Plug and play
    - BIOS (basic input/output system)
    - Expansion slots (ISA, PCI, AGP, ...)
    - Network adapter card
    - Disc controller card
    - Sound card
    - Video card / graphics card
    - Serial and parallel ports
    - Storage (memory (**RAM**), **hard disk or SSD**, floppy disk, flash memory, DVD, CD, BlueRay, Zip disk, Magnetic tape,..)
- Cooling (fan)
- Power Supply

Computer Hardware

- Input
  - Keyboard
  - Mouse
  - Trackpad
  - Camera
  - Tablet
  - Microphone
  - Touch screen
  - Scanner
  - ...
- Output
  - Monitor / screen
  - Printer (laser printer, dot-matrix printer, ink-jet printer, thermal printer, plotter, photo printer,...)
  - Speaker
- Communication
  - Modem
  - WiFi
  - Lan



Boolean & Digital Circuits

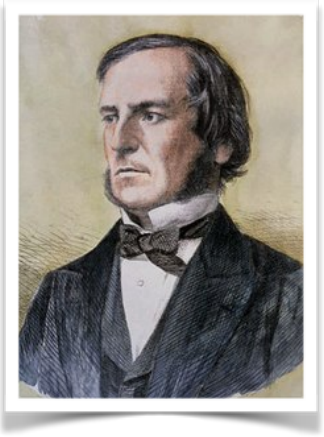
Motivation

- 0 and 1 -> Internal representation of data in a computer
- ALU – arithmetic-logic unit -> digital circuits
- Digital circuits use Boolean algebra
  - Make use of 0 and 1
  - Have a more detailed look on Boolean algebra and propositional logic (as foundations)

Wason‘s selection task (1966)

- Consider 4 cards each having on one side a letter and on the other side a number
- **Rule**: If there is a vowel (a,e,i,o,u) on one side, then there has to be an even number (2,4,6,8,...) on the other side.
- **Task**: Which cards have to be turned in order to check the rule?

Boolean Algebra and Logic



- Propositional logic (PL)
  - Origins from Aristoteles
  - Classical propositional logic:
    - Every proposition has to have been assigned either true (1,T) or false (0,F).
    - The truth value of a composed proposition, i.e., a logic formula, can be determined using the truth values of its parts.
    - In propositional logic we might represent whole sentences or partial sentences using propositions, which can be either true or false, e.g.: „On the card there is a vovel“.
- George Bool was working on digital structures:
  - In Boolean algebra there exists only two values
    - 0 - false und 1- true
    - Foundation of digital cirucits and the calculation using binary numbers
  - Boolean algebra is a special algebraic structure that represents and generalizes logical operators like AND, OR, or NOT but also set-theoretic constructs like intersection, union, or complement.

Example

- Proposition A: "9 can be divided by 3“
- Negation ¬A: “It is not the case that 9 can be divided by 3“.
  - If a proposition A is true, then ¬A is false.
  - If a proposition A is false, then its negation ¬A is true.
  - Any proposition A cannot be true and false at the same time.
  - The propositions A and ¬A cannot be true at the same time.

Boolean Algebra

- Processing of data
- Logical values = truth assignment = true (1, T, ⊤) or false (0, F, ⊥)
  - Can be represented as one bit
- On truth values the logical operators OR, AND and NOT are defined:

OR

A	B	Output
1	1	1
1	0	1
0	1	1
0	0	0

AND

A	B	Output
1	1	1
1	0	0
0	1	0
0	0	0

NOT

A	Output
1	0
0	1

TRUTH TABLE

OR Operator (Logical sum, Disjunction)

- Output is 1 (⊤) if at least one input is 1 (⊤)
- One condition using OR is always true if at least one of its parts is true.
- Formal representation of OR: + or ∨

OR

A	B	Output
⊤	⊤	⊤
⊤	⊥	⊤
⊥	⊤	⊤
⊥	⊥	⊥

Examples

OR

A	B	Output
⊤	⊤	⊤
⊤	⊥	⊤
⊥	⊤	⊤
⊥	⊥	⊥

It is winter $\vee$ *It snows* $\vee$ *You have a cold*

It is winter	It snows	You have a cold	Whole formula
⊤	⊤	⊤	⊤
⊤	⊤	⊥	⊤
⊤	⊥	⊤	⊤
⊤	⊥	⊥	⊤
⊥	⊤	⊤	⊤
⊥	⊤	⊥	⊤
⊥	⊥	⊤	⊤
⊥	⊥	⊥	⊥

OR

A	B	Output
1	1	1
1	0	1
0	1	1
0	0	0

Young $\vee$ *Old*

Young	Old	Young + Old
1	1	1
1	0	1
0	1	1
0	0	0

AND Operator (Logical product, Conjunction)

- Output is false (0,⊥) if at least one of its inputs is false (0,⊥)
- One condition using AND is always true if and only if all of its are true.
- Formal representation of AND: × or ∧

AND

A	B	Output
⊤	⊤	⊤
⊤	⊥	⊥
⊥	⊤	⊥
⊥	⊥	⊥

Examples

It is winter $\wedge$ *It snows* $\wedge$ *You have a cold*

It is winter	It snows	You have a cold	Whole formula
⊤	⊤	⊤	⊤
⊤	⊤	⊥	⊥
⊤	⊥	⊤	⊥
⊤	⊥	⊥	⊥
⊥	⊤	⊤	⊥
⊥	⊤	⊥	⊥
⊥	⊥	⊤	⊥
⊥	⊥	⊥	⊥

AND

A	B	Output
⊤	⊤	⊤
⊤	⊥	⊥
⊥	⊤	⊥
⊥	⊥	⊥

AND

A	B	Output
1	1	1
1	0	0
0	1	0
0	0	0

Young × Old

Young	Old	Young + Old
1	1	1
1	0	0
0	1	0
0	0	0

NOT Operator (Complement, Negation)

- If the input is true (1, ⊤), then the output is false (0, ⊥)
- Negates the truth value of a proposition or formula
- Formal representation: ¬A, -A, or $\overline{A}$

NOT

A	Output
1	0
0	1

EXAMPLE ¬Young

Young	¬Young
1	0
0	1

Sequences of bits

- Sequences of bits can be represented as propositions
- Operations are applied on a bit-by-bit fashion

NOT	1	0	1	1	0	1	0	1
RESULT	0	1	0	0	1	0	1	0

AND	0	0	0	0	1	1	1	1
	1	0	1	1	0	1	0	1
RESULT	0	0	0	0	0	1	0	1

MASK

➤ Used to set parts of a bit sequence to 0 (“masking”)

OR	1	1	1	1	0	0	0	0
	1	0	1	1	0	1	0	1
RESULT	1	1	1	1	0	1	0	1

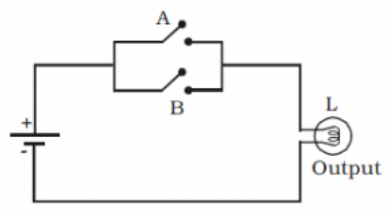
➤ Set bits to 1

Boolean Circuits

- Use simple electrical circuits comprising batteries, bulbs, and switches
  - A switch can be a transistor, a relais,…

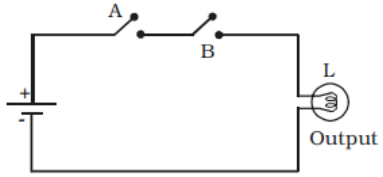
OR

A	B	Output
1	1	1
1	0	1
0	1	1
0	0	0



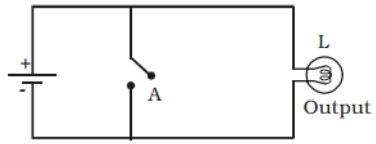
AND

A	B	Output
1	1	1
1	0	0
0	1	0
0	0	0



NOT

A	Output
1	0
0	1

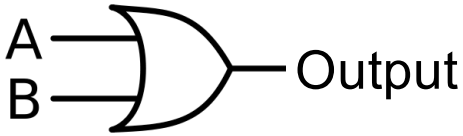


Logic Gates

- Representation ignoring its technical implemenation of Boolean circuits -> abstraction
- Logic gates might be implemented using relais, transistors, using optical devices of mechanical devices...

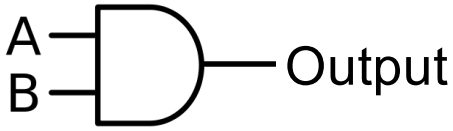
OR

A	B	Output
1	1	1
1	0	1
0	1	1
0	0	0



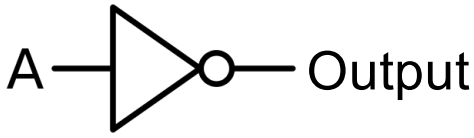
AND

A	B	Output
1	1	1
1	0	0
0	1	0
0	0	0



NOT

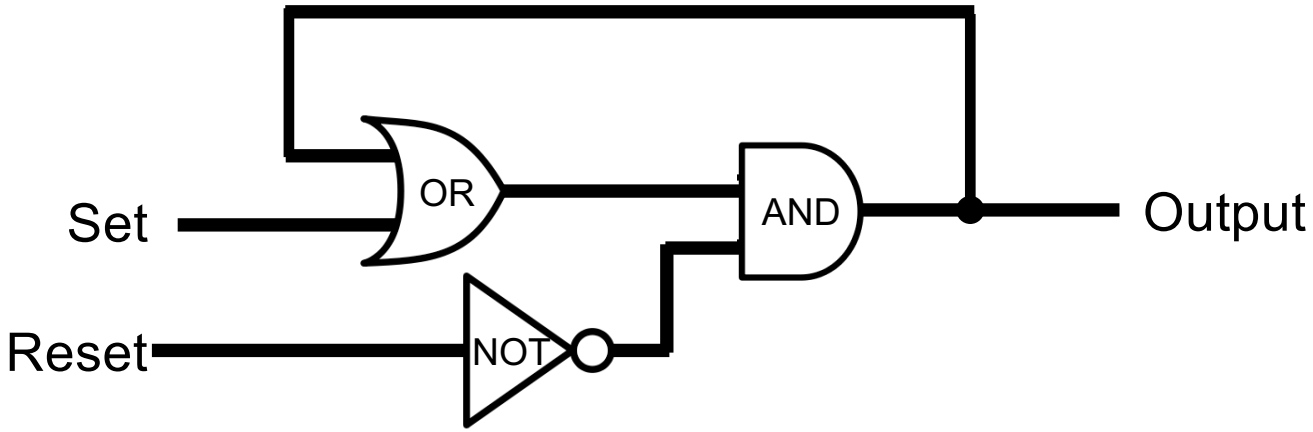
A	Output
1	0
0	1



➤ Gates are the basic building blocks of digital circuits

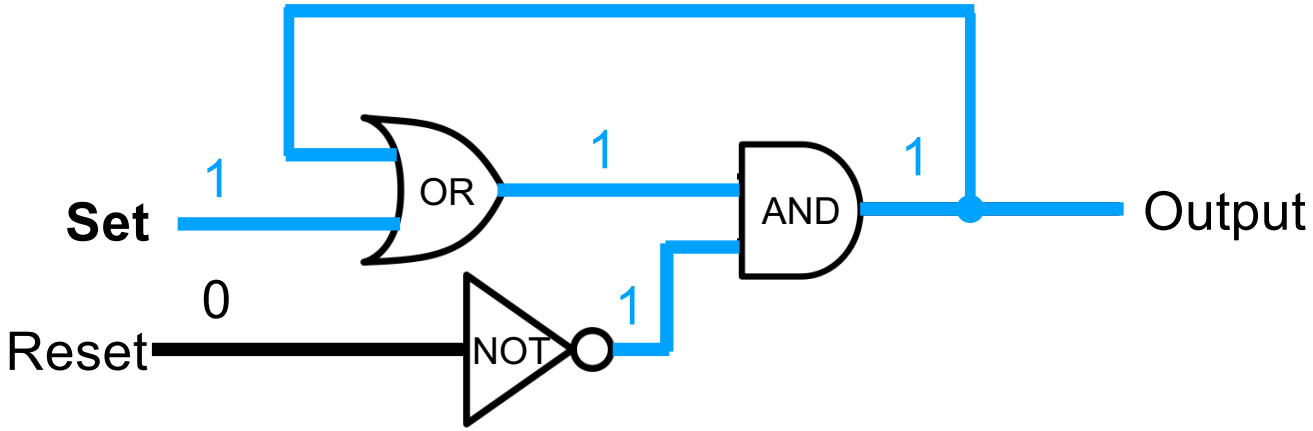
Flip-Flop

- Circuits to store bits
  - Sequentiel circuits: change their output at distinct points in time (time dependent behavior); can be asynchronous or synchronous (using clocks)



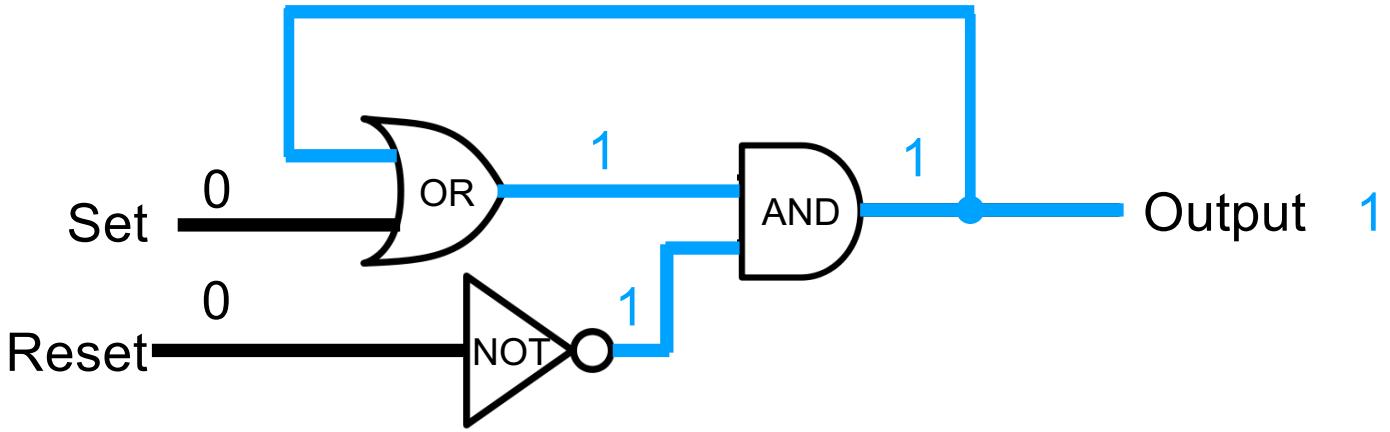
Flip-Flop

- Circuits to store bits
  - Sequentiel circuits: change their output at distinct points in time (time dependent behavior); can be asynchronous or synchronous (using clocks)



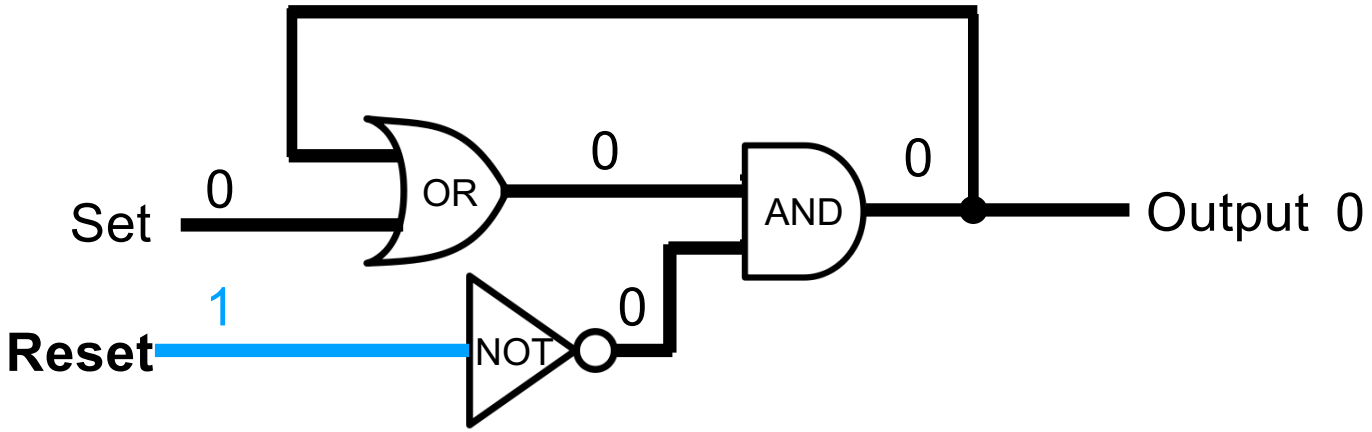
Flip-Flop

- Circuits to store bits
  - Sequentiel circuits: change their output at distinct points in time (time dependent behavior); can be asynchronous or synchronous (using clocks)



Flip-Flop

- Circuits to store bits
  - Sequentiel circuits: change their output at distinct points in time (time dependent behavior); can be asynchronous or synchronous (using clocks)

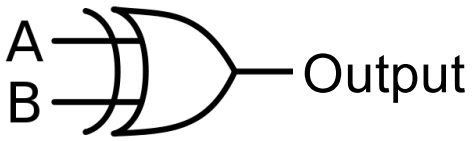


Full Adder

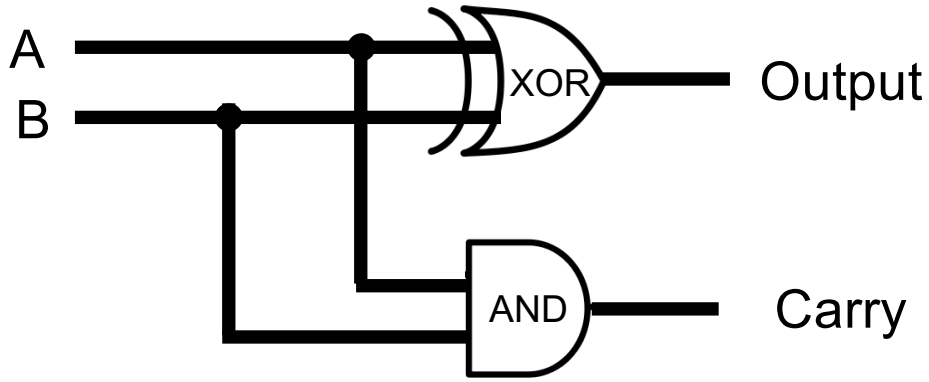
- Add two Boolean numbers using digital circuits
- Combinatorial circuit: The output depends solely on the inputs and not on time!
- Based on half adder
  - Based on exclusive or opertors (XOR)

XOR

A	B	Output
0	**0**	0
0	**1**	1
1	**0**	1
1	**1**	0



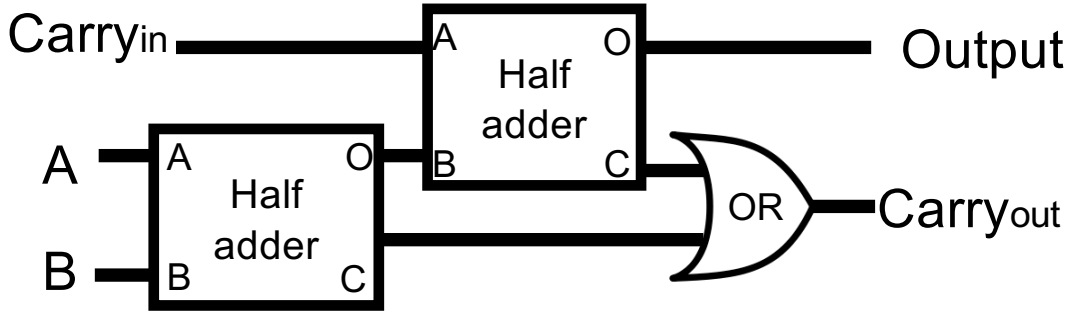
Half Adder



Half Adder

A	B	Output	Carry
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

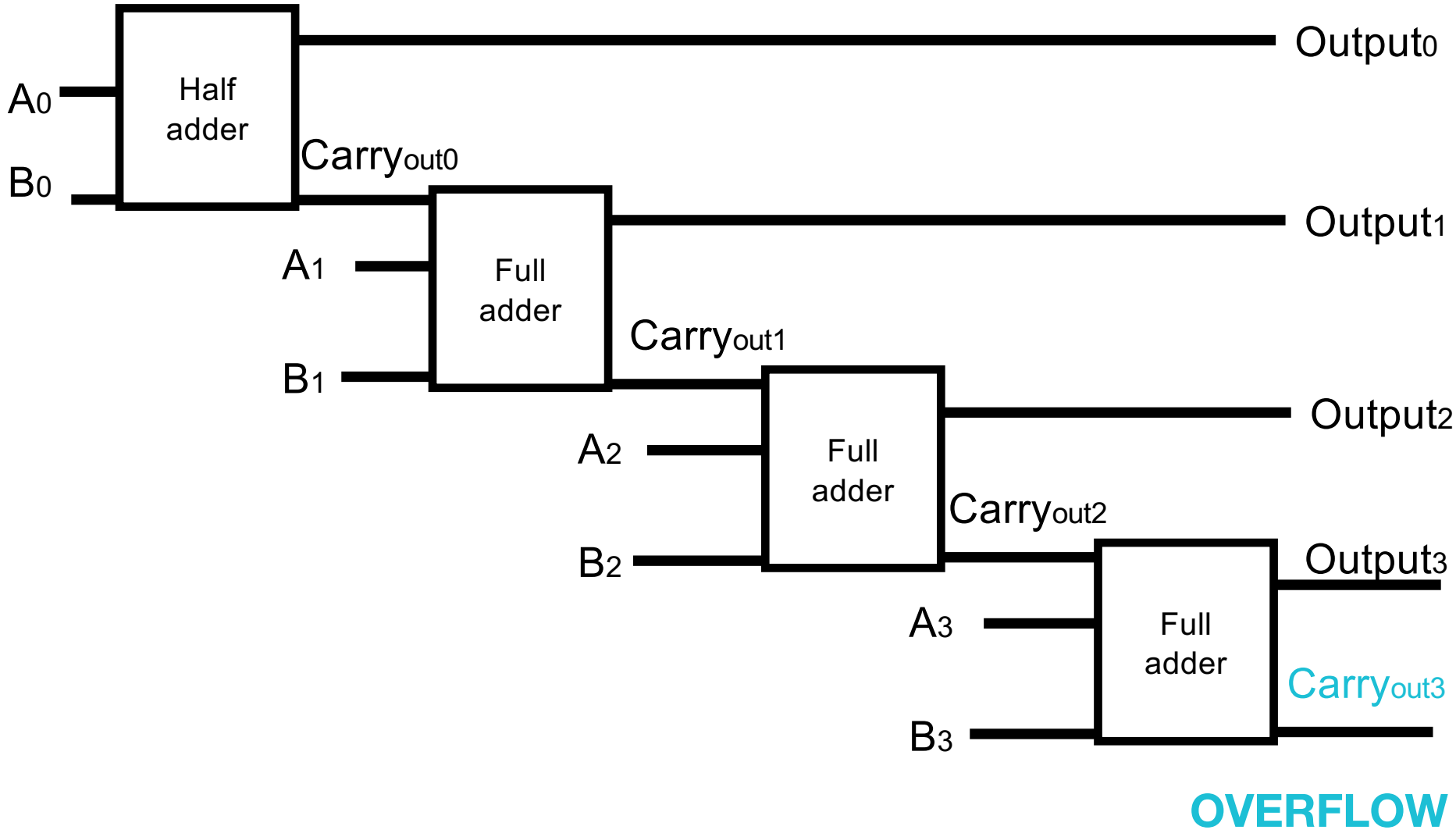
Full adder (using half adders)



Full adder

A	B	Carry$_{in}$	Output	Carry$_{out}$
0	0	0	0	0
0	1	0	1	0
1	0	0	1	0
1	1	0	0	1
0	0	1	1	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	1

4-Bit Adder


A0
B0
Half adder
Output0
Carryout0
A1
B1
Full adder
Output1
Carryout1
A2
B2
Full adder
Output2
Carryout2
A3
B3
Full adder
Output3
Carryout3
OVERFLOW

Differences between types of circuits

Combinatorial circuits

- ➤ Outputs are determined by its inputs
- ➤ Inputs are combined to compute the output
- ➤ There ***are no cylces*** (from outputs back to its inputs or intermediate

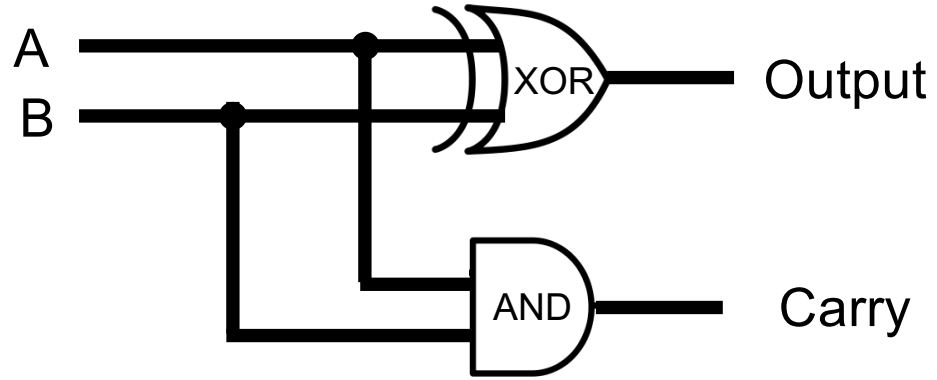
Given inputs
→
unique output values

Sequential circuits

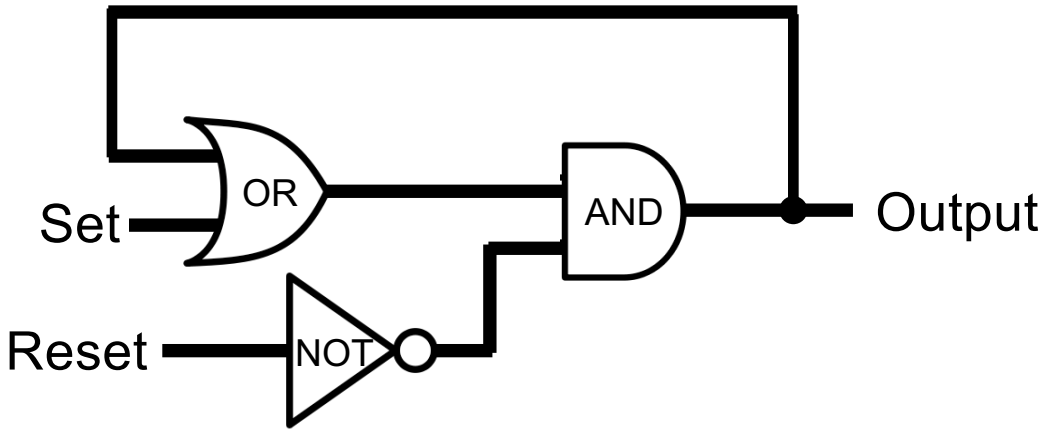
- ➤ Can store logical state (memory!)
- ➤ Additional feedback cycles
- ➤ Output is determined using the inputs and the internal state

Given inputs
→
many possible outputs depending on the internal state

COMBINATORIAL CIRCUITS



SEQUENTIAL CIRCUITS



Boolean Algebra Laws

- Sequence of applying operators
  - Negation ¬
  - Conjunction ∧
  - Disjunction ∨

$$A \vee B \wedge \neg C$$

$$A \vee (B \wedge (\neg C))$$

Duality	$\neg\bot = \top$ $\neg\top = \bot$
Idempotence	$a \vee a = a$ $a \wedge a = a$
Double Negation	$\neg(\neg a) = a$

Complements	$a \vee \neg a = \top$ $a \wedge \neg a = \bot$
Commutative Law	$x \wedge y = y \wedge x$ $x \vee y = y \vee x$

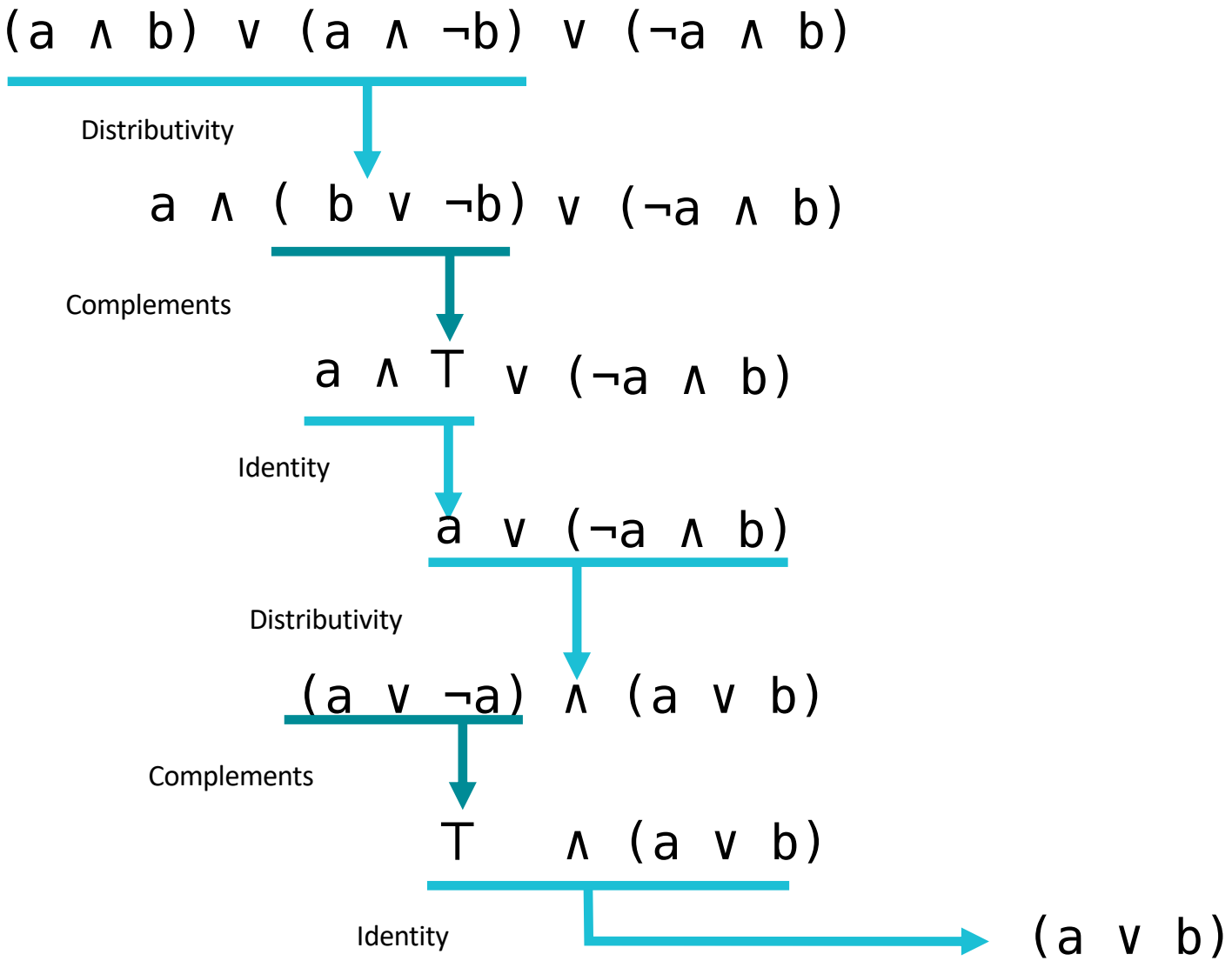
Boolean Algebra Laws

Identity	$a \vee \bot = a$ $a \wedge \top = a$
Annihilator	$a \vee \top = \top$ $a \wedge \bot = \bot$
Absorption	$a \wedge (a \vee y) = a$ $a \vee (a \wedge y) = a$
Associativity Law	$a \vee (x \vee y) = (a \vee x) \vee y$ $a \wedge (x \wedge y) = (a \wedge x) \wedge y$
Distributivity Law	$a \vee (x \wedge y) = (a \vee x) \wedge (a \vee y)$ $a \wedge (x \vee y) = (a \wedge x) \vee (a \wedge y)$
De Morgan's Laws	$\neg(x \vee y) = \neg x \wedge \neg y$ $\neg(x \wedge y) = \neg x \vee \neg y$

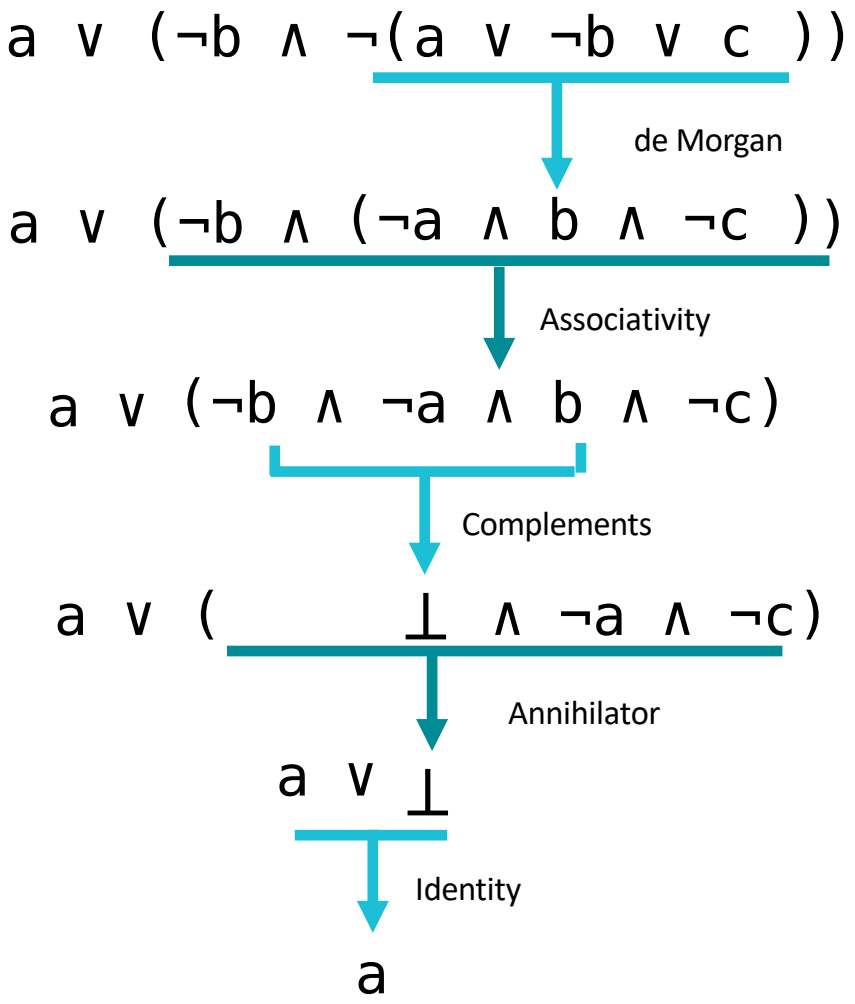
Reductions using Boolean Algebra

- Examples:
  - $(a \wedge b) \vee (a \wedge \neg b) \vee (\neg a \wedge b)$
  - $a \vee (\neg b \wedge \neg(a \vee \neg b \vee c))$

Reductions


(a ∧ b) ∨ (a ∧ ¬b) ∨ (¬a ∧ b)
Distributivity
a ∧ (b ∨ ¬b) ∨ (¬a ∧ b)
Complements
a ∧ T ∨ (¬a ∧ b)
Identity
a ∨ (¬a ∧ b)
Distributivity
(a ∨ ¬a) ∧ (a ∨ b)
Complements
T ∧ (a ∨ b)
Identity
(a ∨ b)

Reductions


a ∨ (¬b ∧ ¬(a ∨ ¬b ∨ c))
de Morgan
a ∨ (¬b ∧ (¬a ∧ b ∧ ¬c))
Associativity
a ∨ (¬b ∧ ¬a ∧ b ∧ ¬c)
Complements
a ∨ (⊥ ∧ ¬a ∧ ¬c)
Annihilator
a ∨ ⊥
Identity
a

Propositional Logic

A more formal introduction

Classical Propositional Logic

- Propositional logic is a two-valued logic
  - A proposition is a propositional sentence that can be either true or false.
- Propositions can be put together using operators in order to form arbitrary logical sentences

Propositional Logic – Syntax

- **Constants**: true (⊤, 1), false (⊥, 0)
- **Symbol**s (atoms, variables,propositions): `A, b, raining, wet, …`
  - **Literal**: a symbol or its negation
- **Logical connectivities**:

1. ¬… Negation (NOT)
2. ∧… Conjunction (AND)
3. ∨…Disjunction (OR)
4. →… Implication
5. ↔… Bijunction (or Equivalence)

Negation

A	¬A
⊤	⊥
⊥	⊤

Conjunction

A	B	A∧B
⊤	⊤	⊤
⊤	⊥	⊥
⊥	⊤	⊥
⊥	⊥	⊥

Disjunction

A	B	A∨B
⊤	⊤	⊤
⊤	⊥	⊤
⊥	⊤	⊤
⊥	⊥	⊥

Propositional Logic – Other operators

IMPLICATION

- „If. . . then. . . “
- Only false, if something false can be derived from truth
- $A \rightarrow B \equiv \neg A \vee B$

Implication

A	B	A→B
⊤	⊤	⊤
⊤	⊥	⊥
⊥	⊤	⊤
⊥	⊥	⊤

BIJUNCTION

- True if both operands are true
- “If and only if”
- $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$

Bijunction

A	B	A↔B
⊤	⊤	⊤
⊤	⊥	⊥
⊥	⊤	⊥
⊥	⊥	⊤

Propositional logic – Syntax

- Well formed formula (WFF)
  - Every constant and logical symbol is a formula
  - If F is a formula, then ¬F is a formula
  - If F and F’ are both formulae, then (F $\wedge$ F'), (F $\vee$ F '),(F $\rightarrow$ F’) and (F $\leftrightarrow$ F ‘) are formulae as well.

It rains.	`rains`
If rains, it is not snowing.	`rains → ¬snowing`
John passes the exam, if and only if he learns enough.	`john_passes ↔ john_learns_enough`

Propositional Logic – Semantics

- A propositional logic formula obtains its truth value from its propositions and involved logical operators
- **Interpretation** I:
  - Is a mapping of symbols to {⊤,⊥}
  - Each symbol in a formula F is assigned true or false.
- **Model**: An interpretation that evaluates to true is called a model.
- The semantics of operators is a function of truth values and can be determined using truth tables.

	A	B	(A∧ ¬B) ∨ A
Interpretation 1	⊤	⊤	⊤
Interpretation 2	⊤	⊥	⊤
Interpretation 3	⊥	⊤	⊥
Interpretation 4	⊥	⊥	⊥

Interpretation of PL formulas

- **Definition** (Interpretation): Given a propositional formula G with atoms $A_1,..,A_k$ occurring in G. An interpretation of G is an assignment of truth values to $A_1,..,A_k$ in which every A_i is assigned either **T** or **F**, but not both.
- **Definition**: A formula G is said to be true under an interpretation iff G is evaluated to **T** in the interpretation. Otherwise, G is said to be false under the interpretation.
- **Definition** (Model): Given a formula G and an interpretation I. If G is true under I, then I is called a model (for G).

Propositional Logic – Semantics

- A formula F is:

SATISFIABLE ➤ If there is at least one interpretation that makes F true

A
⊤
⊥

CONTRADICTORY ➤ **Contradiction**

➤ There is no interpretation that makes F true

A	¬A	A∧ ¬A
⊤	⊥	⊥
⊥	⊤	⊥

VALID ➤ **Tautology**

➤ If F is true in all its interpretations

A	¬A	A∨ ¬A
⊤	⊥	⊤
⊥	⊤	⊤

Checking validity and consistency of PL formulas

- Use truth tables!
- **Definition** (valid): A formula G is said to be valid iff it is true under all its interpretations.
- **Definition** (consistent, satisfiable): A formula G is said to be consistent (satisfiable) iff there is at least one interpretation under which G is true.

Some remarks

- A valid formula G is said to be a TAUTOLOGY, i.e., $\models G$.
- An inconsistent formula G is said to be a CONTRADICTION, i.e., $\neg \models G$.
- A valid formula is consistent but not vice versa.
- An inconsistent formula is invalid but not vice versa.

Logical consequences

- **Definition** (Consequence): Given a formula F and G. G is said to be a logical consequence of F iff for any interpretation I in which F is true, G is also true. We write in this case $F \models G$.
- *Remark*: If $F \models G$, then every model for F is also a model for G.

Propositional Logic – Semantics

- Two formulae F and F' are:

EQUIVALENT

- If F and F‘ have the same interpretations
- A ≡¬(¬A) or A ⧢ ¬(¬A)
- Definition of equivalence:
  a ⧢ b if and only if a ⊨ b and b ⊨ a

A	¬(¬A)
⊤	⊤
⊥	⊥

Special structures

- **Literal**: symbol or its negation
- **Clause**: disjunction of literals — **A∨¬B∨C**
- **Cube**: conjunction of literals — **A∧¬B∧C**
- **Horn clause**: clause with at most one non-negated literal — **¬A∨¬B∨C**
- Conjunctive Normalform (**CNF**)
  - Conjunction of disjunctions — **(A∨¬B∨C)∧(D∨C)**
  - Conjunction of clauses
- Disjunctive Normalform (**DNF**)
  - Disjunction von conjunctions — **(¬B∧C)∨(A∧C)∨(A∧D∧B)**
  - Disjunction of cubes

Conversion to CNF

1. Elimination of $\leftrightarrow$ using $A \leftrightarrow B \equiv (A \rightarrow B) \wedge (B \rightarrow A)$
2. Elimination of $\rightarrow$ using $A \rightarrow B \equiv (\neg A \vee B)$
3. Move negations $\neg$ to basic propositions/atoms using de Morgan‘s laws and $\neg\neg A \equiv A$

   $\neg(x \vee y) = \neg x \wedge \neg y$

   $\neg(x \wedge y) = \neg x \vee \neg y$

4. Move disjunctions $\vee$ close to literals using the distributivity law

   $a \vee (x \wedge y) = (a \vee x) \wedge (a \vee y)$

Conversion to CNF

1. ((a ∨ b)∧ c) ∨ (a∧ b)
2. ¬(a ∧ (b→c))∨ (a→¬b)

Construct a CNF from an arbitray logic function

1. Construct the truth table of the function.
2. Choose all configuration where the formula evaluates to false.
3. Negate all basic proposition.
4. Put together all basic propositions of a configuration using the OR operator.
5. Put together all configurations using the AND operator.

Satisfiability Problem (SAT Problem)

- Can we **assign the truth values to all propositions in a formula** such that the **formula evaluates to true** (or is satisfiable)?
- **SAT** is a **NP complete problem** and therefore difficult to solve
  - We need an **exponential number of checks**!
  - **Exception**: **Horn clause** formulae where SAT can be solved in **polynomial** time
- SAT can be solved using either:
  - **Trying** all different **interpretations**
  - Use the **resolution calculus**