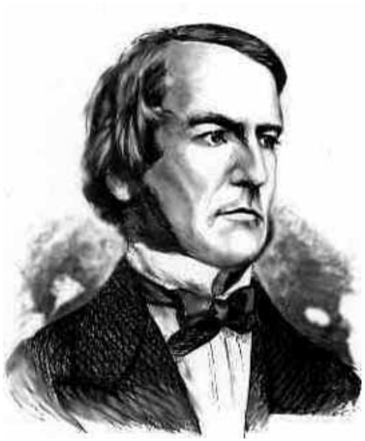
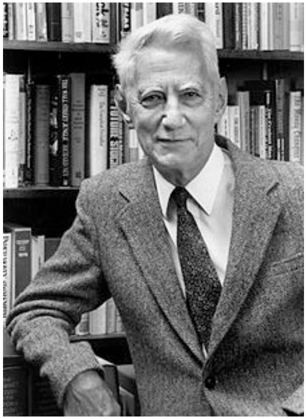
Part I:

Computer arithmetic and logic


George Boole

(1815–1864)


Claude Shannon

(1916–2001)

Why are computers binary?

Computers are electronic devices whose `language' is based on electrical signals; the simplest signals are **on** and **off**. So the **alphabet** of this language has only two symbols **1** and **0** known as **binary digits** or **bits**.

Computer words consist of a fixed number of bits, which depends on the processor design and computer architecture, usually 64 or 32.

Example: 32-bit word

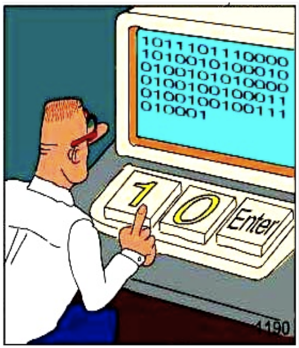
$$\underbrace{11000000}_{byte}\ \underbrace{01010000}_{byte}\ \underbrace{10001010}_{byte}\ \underbrace{00000011}_{byte}$$

Binary words can encode and store numbers of different type, computer instructions, texts in natural languages, etc. We start with the numbers.

George Boole (in `The Laws of Thought' 1854) invented Boolean algebra, an algebra of two values, which is the basis for all modern computer arithmetic.

Claude Shannon (in his MSc thesis) designed electrical switching circuits and showed how they could solve all problems that Boolean algebra could solve. This has become a fundamental concept that underlies all electronic digital computers.

Numbers



are abstract mathematical objects used for counting and measuring. There are several types of numbers:

- $\mathbb{N}$ Natural numbers $(0, 1, 2, 3, 4, 5, \ldots$ *ad infinitum*)
- $\mathbb{Z}$ Integer numbers $(\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots)$
- $\mathbb{Q}$ Rational numbers $(-\frac{7}{3}, \frac{4}{9}$, etc.;
  formally, they are of the form $\frac{m}{n}$, where m and n are integers and $n \neq 0$)
  see https://en.wikipedia.org/wiki/Rational_number
- $\mathbb{R}$ Real numbers (like $\pi = 3.14159265358\ldots$, $\sqrt{2} = 1.41421356237\ldots$, etc.)
  see https://en.wikipedia.org/wiki/Irrational_number
- . . .

How can we represent all of these numbers in computers?

- Computer words are **binary** (no problem: binary can encode everything)
- Computer words are **finite**, usually of 32 or 64 bits
  (can lead—actually has already led—to disaster)

Numeral systems

A **numeral** is a symbol or group of symbols that represents a number

- **Unary:** numeral ||||||| means 7 – Roman numerals
- **Decimal:** numeral 456 means $(4 \times 10^2) + (5 \times 10^1) + (6 \times 10^0)$

  numeral 101_{10} means $(1 \times 10^2) + (0 \times 10^1) + (1 \times 10^0)$
- **Binary:** numeral 101_2 means $(1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$,

  i.e., decimal 5

A 32-bit binary word $a_{31}a_{30}\ldots a_1a_0$ means the number

$$a_{31} \times 2^{31} + a_{30} \times 2^{30} + \cdots + a_1 \times 2^1 + a_0 \times 2^0$$

From now on, we assume that computer words are **binary** and **32** bits long.

This means that we can represent the natural numbers from 0 to $2^{32} - 1$:

$$\begin{aligned}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 &= 0_{10} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 &= 1_{10} \\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 &= 2_{10} \\
\ldots \qquad & \qquad \ldots \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 &= 2^{32} - 2 = 4,294,967,294_{10} \\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 &= 2^{32} - 1 = 4,294,967,295_{10}
\end{aligned}$$

Converting decimal numbers to binaries: divide by 2

Rule: divide repeatedly by 2, keeping track of the remainders as you go

Example: covert 123_{10} to binary

$123/2 = 61$	remainder $= 1$	$(123 = 61 \times 2 + 1)$
$61/2 = 30$	remainder $= 1$	
$30/2 = 15$	remainder $= 0$	
$15/2 = 7$	remainder $= 1$	
$7/2 = 3$	remainder $= 1$	
$3/2 = 1$	remainder $= 1$	
$1/2 = 0$	remainder $= 1$	

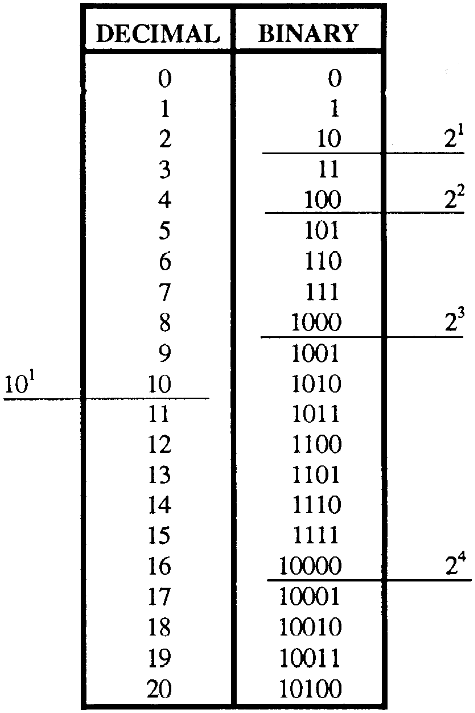
The result is read from the last remainder **upwards**:

$$\boxed{123_{10} = 1111011_2}$$

why does it work?

‘Important’ binary numbers: powers of 2

‘Important’ decimal numbers are $10 = 10^1$, $100 = 10^2$, $1000 = 10^3$, $\ldots$



	DECIMAL	BINARY	
	0	0	
	1	1	
	2	10	2^1
	3	11	
	4	100	2^2
	5	101	
	6	110	
	7	111	
	8	1000	2^3
	9	1001	
10^1	10	1010	
	11	1011	
	12	1100	
	13	1101	
	14	1110	
	15	1111	
	16	10000	2^4
	17	10001	
	18	10010	
	19	10011	
	20	10100	

Binary addition and subtraction (for unsigned numbers)

- The four basic rules for adding binary digits are as follows:

$0 + 0 = 0$	sum of 0 with a carry of 0
$0 + 1 = 1$	sum of 1 with a carry of 0
$1 + 0 = 1$	sum of 1 with a carry of 0
$1 + 1 = 10$	sum of 0 with a carry of 1

Example:

$$\begin{array}{ccccc} & \text{Carry} & \text{Carry} & \text{Carry} & \\ & 1 & 1 & 1 & \\ & & 0 & 1 & 1 \\ + & & 1 & 0 & 1 \\ \hline & 1 & 0 & 0 & 0 \end{array}$$

- Formulate four rules for subtracting bits $(0 - 1 = 1$ with a borrow of 1)
- What about multiplication? (multiplication in binary is exactly as it is in decimal)

Negative numbers: sign-magnitude representation

How to represent negative integer numbers?

Obvious idea: treat the most significant (left-most) bit in the word as a **sign**:
if it is 0, the number is **positive**; if the left-most bit is 1, the number is **negative**;
the remaining 31 bits contain the **magnitude** of the integer

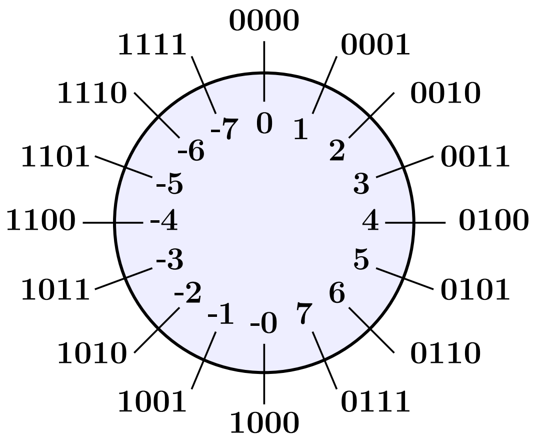
Example:

$$
\begin{aligned}
+18_{10} &= 0000\;0000\;0000\;0000\;0000\;0000\;000\overbrace{1\;0010}^{18}{}_2 \\
-18_{10} &= 1000\;0000\;0000\;0000\;0000\;0000\;0001\;0010_2
\end{aligned}
$$

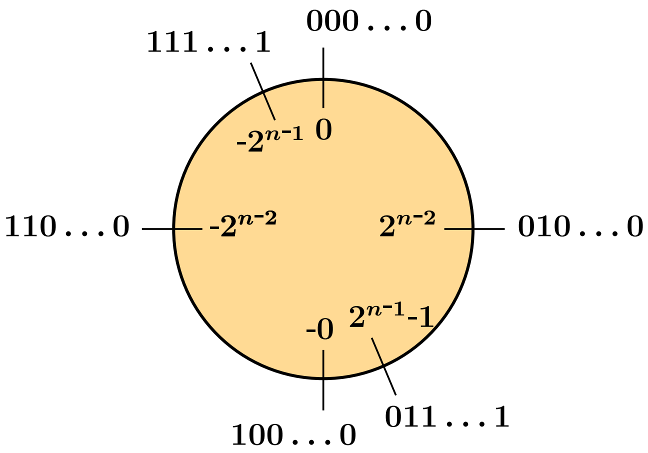
This representation was used in early machines, but several shortcomings have been revealed

- Awkward arithmetic: $18 + (-18) = 0$
- Two zeros: $+0$ and -0
- ...

Sign-magnitude representation



4-bit numbers



n-bit numbers

Ones' complement representation: invert the bits

$$+18_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001\ 0010_2$$

$$-18_{10} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110\ 1101_2$$ (invert the bits of $+18$)

- still two zeros: $+0$ and -0 (used in some computers until the late 1980s)
- still a little bit awkward arithmetic: compute $3 + (-1)$ as an exercise

Two's complement representation

- The most significant bit represents the **sign**, as before
- The positive numbers are also represented as before

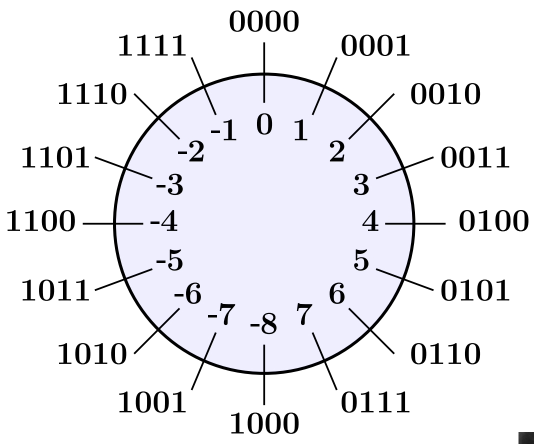
$0a_{30}\dots a_0$ **means** $0\times 2^{31} + a_{30}\times 2^{30} + \cdots + a_1 \times 2 + a_0$

- To represent a negative number, take its **complement** to 2^{31}: more precisely,

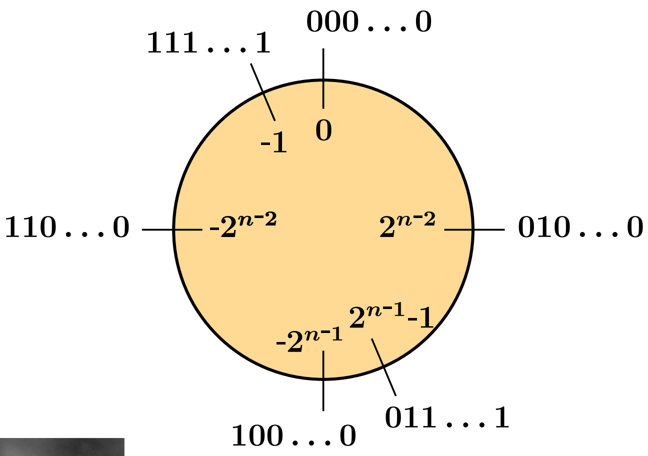
$1a_{30}\dots a_0$ **means** $-1\times 2^{31} + a_{30}\times 2^{30} + \cdots + a_1 \times 2 + a_0$

$$
\begin{array}{lcl}
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 & = & 0_{10}\\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 & = & 1_{10}\\
0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 & = & 2_{10}\\
\dots & & \dots\\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 & = & 2,147,483,645_{10}\\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 & = & 2,147,483,646_{10}\\
0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 & = & 2,147,483,647_{10}\\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_2 & = & -2,147,483,648_{10}\\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 & = & -2,147,483,647_{10}\\
1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2 & = & -2,147,483,646_{10}\\
\dots & & \dots\\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 & = & -3_{10}\\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2 & = & -2_{10}\\
1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_2 & = & -1_{10}
\end{array}
$$

Two's complement representation (cont.)



4-bit numbers



n-bit numbers

John von Neumann suggested use of two's complements in computers in the 1940s

Two's complement arithmetic: negation

$a_{31}a_{30}\dots a_1a_0$ **represents** $-a_{31}\times 2^{31}+a_{30}\times 2^{30}+\dots+a_1\times 2+a_0$

Rules for forming negation $-N$ of an integer N (in two's complement notation)

- Take the Boolean negation of each bit of the integer (**including the sign bit**)
  That is, set each 1 to 0 and each 0 to 1.
- Treating the result as an unsigned binary integer, add 1.

Example 1: Negate $2_{10} = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2$

Invert the bits and add 1:

$$\begin{array}{rl} & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_2 \\ + & 1_2 \\ \hline & 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2\ (=-2_{10}) \end{array}$$

Example 2: Negate $-2_{10} = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_2$

Invert the bits and add 1:

$$\begin{array}{rl} & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_2 \\ + & 1_2 \\ \hline & 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_2\ (=2_{10}) \end{array}$$

- Negate 0. Any problem? Overflow; but is the result correct?
- Negate $10000000000000000000000000000000_2 = -2,147,483,648_{10}$. Any problem? Why?

Two's complement addition and subtraction

- **Subtraction** is achieved using addition $x - y = x + (-y)$:
  to subtract y from x, negate y and add the result to x
- **Addition** is the same as the addition of unsigned numbers on page 7

The result of addition can be **larger** than can be held in a 32 bit word.
This situation is called **overflow**.

- Detecting overflow: if two numbers are added, and they are both positive or both negative, then overflow occurs if the result has the opposite sign

Examples of overflow: assume that we deal with 4 **bit** words only

$$\begin{array}{rcl} & 0101_2 & = 5_{10} \\ + & 0100_2 & = 4_{10} \\ \hline & 1001_2 & = -7_{10} \end{array} \quad \text{overflow}$$

$$\begin{array}{rcl} & 1001_2 & = -7_{10} \\ + & 1010_2 & = -6_{10} \\ \hline \boxed{1} & 0011_2 & = 3_{10} \end{array} \quad \text{overflow}$$

When adding numbers with different signs, overflow cannot occur. Why?

Scientific notation

The two's complement representation allowed us to deal with the integer numbers in the interval between -2^{31} and $+2^{31} - 1$

- What if we need larger numbers (say, in astronomy)?
- How to deal with small fractions (say, in nuclear physics)?

'Scientific' notation: $976,000,000,000,000 = 9.76 \times 10^{14}$

in the UK: **standard form** $0.00000000000000976 = 9.76 \times 10^{-14}$

We dynamically slide the decimal point to a convenient location and use the exponent of 10 to keep track of that decimal point

Any given number can be written in the form $a \times 10^{b}$ in many ways; e.g.,

$$350 = 3.5 \times 10^{2} = 35 \times 10^{1} = 350 \times 10^{0}$$

In **normalised scientific notation**, exponent b is chosen such that $1 \leq |a| < 10$

Using binary numeral system instead of decimal, we can represent any real **non-zero** number in the form $a \times 2^{E}$ with $1 \leq |a| < 2$, or

$$\boxed{(-1)^{S} \times (1 + F) \times 2^{E}, \qquad 0 \leq F < 1}$$

$S = 0, 1$ is the sign

Binary fractions

A binary number $a_n a_{n-1} \ldots a_1 a_0 \,.\, a_{-1} \ldots a_{-m}$ has the following meaning:

$$a_n \times 2^n + a_{n-1} \times 2^{n-1} + \cdots + a_1 \times 2^1 + a_0 \times 2^0 + a_{-1} \times 2^{-1} + \cdots + a_{-m} \times 2^{-m}$$

$$\begin{aligned} 16.625_{10} &= 1 \times 10^1 + 6 \times 10^0 + 6 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3} \\ &= 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 + 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \end{aligned}$$

$= 10000.101_2$ $\quad$ $0.1_2 = 1/2 = 0.5_{10}$ $\quad$ $0.01_2 = 1/4 = 0.25_{10}$ $\quad$ $0.001_2 = 1/8 = 0.125_{10}$

To convert a decimal number with both integer and fractional parts, convert each part separately and combine the answers (integer conversion is on page 5)

Example: convert 0.6875_{10} to binary

Rule: **multiply by 2** keeping track of the resulting integer and fractional part

$0.6875 \times 2 = 1.3750$ integer $= 1$

$0.3750 \times 2 = 0.7500$ integer $= 0$

$0.7500 \times 2 = 1.5000$ integer $= 1$

$0.5000 \times 2 = 1.0000$ integer $= 1$

The result is read from the first integer **downwards**:

$$0.6875_{10} = 0.1011_2$$

IEEE 754 floating-point standard

Every real number different from 0 can be represented in the form

$$\boxed{(-1)^S \times (1 + F) \times 2^E, \qquad 0 \le F < 1}$$

- S is the **sign** (0 for positive and 1 for negative)
- F is the **fraction** ($0 \le F < 1$)
- E is the **exponent** (determining the actual location of the binary point)

The size of E and F may vary: a fixed word size means that one must take a bit from one to add a bit to the other. There existing compromises are:

Single precision floating-point format: 8 bits for E and 23 bits for F

$$\underbrace{\boxed{S}}_{1\text{ bit}}\underbrace{\boxed{\quad\text{Exponent}_b\quad}}_{8\text{ bits}}\underbrace{\boxed{\qquad F \qquad \text{(fraction)}\qquad}}_{23\text{ bits}}$$

Double precision floating-point format: 11 bits for E and 52 bits for F

$$\underbrace{\boxed{S}}_{1\text{ bit}}\underbrace{\boxed{\quad\text{Exponent}_b\quad}}_{11\text{ bits}}\underbrace{\boxed{\qquad F \qquad \text{(fraction)}\qquad}}_{52\text{ bits}}$$

IEEE 754 floating-point standard (cont.)

If we number the bits of the fraction F from left to right $b_1, b_2, \ldots, b_n$ (where $n = 23$ or $n = 52$), then the IEEE 754 standard gives us the number

$$\boxed{(-1)^S \times (1 + b_1 \times 2^{-1} + b_2 \times 2^{-2} + b_3 \times 2^{-3} + \cdots + b_n \times 2^{-n}) \times 2^E}$$

- How to represent E? We need both **positive** and **negative** exponents.

Biased notation: fix $Bias = 127_{10} = 2^7 - 1 = 0111\ 1111_2$ for single precision
$Bias = 1023_{10} = 2^{10} - 1 = 011\ 1111\ 1111_2$ for double precision

$$\boxed{Exponent_b \ = \ E + Bias} \qquad \boxed{E \ = \ Exponent_b - Bias}$$

where $Exponent_b$, **biased exponent**, is the number **stored in the exponent field**

Examples:

- An exponent of -1 is represented by the bit pattern of the value
$-1 + 127_{10} = 0111\ 1110_2$ (single precision)
- An exponent of $+1$ is represented by the bit pattern of the value
$1 + 127_{10} = 1000\ 0000_2$ (single precision)

Biased representation

Biased representation (aka offset binary representation) is a coding scheme where all-0 is the minimal negative value and all-1 the maximal positive value

decimal number (E)	biased representation $(Exponent_b = E + 127)$
-127	$0000\,0000$
-126	$0000\,0001$
$\dots$	$\dots$
-1	$0111\,1110$
0	$0111\,1111$
$+1$	$1000\,0000$
$\dots$	$\dots$
$+128$	$1111\,1111$

NB. There are some special numbers (say all-0 and all-1 exponents)

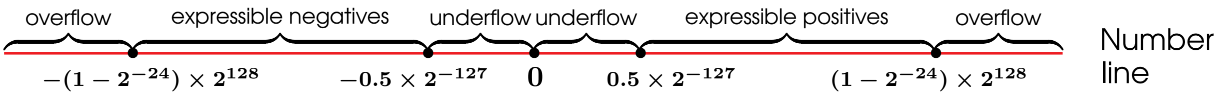
IEEE 754 floating-point standard (cont.)

The range of single precision numbers is then from as small as

$$\boxed{\pm 1.0000\ 0000\ 0000\ 0000\ 0000\ 000_2 \times 2^{-126}}$$

to as large as

$$\boxed{\pm 1.1111\ 1111\ 1111\ 1111\ 1111\ 111_2 \times 2^{127}}$$



- 0 is represented as both $0000\ldots0000$ ($+0$) and $1000\ldots0000$ (-0)
- not all numbers in the intervals between $-(1-2^{-24}) \times 2^{128}$ and -0.5×2^{-127} and between 0.5×2^{-127} and $(1-2^{-24}) \times 2^{128}$ are represented (why?)

Example: floating point representation

Show the IEEE 754 binary representation of the number -0.75_{10} in single and double precision

$$-0.75_{10} = -(3/4)_{10} = -(3/2^2)_{10} = -11_2/2^2_{10} = -0.11_2 =$$
$$\boxed{= -0.11_2 \times 2^0 = -1.1_2 \times 2^{-1}}$$

The general representation is

$$\boxed{(-1)^S \times (1 + \mathit{Fraction}) \times 2^E}$$

In our case: $S = -1$, $\mathit{Fraction} = 0.1$, $E = -1$, $\mathit{Exponent}_b = -1 + 127 = 126$

The single precision binary representation of -0.75_{10} is therefore

$$\underbrace{1}_{1\text{ bit}}\ \underbrace{01111110}_{8\text{ bits}}\ \underbrace{10000000000000000000000}_{23\text{ bits}}$$

- What is the double precision representation?

Example: converting binary to decimal floating point

What decimal number is represented?

$$\underbrace{1}_{1 \text{ bit}} \; \underbrace{10000001}_{8 \text{ bits}} \; \underbrace{01000000000000000000000}_{23 \text{ bits}}$$

The sign bit is 1, the exponent field contains $10000001_2 = 129_{10} = \mathit{Exponent}_b$, and the fraction field contains $0.01_2 = 1 \times 2^{-2} = \frac{1}{4} = 0.25$

$$(-1)^S \times (1 + \mathit{Fraction}) \times 2^{\mathit{Exponent}_b - \mathit{Bias}} = (-1)^1 \times (1 + 0.25) \times 2^{129-127} =$$
$$= -1 \times 1.25 \times 2^2 = -1.25 \times 4 = \boxed{-5}$$

How do computers add, subtract etc.?

Logic

Logic is the formal systematic study of the principles of **valid inference** and **correct reasoning**

Are the following inferences (or arguments) valid?

- If it is raining then I take an umbrella
- It is raining ✓
- Therefore I take an umbrella

- If it is raining then I take an umbrella
- It is not raining ✗
- Therefore I don't take an umbrella

What does it mean for one sentence to **follow logically** from certain others?

The sentences above are **declarative sentences**, or **propositions**, which one can, in principle, argue as being **true** or **false**

Boolean algebra (or **Boolean logic**) is a logical calculus of truth values, developed by George Boole in the 1840s

Elements of Boolean logic

Basic assumption: every **proposition** can either be **true** or **false** (but not both)

Examples:

(A) Donald Trump is the current president of the United States of America.

(B) Joe Biden is the current president of the United States of America.

(C) Extraterrestrial life does not exist (even though we **don't know** whether it's true or false)

NB: Questions/exclamations, paradoxical statements like 'this proposition is false' are **not** propositions.

Propositional connectives:

- **not** (negation) denoted by $\neg$ (! in C++/Java) Is $\neg A$ true?
- **and** (conjunction) denoted by $\wedge$ (&& in C++/Java) Is $A \wedge B$ true?
- **or** (disjunction) denoted by $\vee$ (|| in C++/Java) Is $A \vee B$ true?
- **if ... then ...** (implication) denoted by $\to$ Is $A \to C$ true?

? Are there any other propositional connectives?

Complex propositions (formulas): $(\neg A) \to (B \vee C)$, $((\neg B) \wedge (\neg\neg C)) \to \neg A$, etc.

Semantics: truth-tables

Notation: 1 for 'true', 0 for 'false'

$A, B, C, A_1, B_1, \ldots$ for atomic (in a given context) propositions
a.k.a. **propositional variables**

$A \vee B,\ (\neg A) \to (A_1 \wedge \neg B_2), \ldots$ for complex propositions
a.k.a. **propositional** or **Boolean formulas**

Truth-tables for $\wedge$, $\vee$, $\to$ and $\neg$:

A	B	$A \wedge B$	$A \vee B$	$A \to B$	$\neg A$
0	0	0	0	1	1
0	1	0	1	1	1
1	0	0	1	0	0
1	1	1	1	1	0

so the proposition 'if the Moon is made of green cheese, then $2 \times 2 = 7$' is true

Explaining ‘implication’

One possible ‘explanation’ of the truth-table for the implication $\to$ is as follows:

The following statement is known to be true for **every** natural number n:

If n is divisible by 4, then n is divisible by 2.

So the following instances of this general statement must be true as well:

If 8 is divisible by 4, then 8 is divisible by 2	$(1 \to 1) = 1$
If 7 is divisible by 4, then 7 is divisible by 2	$(0 \to 0) = 1$
If 2 is divisible by 4, then 2 is divisible by 2	$(0 \to 1) = 1$

And of course, ‘if 8 is divisible by 4, then 7 is divisible by 2’ is false $(1 \to 0) = 0$

(also: analyse the wrong inference on page 23)

This interpretation of logical connectives is a **mathematical abstraction**. Under such abstractions, meaningless sentences may become sensible, and the other way round.

There are different interpretations of logic connectives, e.g., with three or more truth-values.

Truth-tables for Boolean formulas

Exercise: construct the truth-table for the formula $\boxed{(\neg\, A) \lor B}$
by first computing the truth-values for $\neg A$ and then for $(\neg\, A) \lor B$

A	B	$\neg$	A	$\lor$	B
0	0	1	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	1	0	1	1	1

Note that this truth-table (the column under the main connective $\lor$)
is the same as the truth-table for $A \to B$

So we can say that $(\neg A) \lor B$ is **equivalent to** $A \to B$

Exercise Brown, Jones and Smith are suspected of income tax evasion.
They testify under oath as follows:

- Brown: Jones is guilty and Smith is innocent.
- Jones: If Brown is guilty, then so is Smith.
- Smith: I'm innocent, but at least one of the others is guilty.

Assuming everyone's testimony is true, who is innocent and who is guilty?

Solution

BG stands for 'Brown is guilty', JG for 'Jones is guilty' and SG for 'Smith is guilty'

- Brown says: $JG \wedge \neg SG$ Jones says: $BG \rightarrow SG$ Smith says: $\neg SG \wedge (BG \vee JG)$

BG	JG	SG	$JG \wedge \neg SG$	$BG \rightarrow SG$	$\neg SG \wedge (BG \vee JG)$
0	0	0	0	1	0
0	0	1	0	1	0
0	1	0	1	1	1
0	1	1	0	1	0
1	0	0	0	0	1
1	0	1	0	1	0
1	1	0	1	0	1
1	1	1	0	1	0

The only case where all of the statements are true: $BG = 0$, $JG = 1$, $SG = 0$

This problem can be solved in a more direct way. From the first statement we can infer that Jones is guilty and Smith is innocent. From the second statement, it follows that if Smith is not guilty then Brown is not guilty. Therefore we can infer that Brown is innocent.

Formalising English sentences

Exercise: Translate the following English sentences to propositional logic:

(1) *If I am not playing tennis, I am watching tennis.*

(2) *If I am not watching tennis, I am reading about tennis.*

(3) *I cannot do more than one of these activities at the same time.*

Solution: First we choose our propositional variables.

(NB: they must denote propositions! say T: `*playing tennis*' is no good)

T: `*I am playing tennis*'

W: `*I am watching tennis*'

R: `*I am reading about tennis*'

Then we can formalise the above English sentences as follows:

(1) $\neg T \rightarrow W$

(2) $\neg W \rightarrow R$

(3) $\neg(T \wedge W) \wedge \neg(T \wedge R) \wedge \neg(W \wedge R)$

or $(T \wedge \neg W \wedge \neg R) \vee (\neg T \wedge W \wedge \neg R) \vee (\neg T \wedge \neg W \wedge R) \vee (\neg T \wedge \neg W \wedge \neg R)$

Logically correct arguments in propositional logic

An **argument** is a sequence of propositions:

$$\left.\begin{array}{l} p_1 \\ p_2 \\ \vdots \\ p_n \end{array}\right\} \quad n \text{ premises (aka assumptions)}$$

Therefore ________

q one **conclusion**

An argument is **logically correct** if

in every 'situation' that makes all the premises true, the conclusion is true as well

a **situation** = a row in the truth table for $p_1, \ldots, p_n, q$

= a possible assignment for the propositional variables in $p_1, \ldots, p_n, q$

$\boxed{\begin{array}{l} p_1 \\ p_2 \\ \ldots \\ p_n \\ \hline q \end{array}}$ is logically correct iff the formula $(p_1 \wedge p_2 \wedge \cdots \wedge p_n) \rightarrow q$ is always true

a formula that is always true is called a **tautology**

Logically correct argument: an example

If I am not playing tennis, I am watching tennis	$\neg T \rightarrow W$	
If I am not watching tennis, I am reading about tennis	$\neg W \rightarrow R$	3 premises
I cannot do more than one of these activities at the same time	$\neg(T \wedge W) \wedge \neg(T \wedge R) \wedge \neg(W \wedge R)$	
Therefore, *I am watching tennis*	W	conclusion

In every situation that makes all the premises true, the conclusion is true as well:

T	W	R	$\neg T \rightarrow W$	$\neg W \rightarrow R$	$\neg(T \wedge W) \wedge \neg(T \wedge R) \wedge \neg(W \wedge R)$	W
1	1	1	1	1	0	1
1	1	0	1	1	0	1
1	0	1	1	1	0	0
1	0	0	1	0	1	0
0	1	1	1	1	0	1
0	1	0	**1**	**1**	**1**	**1**
0	0	1	0	1	1	0
0	0	0	0	0	1	0

premises conclusion

Incorrect argument: a simple example

If you solve every exercise in the textbook, then you will get an A.	$S \to A$
You did not solve every exercise in the textbook.	$\neg S$
Therefore	
You won't get an A.	$\neg A$

Incorrect argument: It is not the case that
'in every situation that makes all the premises true, the conclusion is true as well.'

So, to show that an argument is incorrect, it is **enough to find one situation** where

- all premises are true,
- but the conclusion is false.

S	A	$S \to A$	$\neg S$	$\neg A$
1	1	1	0	0
1	0	0	0	1
0	1	**1**	**1**	**0**
0	0	1	1	1

If such a situation **doesn't exist** the argument is **correct**

Example: The argument **'if A and $\neg A$, then B'** is correct for any A and B

Boolean logic in computers

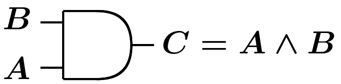
In the world of computers, 0 and 1 are called **bits** (for binary digit)

- 0 is represented by the lower voltage level (LOW), say, 0V – 0.1V
- 1 is represented by the higher voltage level (HIGH), say, 0.9V – 1.1V

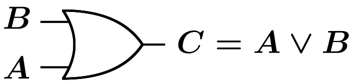
All computer circuits consist of hundreds of millions of interconnected primitive elements called **gates**, which correspond to the basic logic connectives:

Basic logic gates:

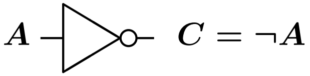
AND gate

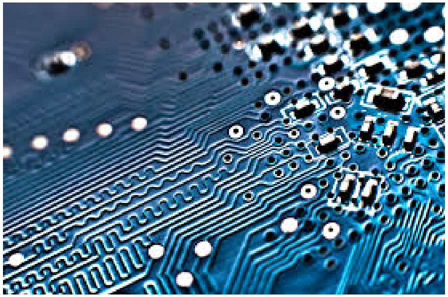


OR gate



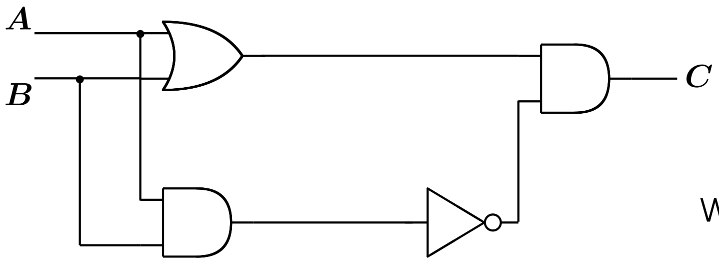
NOT gate





Since the 1990s, logic gates are made of transistors (semiconductor devices used to amplify and switch electric signals)

Example: Boolean circuit



What does this circuit compute?

- Represent the circuit as a **Boolean equation**

$$\boxed{C \;=\; (A \vee B) \wedge \neg(A \wedge B)}$$

- Construct the truth-table

A	B	$($	A	$\vee$	$B)$	$\wedge$	$\neg$	$(A$	$\wedge$	$B)$
0	0		0	0	0	0	1	0	0	0
0	1		0	1	1	1	1	0	0	1
1	0		1	1	0	1	1	1	0	0
1	1		1	1	1	0	0	1	1	1

The circuit, the truth-table and the formula $(A \vee B) \wedge \neg(A \wedge B)$ represent the **Boolean function** known as **exclusive or** and denoted by **XOR**, or $A \oplus B$

Boolean functions of one argument

(intuitive explanation of ‘function’: `https://en.wikipedia.org/wiki/Function_(mathematics)`)

A	0
0	0
1	0

— **constant function** 0 (always returns 0 and doesn’t depend on A)

draw a Boolean circuit for this function

A	1
0	1
1	1

— **constant function** 1 (always returns 1 and doesn’t depend on A)

draw a Boolean circuit for this function

A	A
0	0
1	1

— **identical function** (always returns the input A)

draw a Boolean circuit for this function

A	$\neg A$
0	1
1	0

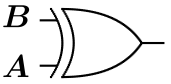
— **function NOT** or $\neg$ (inverts the input A)

draw a Boolean circuit for this function

Boolean functions of two arguments

A	B	$A \wedge B$	$A \vee B$	$A \rightarrow B$	$A \oplus B$	$A \leftrightarrow B$	$A \mid B$	$A \downarrow B$
0	0	0	0	1	0	1	1	1
0	1	0	1	1	1	0	1	0
1	0	0	1	0	1	0	1	0
1	1	1	1	1	0	1	0	0

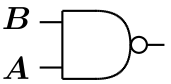
XOR gate



$C = A \oplus B$

$A \leftrightarrow B$ — **equivalence** (if and only if, or iff), equivalent to $(A \rightarrow B) \wedge (B \rightarrow A)$

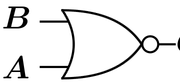
NAND gate



$C = A \mid B \;=\; \neg(A \wedge B)$

Scheffer stroke

NOR gate



$C = A \downarrow B \;=\; \neg(A \vee B)$

Pierce arrow

- What functions are missing here?
- What is the number of Boolean functions of two arguments?

Important questions

There are very many Boolean functions: 2^{2^n} distinct functions of n variables

For example, there are $2^{2^5} = 4,294,967,296$ functions with 5 inputs

We don't know *a priori* which of them are required in computer architecture

- Is it possible to fix some, relatively simple set(s) of Boolean functions (gates), using which one can build **all** other Boolean functions?

We have already seen that the same Boolean functions can be realised in different ways using different gates.

Of course we need smallest possible circuits (formulas). . .

- How to build 'optimal' Boolean circuits (formulas)?
- How to simplify Boolean circuits (formulas)?
- What basic gates to choose?

We consider some aspects of these problems.

Example: the majority function

Suppose we want to realise, using only the AND, OR and NOT gates, the **majority function** $\mu(A, B, C)$ whose output takes the same value as the **majority** of inputs:

A	B	C	$\mu(A, B, C)$
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

Each triple of truth-values for A, B, C on the left-hand side of the table for which $\mu(A, B, C) = 1$ can be represented as conjunctions in the following way:

$0\ \ 1\ \ 1$ is represented by $\neg A \wedge B \wedge C$

it equals 1 if and only if $A = 0,\ B = 1,\ C = 1$

$1\ \ 0\ \ 1$ is represented by $A \wedge \neg B \wedge C$

it equals 1 if and only if $A = 1,\ B = 0,\ C = 1$

etc.

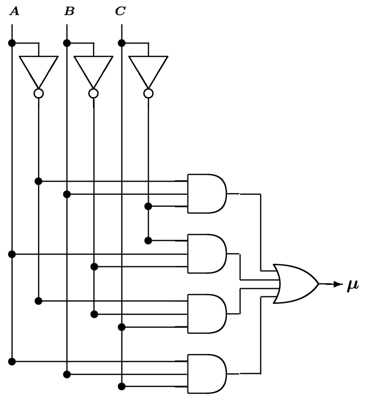
The function $\mu(A, B, C)$ can then be realised as a disjunction of the resulting four conjunctions:

$$\boxed{(\neg A \wedge B \wedge C) \vee (A \wedge \neg B \wedge C) \vee (A \wedge B \wedge \neg C) \vee (A \wedge B \wedge C)}$$

such formulas are said to be in **disjunctive normal form**

Example: the majority function (cont.)

- Use the formula above to construct a Boolean circuit for $\mu(A, B, C)$



- Can you simplify it?

Consider, for instance, the formula

$$\boxed{(A \wedge B) \vee (B \wedge C) \vee (A \wedge C)}$$

- Does it define the same function? (Construct the truth-table)
- Is the corresponding circuit simpler?
- Can you simplify it?

  what about the formula $(A \wedge (B \vee C)) \vee (B \wedge C)$?

Universal sets of Boolean functions

The method shown on page 38 can be used to represent **any Boolean function** by means of a formula with the connectives ¬, ∧ and ∨

if there is no 1 among the function values then this function is $\mathbf{0}$, which can be represented as $\boldsymbol{A} \wedge \neg \boldsymbol{A}$

We say that $\{\neg, \wedge, \vee\}$ is a **universal** set of Boolean connectives/functions or a **functionally complete set**

- Are there other universal sets of Boolean formulas?
- Can simplifications like those on page 39 be done in a systematic way?

Boolean formulas φ, ψ are called **equivalent** if their truth-tables are identical. In this case we write $\varphi \equiv \psi$.

(Greek letters φ, ψ, χ are often used to denote formulas)

As equivalent formulas φ and ψ determine the same Boolean function, we can use either of them to construct Boolean circuits

Useful equivalences

$$\neg(\varphi \land \psi) \equiv \neg\varphi \lor \neg\psi$$
$$\neg(\varphi \lor \psi) \equiv \neg\varphi \land \neg\psi$$
(De Morgan laws)

(φ and ψ are arbitrary Boolean formulas)

$\neg\neg\varphi \equiv \varphi$ (the law of double negation)

$\neg\varphi \lor \varphi \equiv \mathbf{1}$ (the law of the excluded middle, 'to be or not to be')

$\neg\varphi \land \varphi \equiv \mathbf{0}$ (the law of contradiction)

$\varphi \land (\psi \lor \chi) \equiv (\varphi \land \psi) \lor (\varphi \land \chi)$ (distributivity of $\land$ over $\lor$)

$\varphi \lor (\psi \land \chi) \equiv (\varphi \lor \psi) \land (\varphi \lor \chi)$ (distributivity of $\lor$ over $\land$)

$\varphi \land \mathbf{1} \equiv \varphi, \quad \varphi \land \mathbf{0} \equiv \mathbf{0}, \quad \varphi \lor \mathbf{1} \equiv \mathbf{1}, \quad \varphi \lor \mathbf{0} \equiv \varphi, \quad \varphi \land \varphi \equiv \varphi, \quad \varphi \lor \varphi \equiv \varphi$

It follows, for instance, that
$$\varphi \lor \psi \equiv \neg((\neg\varphi) \land (\neg\psi))$$
$$\varphi \land \psi \equiv \neg((\neg\varphi) \lor (\neg\psi))$$

Thus, we can express $\lor$ by means of $\neg$ and $\land$;

likewise, $\land$ can be expressed by means of $\neg$ and $\lor$

So both $\{\neg, \land\}$ and $\{\neg, \lor\}$ are universal (e.g., $\varphi \to \psi \equiv \neg\varphi \lor \psi$)

How to show equivalence: method 1

$$\begin{aligned}
A \lor (A \land B) &\equiv (A \land 1) \lor (A \land B) \\
&\equiv (A \land (B \lor \neg B)) \lor (A \land B) \\
&\equiv (A \land B) \lor (A \land \neg B) \lor (A \land B) \\
&\equiv (A \land B) \lor (A \land B) \lor (A \land \neg B) \\
&\equiv (A \land B) \lor (A \land \neg B) \\
&\equiv A \land (B \lor \neg B) \\
&\equiv A \land 1 \\
&\equiv A
\end{aligned}$$

Thus, $A \lor (A \land B) \equiv A$

How to show equivalence: method 2

Exercise 1: Show that $P \oplus Q \equiv (P \lor Q) \land \neg(P \land Q)$

Solution:

P	Q	$P \oplus Q$
1	1	0
1	0	1
0	1	1
0	0	0

P	Q	$P \lor Q$	$P \land Q$	$\neg(P \land Q)$	$(P \lor Q) \land \neg(P \land Q)$
1	1	1	1	0	0
1	0	1	0	1	1
0	1	1	0	1	1
0	0	0	0	1	0

Exercise 2: Show that $P \leftrightarrow Q \equiv (P \land Q) \lor (\neg P \land \neg Q)$

Solution:

P	Q	$P \leftrightarrow Q$
1	1	1
1	0	0
0	1	0
0	0	1

P	Q	$P \land Q$	$\neg P$	$\neg Q$	$\neg P \land \neg Q$	$(P \land Q) \lor (\neg P \land \neg Q)$
1	1	1	0	0	0	1
1	0	0	0	1	0	0
0	1	0	1	0	0	0
0	0	0	1	1	1	1

Other universal sets of Boolean functions

NAND is universal

To prove this, it is enough to show that, using NAND, we can express both NOT and AND:

- $\neg A \;\equiv\; (A \mid A) \;\equiv\; \neg(A \land A) \;\equiv\; \neg A,$ because $A \land A \;\equiv\; A$
- $A \land B \;\equiv\; (A \mid B) \mid (A \mid B)$ why?

NOR is universal Exercise

$\{1,\ \oplus,\ \land\}$ is universal (recall that $\oplus$ is XOR)

- $\neg A \;\equiv\; (A \oplus 1)$

The SAT problem

A Boolean formula $\varphi(A_1, \ldots, A_n)$ with propositional variables $A_1, \ldots, A_n$ is **satisfiable** if there is an assignment of the truth-values 0 and 1 to the variables $A_1, \ldots, A_n$ under which φ evaluates to 1

Problem: design an algorithm that, given an arbitrary formula $\varphi(A_1, \ldots, A_n)$, returns YES if this formula is satisfiable and NO otherwise

Solution: construct the truth-table for $\varphi(A_1, \ldots, A_n)$ and check whether it contains 1 in the column for φ

How complex is this algorithm? there are 2^n rows in the truth-table

If $n = 1000$, checking every one of the 2^{1000} possible combinations of truth values of the variables cannot be done by a computer in even trillions of years

There is no known algorithm that ***efficiently*** solves SAT (for any given φ). It is generally believed that no such algorithm exists; yet this belief has not been proven mathematically, and resolving the question of whether SAT has a polynomial-time algorithm is equivalent to the **P versus NP** problem, which is a famous open problem in the theory of computing.

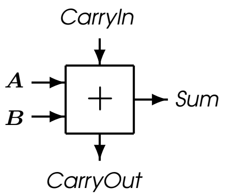
Arithmetic and logic unit (ALU)

The **ALU** is the part of computer that performs arithmetic and logical operations.
It is the workhorse of the CPU because it carries out all the calculations.

The hardware required to build an ALU is the basic gates: AND, OR and NOT
We use these gates to construct a 32-bit adder for integers
Such an adder can be created by connecting 32 1-bit adders

Inputs and outputs of a single-bit adder:

- two inputs for the operands (the bits we want to add), say, $\boldsymbol{A}$ and $\boldsymbol{B}$;
- a single-bit output for the sum;
- a second output to pass on the carry *CarryOut*
- a third input is *CarryIn* — the *CarryOut* from the previous adder



Designing a single-bit adder

We begin by giving precise input and output specifications (truth-table)

A	B	$CarryIn$	$CarryOut$	Sum	Comments
0	0	0	0	0	$0+0+0=00_2$
0	0	1	0	1	$0+0+1=01_2$
0	1	0	0	1	$0+1+0=01_2$
0	1	1	1	0	$0+1+1=10_2$
1	0	0	0	1	$1+0+0=01_2$
1	0	1	1	0	$1+0+1=10_2$
1	1	0	1	0	$1+1+0=10_2$
1	1	1	1	1	$1+1+1=11_2$

Then we turn this truth-table into logical equations:

$$\begin{aligned} CarryOut &= (\neg A \wedge B \wedge CarryIn) \vee (A \wedge \neg B \wedge CarryIn) \vee \\ &\qquad (A \wedge B \wedge \neg CarryIn) \vee (A \wedge B \wedge CarryIn) \end{aligned}$$

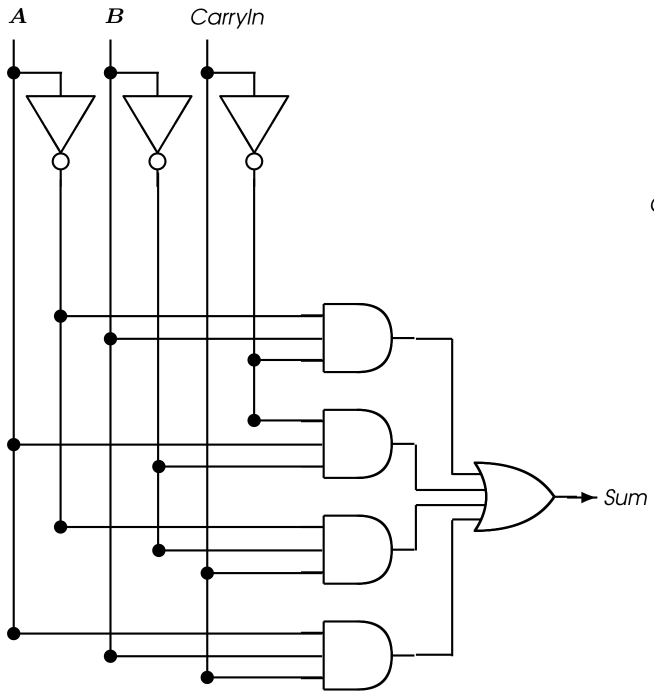
can be simplified to (remember the **majority function**?)

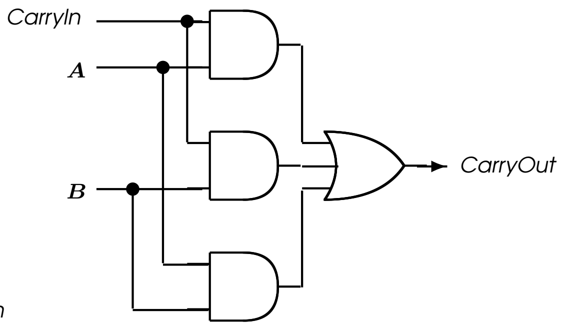
$$CarryOut = (B \wedge CarryIn) \vee (A \wedge CarryIn) \vee (A \wedge B)$$

and

$$\begin{aligned} Sum &= (A \wedge \neg B \wedge \neg CarryIn) \vee (\neg A \wedge B \wedge \neg CarryIn) \vee \\ &\qquad (\neg A \wedge \neg B \wedge CarryIn) \vee (A \wedge B \wedge CarryIn) \end{aligned}$$

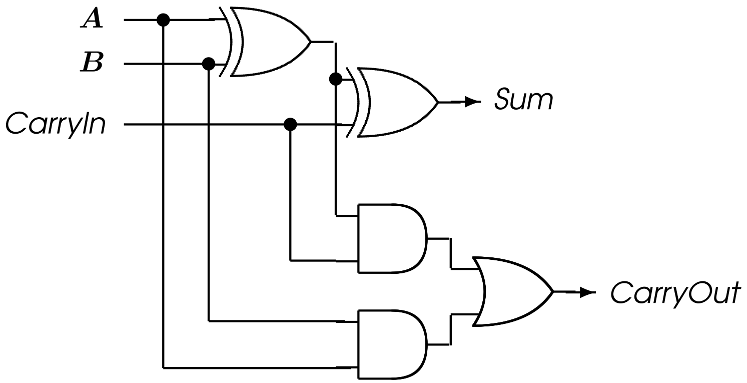
Boolean circuits for the 1-bit adder





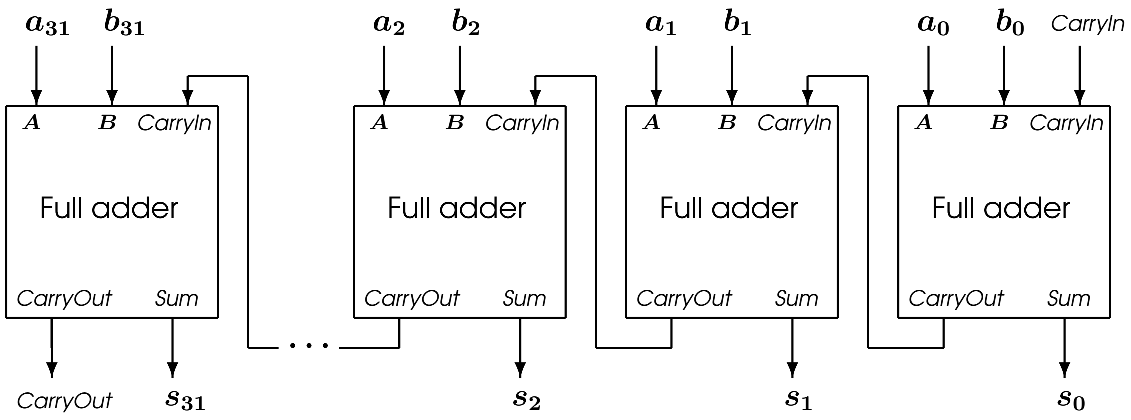
Boolean circuit for the 1-bit adder (cont.)

Check that the following circuit with XOR gates realises the same functions *CarryOut* and *Sum*



32-bit adder

Adding $A = a_{31}a_{30}\ldots a_2a_1a_0$ and $B = b_{31}b_{30}\ldots b_2b_1b_0$

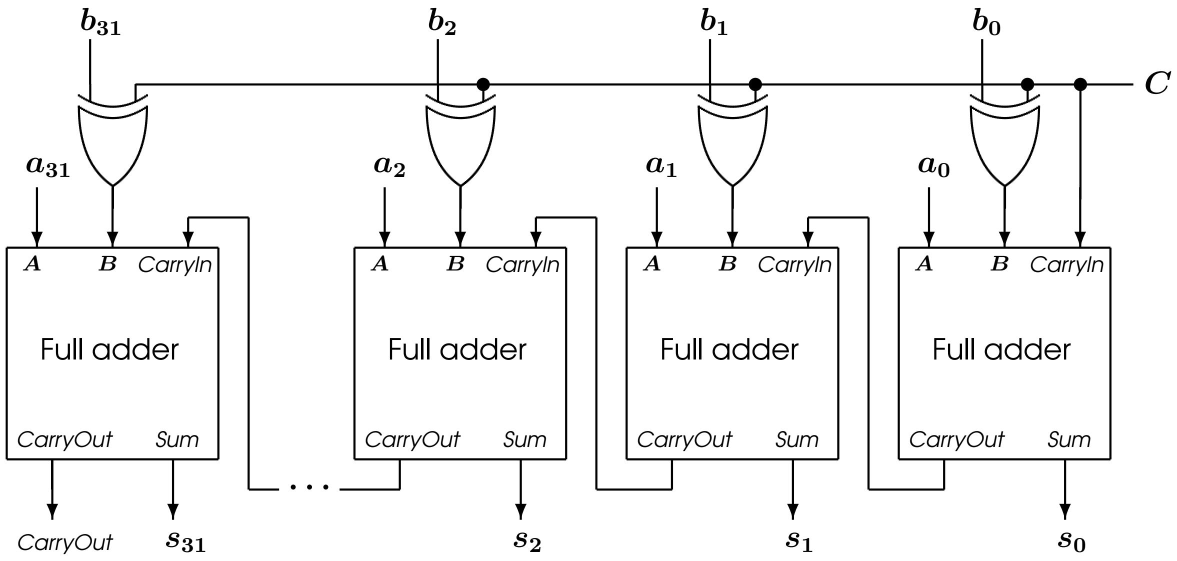


- *CarryIn* in the least significant adder is supposed to be $\mathbf{0}$
- What happens if we set this *CarryIn* to 1 instead of 0?

32-bit adder and subtractor

Control bit C:

- $C = 0$: Add $A + B$ (because $b_i \oplus 0 = b_i$)
- $C = 1$: Subtract $A - B$ (because $b_i \oplus 1 = \neg b_i$)



b_{31}
b_2
b_1
b_0
C
a_{31}
a_2
a_1
a_0
A B CarryIn
Full adder
CarryOut Sum
CarryOut
s_{31}
s_2
s_1
s_0