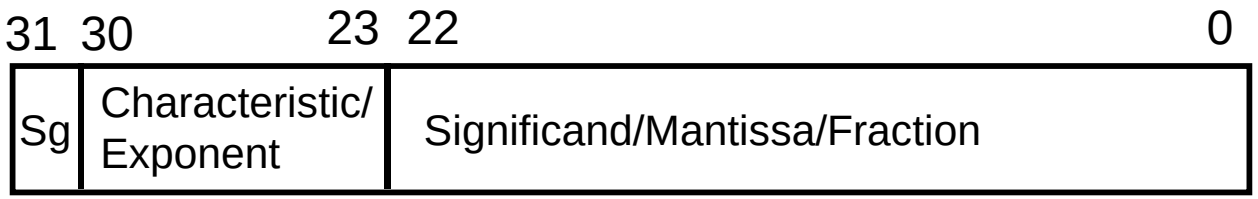


TI II: Computer Architecture
Data Representation and Computer Arithmetic

Systems

Representations

Basic Arithmetic

ALU

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

$$1 + \varepsilon = 1, \varepsilon > 0$$

$$x + y < x, y > 0$$

Content

1. Introduction
   - Single Processor Systems
   - Historical overview
   - Six-level computer architecture

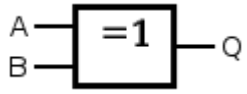
2. Data representation and Computer arithmetic
   - **Data and number representation**
   - **Basic arithmetic**

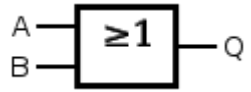
3. Microarchitecture
   - Microprocessor architecture
   - Microprogramming
   - Pipelining

4. Instruction Set Architecture
   - CISC vs. RISC
   - Data types, Addressing, Instructions
   - Assembler

5. Memories
   - Hierarchy, Types
   - Physical & Virtual Memory
   - Segmentation & Paging
   - Caches

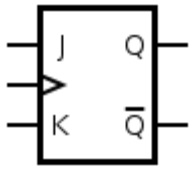
Computer Arithmetic

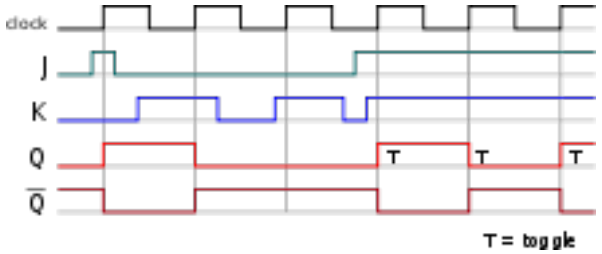




Basics: How to operate on single bits?

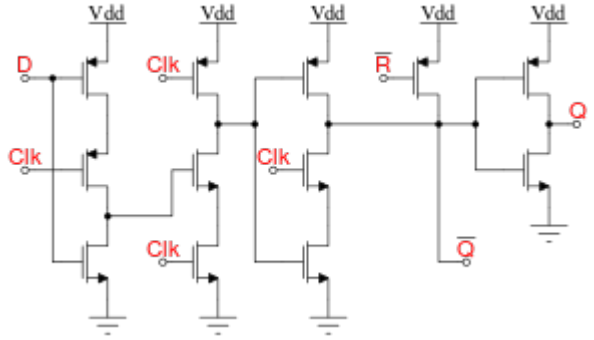
- Combinational (or: combinatorial) circuits (pure logic), sequential circuits (includes memory)
- See lectures on Boolean algebra, circuits, semiconductors etc.





https://en.wikipedia.org/wiki/Logic_gate

https://en.wikipedia.org/wiki/Flip-flop_(electronics)



Here: computer arithmetic serves as an example for handling larger data units (tables, graphics, …)

1. Some more formal basics

2. Methods and circuits for the implementation of the four basic operations +, -, *, /

3. Operation of an ALU (Arithmetic Logic Unit) of a computer

Formal Basics

Humans: typically use the decimal system for calculations (although other systems exist)

Computers: typically use the binary system for calculations

➔ Thus, a conversion is necessary

Additionally, computer systems use other representations such as octal or hexadecimal for the more compact representation of larger binary numbers

➔ Therefore, it is important to understand some mathematical foundations of and relations between the different numbering systems

See also math for CS students!

- Here: only a quick overview

NUMBERING SYSTEMS

Requirements for Number Systems

It should be able to represent positive and negative numbers of a certain interval [-x : y]

It should be able to represent (roughly) the same amount of positive and negative numbers

The representation should be unambiguous

The representation should make calculations simpler – compare “our” decimal system with

- Roman numerals
- Chinese numerals
- Babylonian numerals
- …

Try, e.g., LXIX * XCIX
like you’re used to …

0123456789
𐡹𐡸𐡷𐡶𐡵𐡴𐡳𐡲𐡱𐡰
Ⅰ Ⅱ Ⅲ Ⅳ Ⅴ Ⅵ Ⅶ Ⅷ Ⅸ Ⅹ
০১২৩৪৫৬৭৮৯
๐๑๒๓๔๕๖๗๘๙
൦൧൨൩൪൫൬൭൮൯
〇一二三四五六七八九

https://en.wikipedia.org/wiki/Numeral_system

Number Systems

Most common: positional number systems (https://en.wikipedia.org/wiki/Positional_notation)

Representation of numbers as a sequence of digits z_i, with the radix point between z_0 and z_{-1}:

- $z_n\ z_{n-1}\ \dots\ z_1\ z_0 \bullet z_{-1}\ z_{-2}\ \dots\ z_{-m}$ e.g. 1234.567

Each position i of the sequence of digits is assigned a value, which is a power b^i of the base (or: radix) b of the numbering system

- b-ary numbering system

The value X_b of the number is the sum of all single values of the positions $z_i b^i$:

$$X_b = z_n b^n + z_{n-1} b^{n-1} + \cdots + z_1 b^1 + z_0 b^0 + z_{-1} b^{-1} + \cdots + z_{-m} b^{-m} = \sum_{i=-m}^{n} z_i b^i$$

Number Systems

Common number systems in computer science

Base (b)	Number system	Alphabet
2	Binary system	0,1
8	Octal system	0,1,2,3,4,5,6,7
10	Decimal system	0,1,2,3,4,5,6,7,8,9
16	Hexadecimal system	0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Hexadecimal system: we typically use the letters A to F to represent the digits with values 10 to 15

Binary system: most important system inside a computer

Octal and Hexadecimal systems: very simple to convert into the binary system, easier to read

```
fe80::9c0b:605c:16ba:55c2     0x5A3D          $47AF3D1E       #FFAA33
```

CONVERSION INTO B-ARY NUMBER SYSTEMS

Method 1 (following Euclid)

Conversion from the decimal system to a system with base b

Representation of a number

$$Z = z_n 10^n + z_{n-1} 10^{n-1} + \ldots + z_1 10 + z_0 + z_{-1} 10^{-1} + \ldots + z_{-m} 10^{-m}$$
$$= y_p b^p + y_{p-1} b^{p-1} + \ldots + y_1 b + y_0 + y_{-1} b^{-1} + \ldots + y_{-q} b^{-q}$$

Generate the digits step-by-step starting with the most significant (leftmost) digit:

Step 1: search $\boldsymbol{p}$ according to the inequation $b^p \leq Z < b^{p+1}$

assign $i = p$ and $Z_i = Z$

Step 2: derive $\boldsymbol{y_i}$ and the remainder $\boldsymbol{R_i}$ by division of $\boldsymbol{Z_i}$ by $\boldsymbol{b^i}$

$\boldsymbol{y_i} = \boldsymbol{Z_i}$ **div** $\boldsymbol{b^i}$

$\boldsymbol{R_i} = \boldsymbol{Z_i}$ **mod** $\boldsymbol{b^i}$

Step 3: Repeat step 2 for $i = p\text{-}1,\ldots$ and replace after each step $\boldsymbol{Z_i}$ by $\boldsymbol{R_i}$, until $\boldsymbol{R_i} = \boldsymbol{0}$ or until $\boldsymbol{b^i}$ is small enough (thus the precision high enough).

Method 1: example

Conversion of 15741.233_{10} into the hexadecimal system

Step 1: $16^3 \leq 15741{,}233 < 16^4$ ➡ highest power is 16^3

Step 2:	15741.233	:	16^3	=	3	remainder	3453.233
Step 3:	3453.233	:	16^2	=	D	remainder	125.233
Step 4:	125.233	:	16^1	=	7	remainder	13.233
Step 5:	13.233	:	$16^0=1$	=	D	remainder	0.233
Step 6:	0.233	:	16^{-1}	=	3	remainder	0.0455
Step 7:	0.0455	:	16^{-2}	=	B	remainder	0.00253
Step 8:	0.00253	:	16^{-3}	=	A	remainder	0.000088593
Step 9:	0.000088593	:	16^{-4}	=	5	remainder	0.000012299

➡ **error!**

➡ **$15741.233_{10} \approx 3D7D.3BA5_{16}$**

Method 2 (following Horner)

Conversion from the decimal system to a system with base b

Two steps: First consider the integer part of a number, than the decimals

Conversion of the integer part:

If we factor out the integer $X_b = \sum_{i=0}^{n} z_i b^i$ we get:

$$X_b = (((...(((z_n\ b + z_{n-1})\ b + z_{n-2})\ b + z_{n-3})\ b\ ...\ )\ b + z_1)\ b + z_0$$

Method 2: example (part 1: integer)

Repeated **division** of the integer part by the base b.

The remainders are the digits of the number X_b from the least to the most significant position.

Conversion of 15741_{10} into the hexadecimal system

$15741_{10} : 16 = 983$	remainder 13	(D_{16})
$983_{10} : 16 = 61$	remainder 7	(7_{16})
$61_{10} : 16 = 3$	remainder 13	(D_{16})
$3_{10} : 16 = 0$	remainder 3	(3_{16})

➡ **$15741_{10} = 3D7D_{16}$**

Method 2: part 2 – conversion of the decimals

We can also write the decimals of a number $Y_b = \sum_{i=-m}^{-1} y_i b^i$ in the following way:

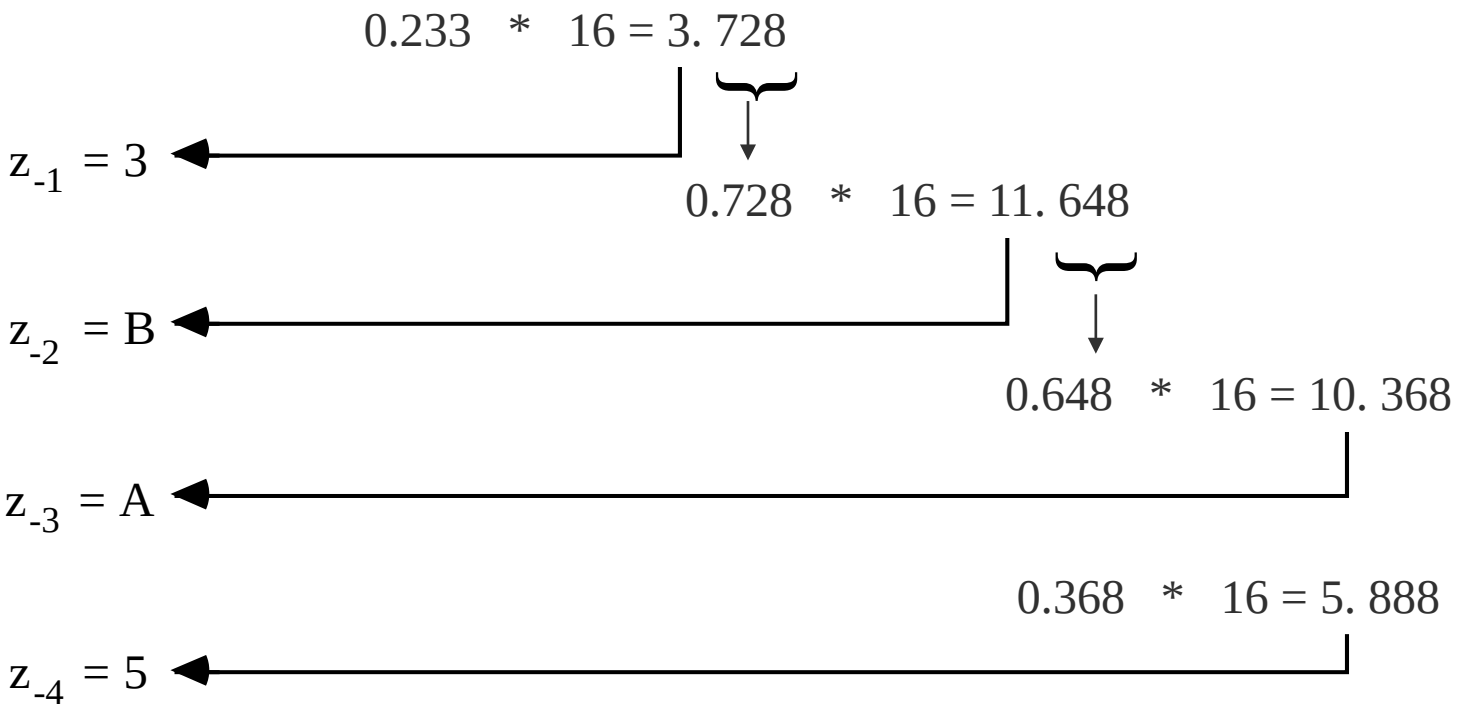
$$Y_b = ((...((y_{-m}\, b^{-1} + y_{-m+1})\, b^{-1} + y_{-m+2})\, b^{-1} + ... + y_{-2})\, b^{-1} + y_{-1})\, b^{-1}$$

Method**:**

We **multiply** the decimals of the number by base b to get the fractional digits y_{-i} from the most to the least significant position. (But we have to stop if the precision is good enough…)

Method 2: example (part 2: decimals)

Conversion of 0.233_{10} into the hexadecimal system:



➡ $0.233_{10} \approx 0.3BA5_{16}$

Stop if precision is good enough

➡ **error!**

CONVERSION INTO THE DECIMAL SYSTEM

Conversion: Base b → Base 10

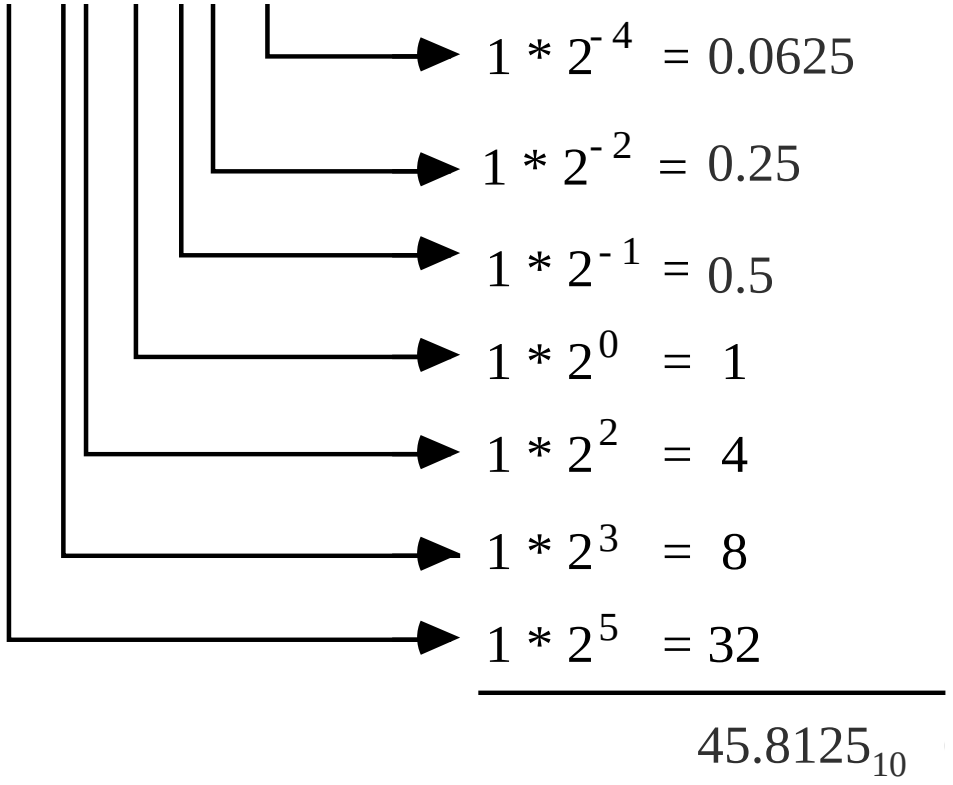
We represent the values of the single positions of the number we want to convert in our common decimal system and sum all values.

The value X_b of the number is the sum of all single values of all positions $z_i b^i$:

$$X_b = z_n b^n + z_{n-1} b^{n-1} + \cdots + z_1 b^1 + z_0 b^0 + z_{-1} b^{-1} + \cdots + z_{-m} b^{-m} = \sum_{i=-m}^{n} z_i b^i$$

Conversion: Base $b \rightarrow$ decimal system – Example

Convert 101101.1101_2 into the decimal system



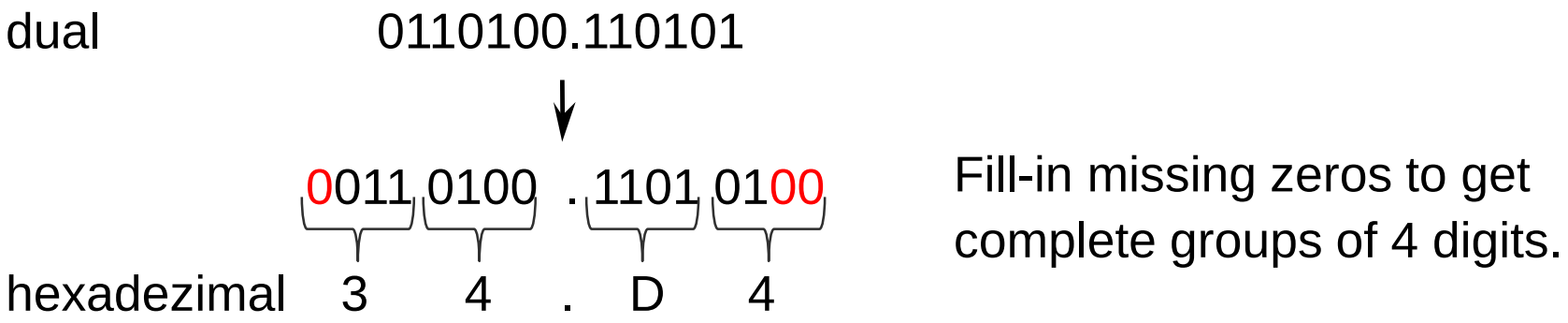
Conversion of arbitrary positional number systems

First, convert the number into the decimal system, then use method 1 or 2 to convert into the final system

Special case:

- If the base of one system is a power of the base of another system the conversion is quite simple: Replace a sequence of digits by a single digit or replace a digit by a sequence of digits, respectively.
- Example: Conversion of 0110100.110101_2 into the hexadecimal system

$2^4 = 16 \Rightarrow$ 4 binary digits $\rightarrow$ 1 hexadecimal digit



Fill-in missing zeros to get complete groups of 4 digits.

Questions & Tasks

- So, our computer does not even know simple math – what does this tell us?
- Why is the binary system so common in computers?
- Where can you find hex-notations in the context of computer systems?
- How can number conversion introduce errors?

NEGATIVE NUMBERS

Representation of negative numbers

We can use four different formats for the representation of negative numbers in computers:

- Absolute value plus sign (V+S)

- Ones' complement

- Two's complement

- Offset binary / excess / biased

Representation with absolute value plus sign (V+S)

One digit represents the sign, typically the MSB

- MSB = Most Significant Bit

The leftmost bit represents the sign of a number (by convention)

- MSB = 0 ➡ positive number
- MSB = 1 ➡ negative number

Example:

- 0001 0010 = +18
- 1001 0010 = -18

Disadvantages:

- Separate handling of the signs during addition and subtraction
- There are two representations of the number 0
  - One with positive and one with negative sign (+0 and -0)

Ones' complement

Flip all single bits of a binary number to get the number with a reversed sign.

This is called a ones' complement

- n is the number of digits, e.g. $n=4$ ➡ 4 bit numbers

$$z_{oc} = (2^n - 1) - z$$

Example:

$4_{10} = 0100_2$ ➡ $-4_{10} = 1011_{oc}$

$-4_{10} = (2^4 - 1) - 4 = 11_{10} = 1011_2$

Again, negative numbers have the MSB = 1

Advantage (compared to absolute value plus sign)

- No separate handling of the MSB during addition or subtraction

Disadvantage

- Still two representations of zero (0000 and 1111 for 4 bit numbers)

Two's complement

Avoid the disadvantage by adding 1 after applying ones' complement:

This results in the two's complement:

$$z_{tc} = 2^n - z$$

Only one representation of the zero!

$0 \ldots 0$ ➡ $1 \ldots 1_{oc}$

➡ $0 \ldots 0_{tc}$

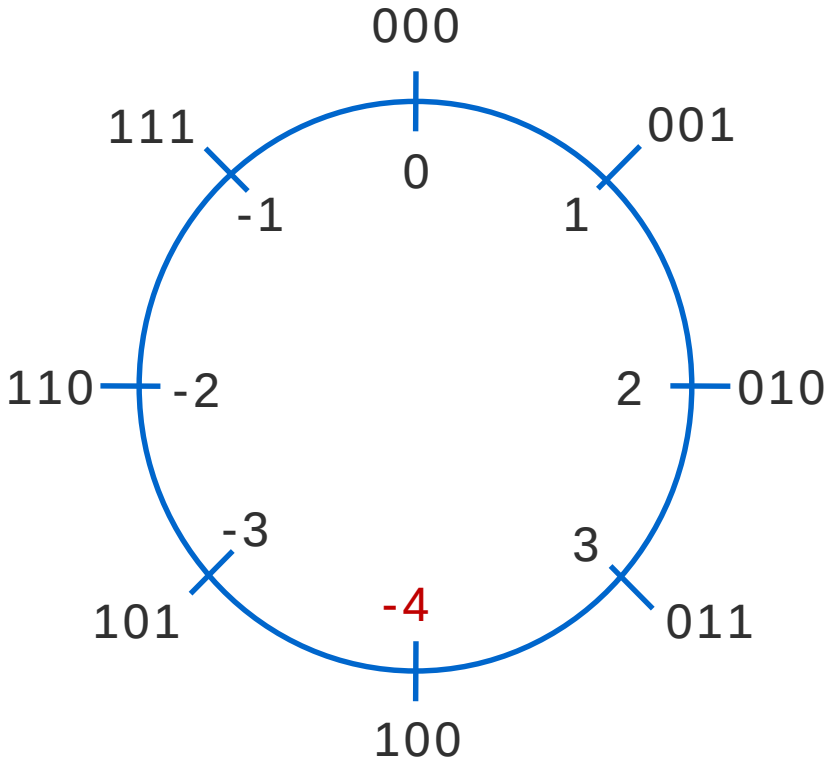
Two’s complement

Disadvantage:

- Asymmetric interval of numbers that can be represented
- The lowest number has a greater absolute value (by 1) than the highest number

Example: 3 bit two’s complement numbers

Again, negative numbers have the MSB = 1



Representation of negative numbers – examples

Represent -77_{10} using 8 bits

$$77_{10} = 0100\ 1101_2$$

Flip all the bits

Value plus sign: $-77 = 1100\ 1101_2$

Ones‘ complement : $-77 = 1011\ 0010_2$

Add 1

Two‘s complement : $-77 = 1011\ 0011_2$

Offset binary / excess / biased representation

Commonly used for the representation of exponents of floating point numbers (but also e.g. in signal processing as the converters are unipolar, i.e., they cannot handle negative values).

This representation of an exponent is also called **characteristic**.

The whole number range is shifted by adding a constant value (offset/excess/bias) so that the smallest number (largest negative value) gets the representation **0…0**.

Assuming n digits: **Offset** = 2^{n-1}

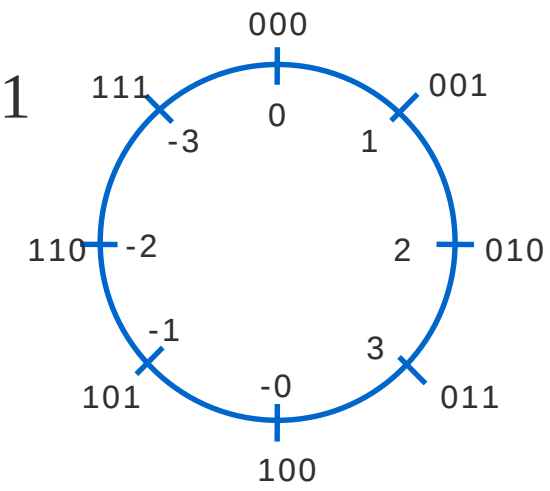
- Example: $n=8$ ➡ Offset 128

The number range is **asymmetric**.

Comparison of different representations

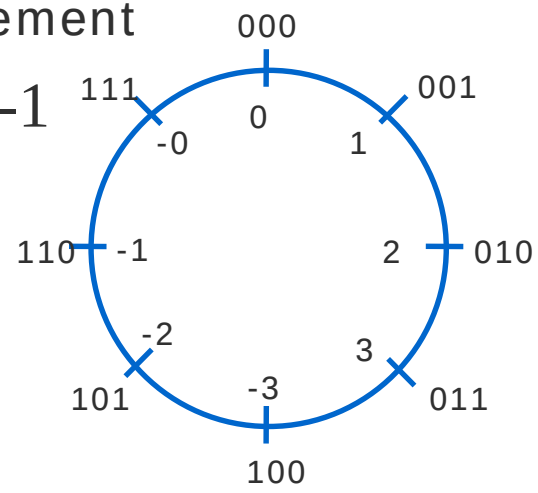
Value plus sign

$-(2^{n-1}-1) : 2^{n-1}-1$



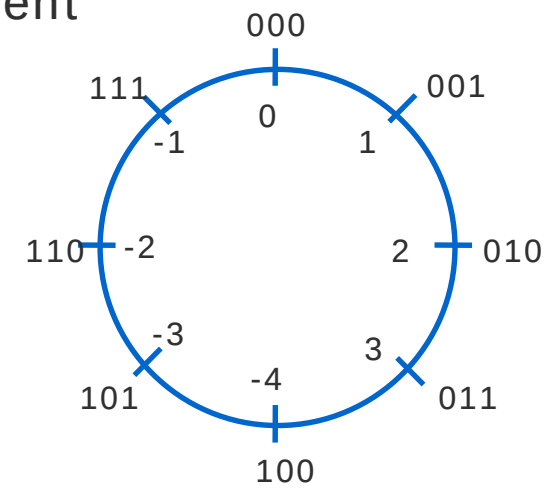
Ones‘ complement

$-(2^{n-1}-1) : 2^{n-1}-1$



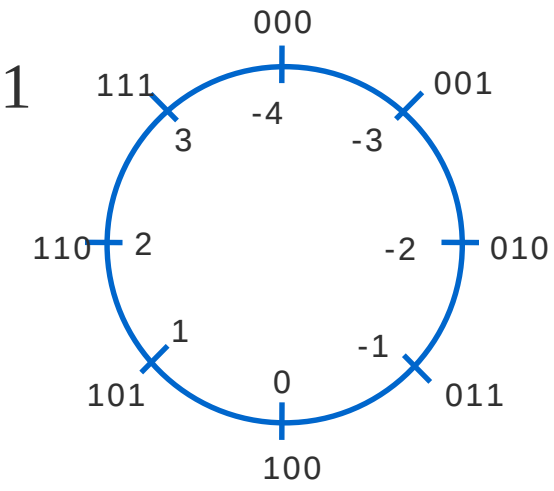
Two‘s complement

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



Offset

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



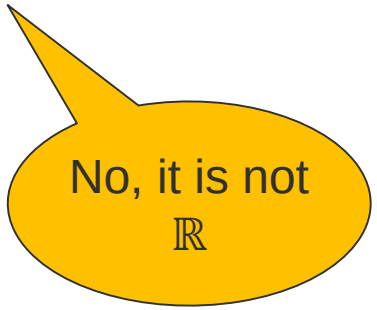
Overview of the representations

Representation as				
Decimal value	Value + Sign	Ones‘ complement	Two‘s complement	Characteristic
-4	- - -	- - -	1 0 0	0 0 0
-3	1 1 1	1 0 0	1 0 1	0 0 1
-2	1 1 0	1 0 1	1 1 0	0 1 0
-1	1 0 1	1 1 0	1 1 1	0 1 1
0	1 0 0, 0 0 0	1 1 1, 0 0 0	0 0 0	1 0 0
1	0 0 1	0 0 1	0 0 1	1 0 1
2	0 1 0	0 1 0	0 1 0	1 1 0
3	0 1 1	0 1 1	0 1 1	1 1 1

Questions & Tasks

- What do you get if you add 1 to the largest positive number?
- What happens if you subtract 1 from the largest (by absolute value) negative number?
- Thus, what should be done even for simple arithmetic operations like + and – (think of $x + y < x, y > 0$)?
- Can you represent $\mathbb{N}$ or $\mathbb{Z}$ in a computer?

“REAL” NUMBERS (FIXED AND FLOATING POINT)



Fixed and floating point numbers

Writing numbers on paper we use:

- digits 0 1 2 3 4 5 6 7 8 9
- sign + -
- point .

Representing numbers in a computer we only have:

- Binary digits (i.e. bits) 0 1

➡ We need rules for representing the value, the sign and the radix point (typically binary point) in a computer

Representing the sign and value: done (see above)

Two ways of representing the point

- Fixed point
- Floating point

Fixed point numbers

Convention

- The point is (virtually) located at a **fixed position** within the bit vector representing a binary number.
- Typically, the point follows the LSB (least significant bit).

Characteristic

- Arbitrary numbers can be scaled into this format.
- Negative numbers: use two’s complement.
- Computers typically do not use fixed point numbers internally, but for input and output (e.g. think of amount of money, 37.42€)

Fixed point numbers

The type "integer" is a special fixed point format.

Some programming languages allow for the definition of integers of different length.

Size (bit)	Typical names	Sign	Number range (using two's complement)	
			min	max
8	char, octet, byte, modern: **int8_t** or **uint8_t**	signed	−128	127
		unsigned	0	255
16	Word, Short/short, Integer, modern: **int16_t** or **uint16_t**	signed	−32,768	32.767
		unsigned	0	65.535
32	DWord/Double Word, int, long (Windows on 16/32/64 bit systems; Unix/Linux on 16/32 bit systems), modern: **int32_t** or **uint32_t**	signed	−2,147,483,648	2,147,483,647
		unsigned	0	4,294,967,295
64	Int64, QWord/Quadword, long long, Long/long (Unix/Linux on 64 bit systems), modern: **int64_t** or **uint64_t**	signed	−9,223,372,036,854,775,808	9,223,372,036,854,775,807
		unsigned	0	18,446,744,073,709,551,615
128	Int128, Octaword, Double Quadword	signed	$\approx -1.70141 \cdot 10^{38}$	$\approx 1.70141 \cdot 10^{38}$
		unsigned	0	$\approx 3.40282 \cdot 10^{38}$

Source: Wikipedia

FLOATING POINT NUMBERS

FIRST: ABSTRACT VIEW

BE AWARE: COMPUTERS USE THE IEEE-P 754-FLOATING-POINT-STANDARD

Floating point representation of numbers

To represent very large or very small numbers we typically use the scientific notation know from school chemistry or physics (so-called log-linear representation):

$$X = \pm \text{significand} \cdot b^{\text{exponent}}$$

The base b is fixed for a certain floating point representation (typically 2 or 16) and, thus, is typically not represented explicitly.

Be aware: floating point numbers typically do not use the two's complement but value plus sign.

The significand is sometimes also called mantissa or fraction, see https://en.wikipedia.org/wiki/Significand. (several different definitions exist…)

Floating point representation

The position of the radix point of the significand is by convention (e.g. left of the MSB)

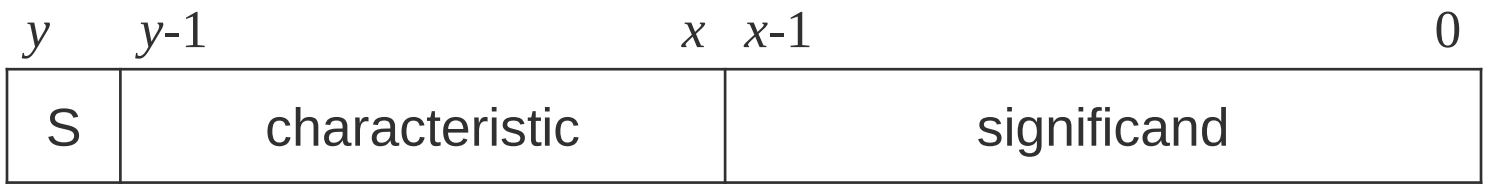
The exponent is an integer represented by its characteristic.

The computer uses a fixed number of digits for the significand and the characteristic.

The size of the characteristic determines the number range.

The size of the significand determines the precision of the representation.

Floating point format



$$\text{decimal value} = (-1)^{S} \times (0.\text{significand}) \times b^{\text{exponent}}$$

$$\text{exponent} = \text{characteristic} - b^{(y-1)-x}$$

Normalization

A floating point number is called normalized if the following holds for the mantissa:

$$\frac{1}{b} \leq significand < 1$$

Using a binary representation this means that the first digit after the binary point equals 1, i.e., all normalized numbers start with 0.1….

Exception: For the number 0 all digits are zero (0.0…0)

Normalization

Assuming a special bit pattern for the 0, the first digit of the significand always equals 1.

Therefore, it is not necessary to represent this first digit of the significand internally.

This bit is called the “hidden bit”.

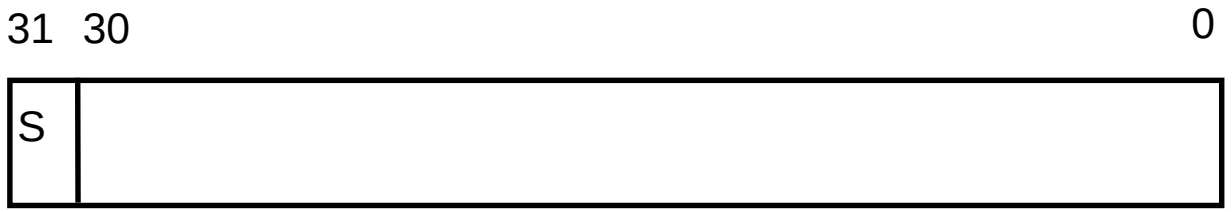
This saves a bit of memory per number or increases the precision using the same number of bits.

Be aware: for all arithmetic operations and during the conversion into other representations this digit must not be neglected!

Example: representation of 7135_{10}

3 different formats with 32 bit each using the base b = 2

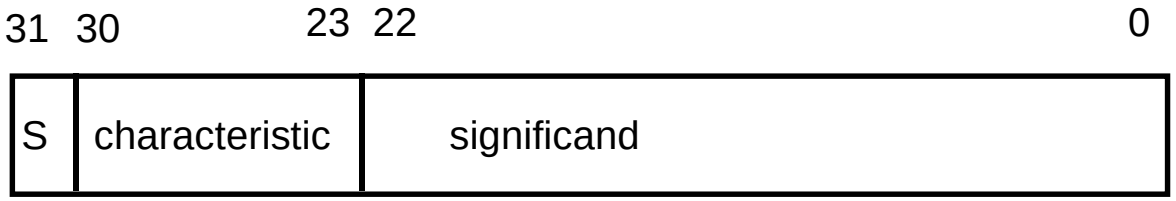
Format a: Fixed point using the two's complement (typical integer)



0 000 0000 0000 0000 0001 1011 1101 1111_2 = 0000 $1BDF_{16}$

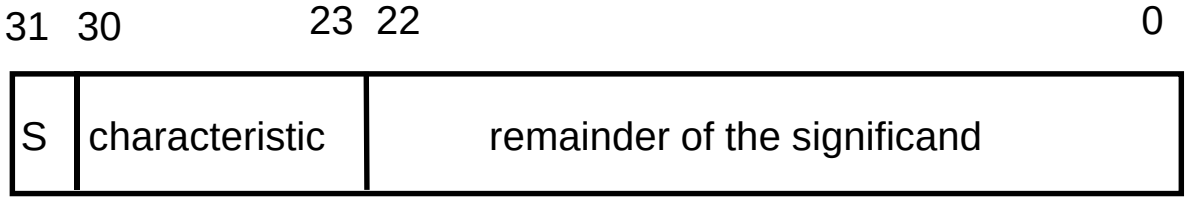
Example: representation of 7135_{10}

Format b: Floating point, normalized:



$0\ 100\ 0110\ 1\ 110\ 1111\ 0111\ 1100\ 0000\ 0000_2 \quad = \quad 46EF\ 7C00_{16}$

Format c: Floating point, normalized, first "1" implicit (hidden bit):



$0\ 1000\ 1101\ \ 101\ 1110\ 1111\ 1000\ 0000\ 0000_2 \quad = \quad 46DE\ F800_{16}$

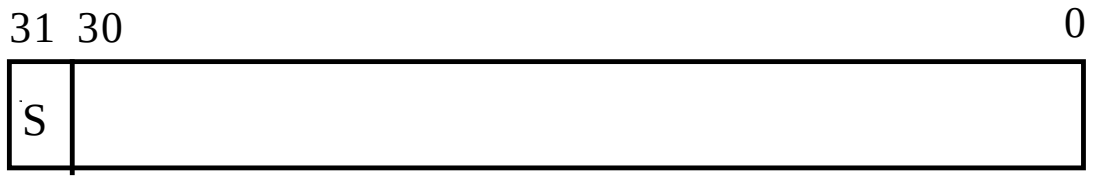
Representable number range

The number bit combinations is the same for all three examples (2^{32})

However, the number range and, thus, the density of representable values on the number line is quite different!

Representable number range

Format a) Numbers between -2^{31} und $2^{31}-1$

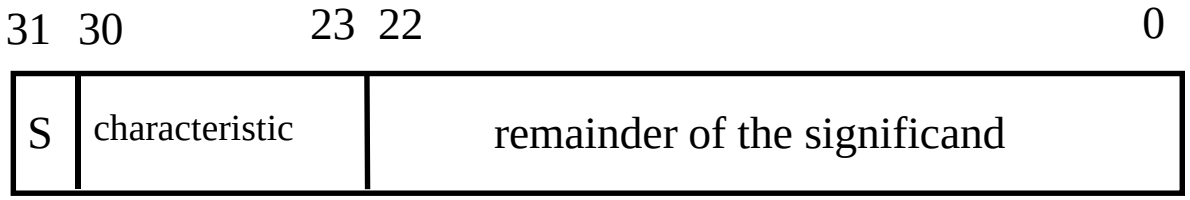


Format b)

negative numbers	$-(1-2^{-23}) \cdot 2^{127}$	...	$-0.5 \cdot 2^{-128}$
positive numbers	$0.5 \cdot 2^{-128}$	...	$(1-2^{-23}) \cdot 2^{127}$
and	± 0		

Representable number range

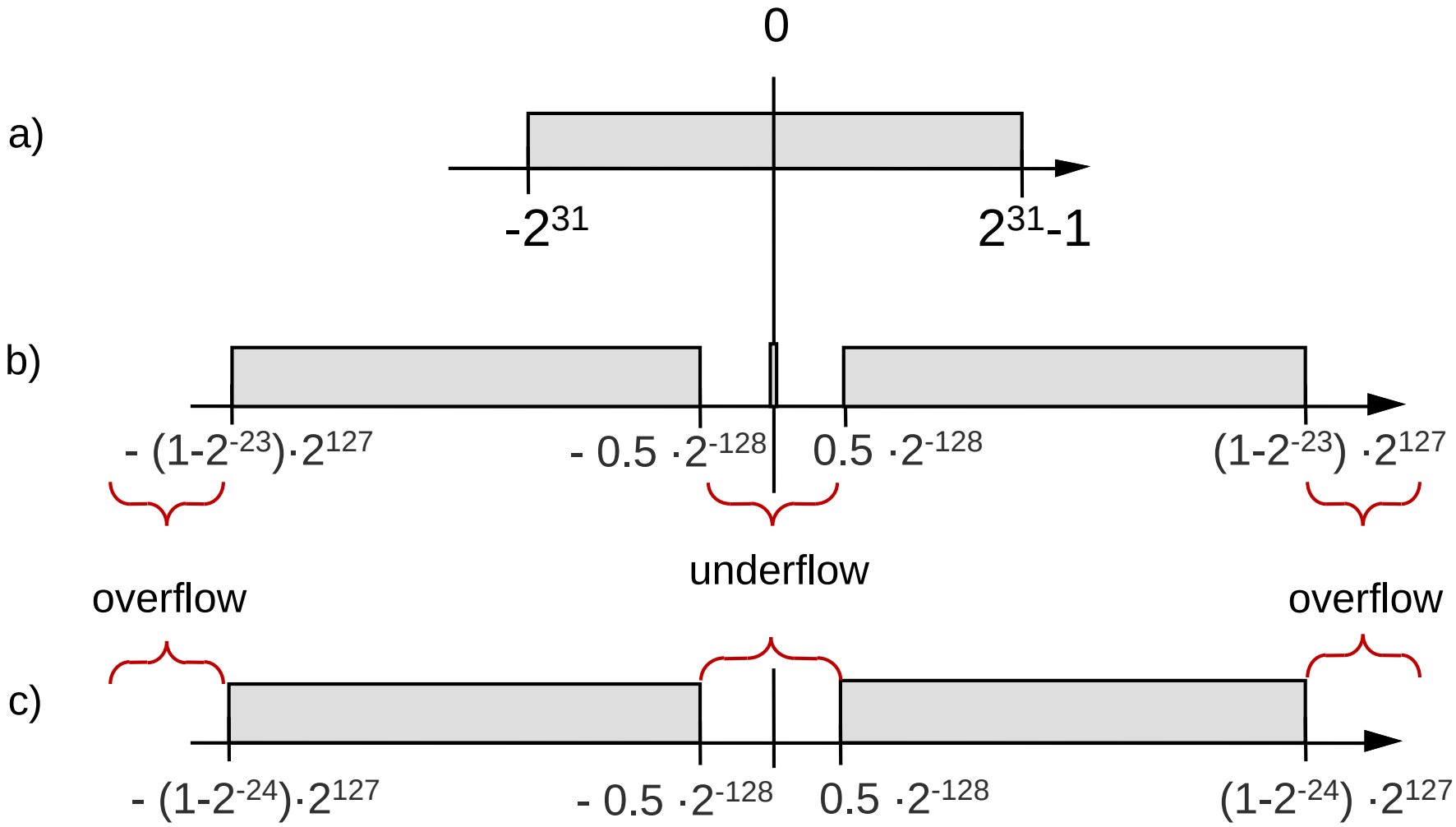
Format c) normalized floating point



negative numbers	$-(1-2^{-24})\cdot 2^{127}$	...	$-0.5\cdot 2^{-128}$
positive numbers	$0.5\cdot 2^{-128}$	...	$(1-2^{-24})\cdot 2^{127}$

No representation of the 0!

Representable number range



Characteristic numbers

In order to compare different floating point formats three characteristic numbers are useful:

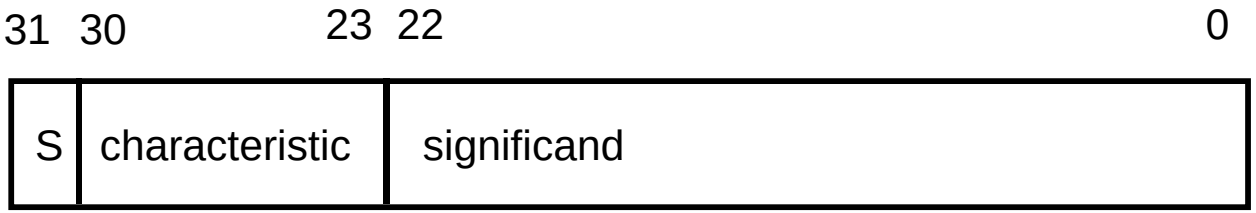
- Maxreal: the largest representable, normalized, positive number

- Minreal: the smallest representable, normalized, positive number

- Smallreal: the smallest number that can be added to 1 to get a result larger than 1

Characteristic numbers – example

Using format b)



maxreal = $(1 - 2^{-23}) \cdot 2^{127}$ minreal = $0.5 \cdot 2^{-128}$

If we normalize 1 we get $0.5 \cdot 2^{1}$ i.e. the significand is 10000000000000000000000.

The closest number larger than 1 representable in this format has in addition to the “1” in bit 22 also a “1” in bit 0.

This results in the significand: 10000000000000000000001

Thus, smallreal = $0.00000000000000000000001_2 \cdot 2^{1} = 2^{-23} \cdot 2^{1} = 2^{-22}$

Imprecisions

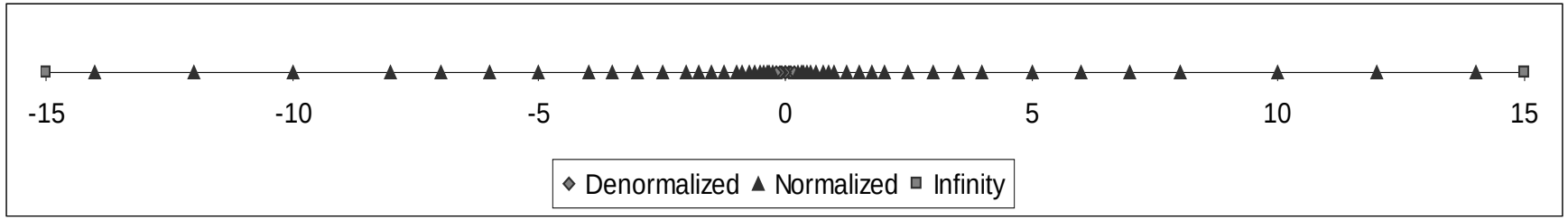
The gap between two representable floating point numbers grows exponentially with the absolute value of the numbers, while this gap is constant for fixed point numbers.

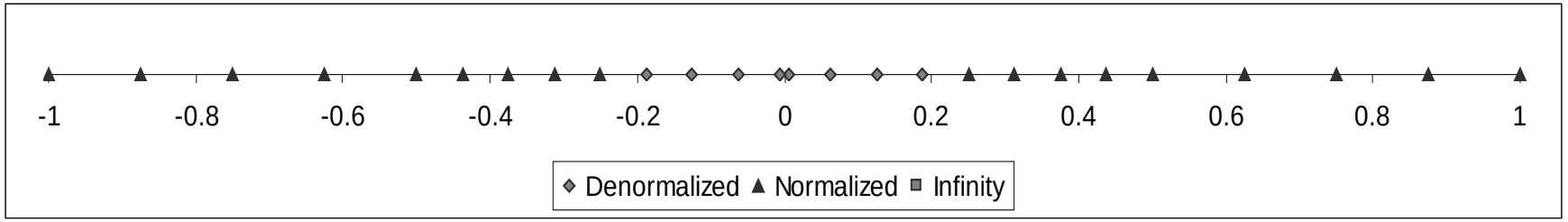
Therefore, the imprecision when representing large values is higher. There is a trade-off between range and precision.

Computers violate the laws of math valid for real numbers $x, x \in \mathbb{R}$!

- (although some programming languages call this type **`real`**)

Imprecisions





Source: Computer Systems: A Programmer's Perspective

Example

The associative property $(x + y) + z = x + (y + z)$ does not always hold, even if there is no overflow or underflow.

Let's assume: $x = 1; y = z = \text{smallreal}/2$

$$
\begin{aligned}
(x + y) + z &= (1 + \text{smallreal}/2) + \text{smallreal}/2 \\
&= 1 + \text{smallreal}/2 \\
&= 1
\end{aligned}
$$

$$
\begin{aligned}
x + (y + z) &= 1 + (\text{smallreal}/2 + \text{smallreal}/2) \\
&= 1 + \text{smallreal} \\
&\neq 1
\end{aligned}
$$

Questions & Tasks

- Which format can represent more different values: 64 bit integer or floating point?
- Why using floating points at all?
- How can you increase the range or precision of floats?
- Why do we normalize floats?
- Why can we live with the imprecisions of floats for many scenarios?
- How can we achieve a higher precision or greater range? At what cost?
- What can we learn from the smallreal example when it comes to the addition of numbers?

IEEE P 754 FLOATING POINT STANDARD

Standardization (IEEE Standard)

IEEE P 754 floating point standard

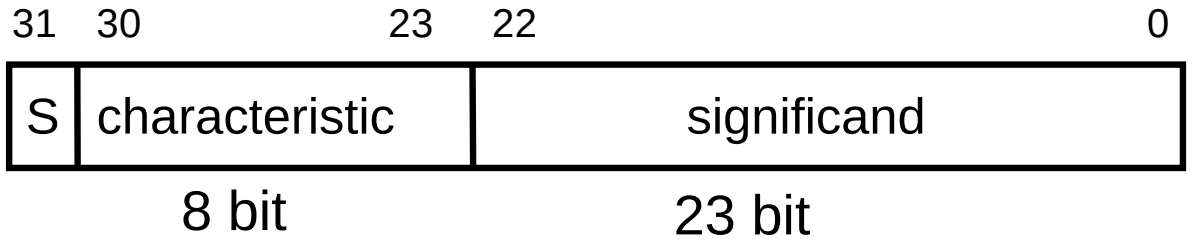
Many programming languages know floating point numbers with different precision and range

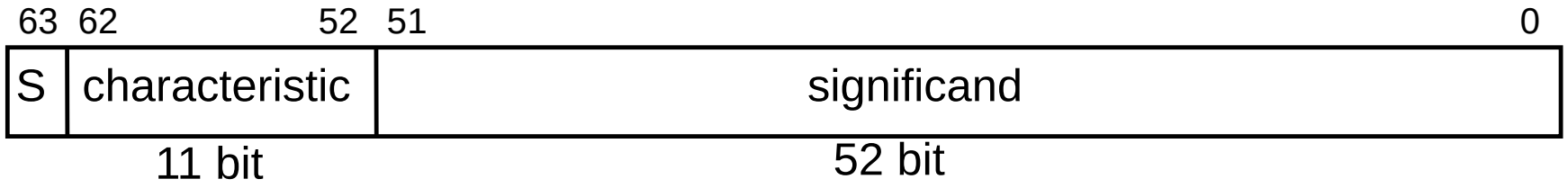
- E.g. using C: float
  double
  long double

The IEEE Standard defines several formants for representation, e.g.,

- IEEE single: 32 bit
  IEEE double: 64 bit
  IEEE extended: 80 bit

IEEE P 754 floating point standard





Example formats of the IEEE standard

Be aware that you will also find the term exponent for characteristic and the terms mantissa or fraction for significand!

Properties of IEEE P 754

Base b equals 2.

The first bit of the significand is assumed to be “1” (hidden bit), if the characteristic ≠ 0.

Normalization: the hidden bit is left of the binary point, i.e., 1.xxxxx (hidden bit convention)

If the characteristic = 0 it represents the same exponent as if the characteristic = 1.

However, the first bit of the significand is the represented explicitly. The numbers represented this way are called subnormal (denormalized/denormal, https://en.wikipedia.org/wiki/Denormal_number).

This allows the representation of ±0 (significand and characteristic = 0).

Properties of IEEE P 754

If all bits of the characteristic are “1” this indicates an exception.

- If all bits of the significand are “0” this represents an overflow, i.e. ± ∞. The processor can then trigger error handling.
- If the significand is ≠ 0 this is called NaN (not a number) and is used for signaling, e.g., invalid operations like the square root of a negative number

Internally, processors may use the IEEE standard with 80 bit to minimize rounding errors.

Some parameters of IEEE P 754

Name	Common name	Base	Significand bits[b] or digits	Decimal digits	Exponent bits	Decimal E max	Exponent bias[11]	E min	E max
binary16	Half precision	2	11	3.31	5	4.51	$2^4-1 = 15$	−14	+15
binary32	Single precision	2	24	7.22	8	38.23	$2^7-1 = 127$	−126	+127
binary64	Double precision	2	53	15.95	11	307.95	$2^{10}-1 = 1023$	−1022	+1023
binary128	Quadruple precision	2	113	34.02	15	4931.77	$2^{14}-1 = 16383$	−16382	+16383
binary256	Octuple precision	2	237	71.34	19	78913.2	$2^{18}-1 = 262143$	−262142	+262143
decimal32		10	7	7	7.58	96	101	−95	+96
decimal64		10	16	16	9.58	384	398	−383	+384
decimal128		10	34	34	13.58	6144	6176	−6143	+6144

https://en.wikipedia.org/wiki/IEEE_754

Overview of the 64 bit IEEE format

characteristic	number	remark
0	$(-1)^S$ 0.significand · 2^{-1022}	subnormal
1	$(-1)^S$ 1.significand · 2^{-1022}	
…	$(-1)^S$ 1.significand · $2^{\text{characteristic} - 1023}$	
2046	$(-1)^S$ 1.significand · 2^{1023}	
2047	significand = 0: $(-1)^S \infty$	overflow
2047	significand ≠ 0: NaN	not a number

Literature

IEEE Computer Society:

- IEEE Standard for Binary Floating-Point Arithmetic ANSI/IEEE Standard 754-1985, SIGPLAN Notices, Vol. 22, No. 2, pp 9-25, 1978

D. Goldberg:

- What every computer scientist should know about floating point arithmetic, ACM Computing Surveys, Vol. 13, No. 1, pp. 5-48,1991

Rounding modes

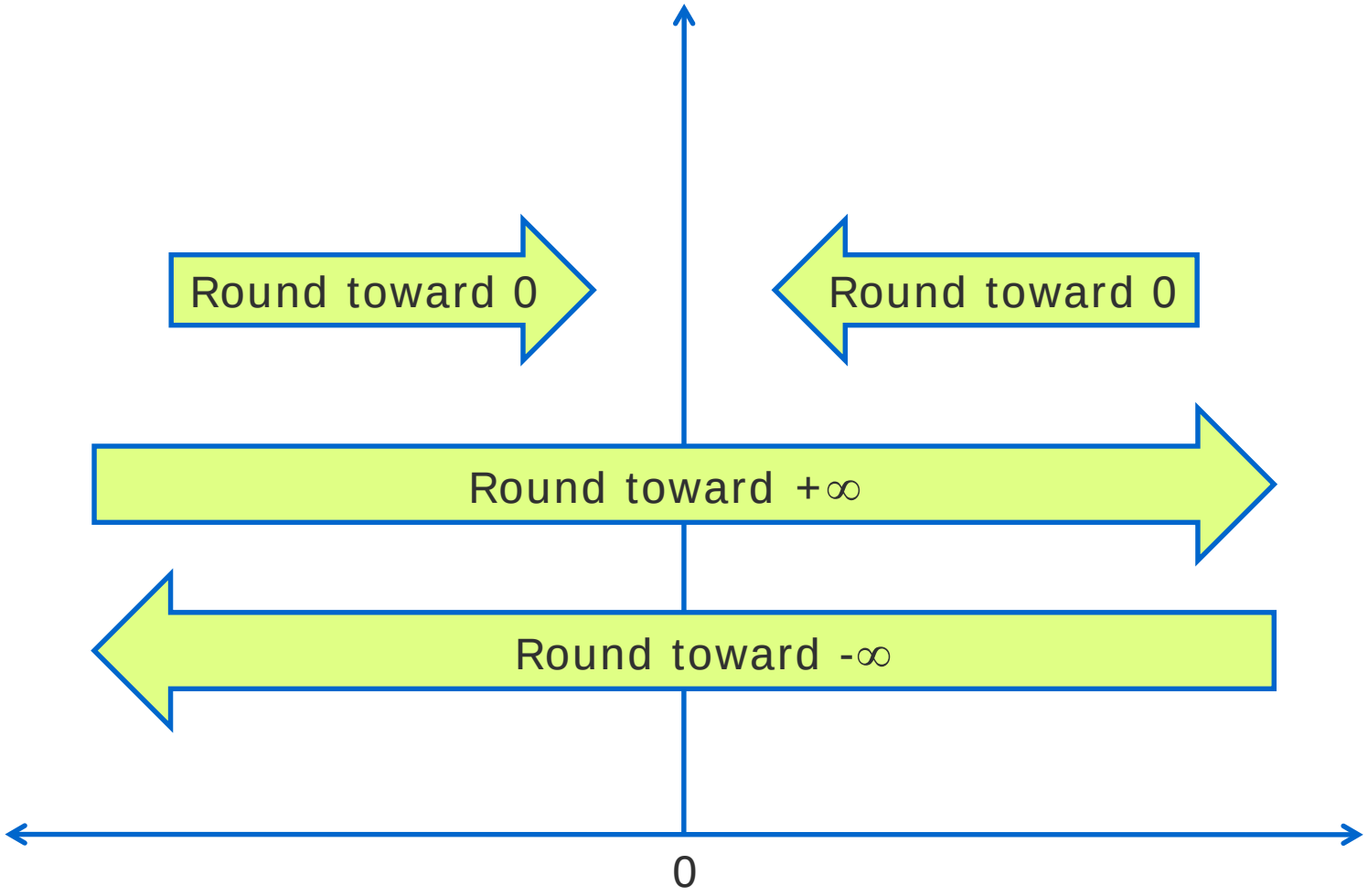
IEEE requires “correct rounding”:

- The rounded result is as if infinitively precise arithmetic was used to compute the value and then rounded at the end.
- As we will see: three extra bits are enough to achieve this!

IEEE standard defines five rounding modes:

- Round to the nearest number
  - Where ties round to the nearest even digit in the required position (“round to even”, the most common rule)
  - Where ties round away from zero
- Round to the nearest number toward 0
- Round up to the nearest number toward +∞
- Round down to the nearest number toward -∞

The three “simple” rounding modes



Round to even

The most “difficult” rounding mode (but default and the most common):

- Round to the nearest number, where ties round to even

One method: use infinite precise arithmetic, then round

- Requires very long registers, not really efficient

Can this be done with less hardware?

Two situations require rounding when adding or subtracting numbers:

- Carry
- Exponent alignment (remember: add or subtract only if the exponents are equal!)

Example a: carry during addition – 234 + 851 = 1080

We use base 10 and 3 significant digits in this example!

$$
\begin{array}{ll}
 & 2.34 \times 10^2 \\
 & +8.51 \times 10^2 \\
\text{carry} & \text{----------} \\
 & 10.85 \times 10^2 \\
\text{rounded to} & 1.08 \times 10^3
\end{array}
$$

Example b: different exponents – 234 + 2.56 = 237

(base 10 and 3 significant digits)

exponent alignment required

2.34×10^2

$+2.56 \times 10^0$ → $+0.0256 \times 10^2$

2.34×10^2

$+0.0256 \times 10^2$

2.3656×10^2

rounded to

2.37×10^2

Example c: carry and different exponents – 951 + 64.2 = 1020

(base 10 and 3 significant digits)

(alignment and carry)

$$
\begin{array}{rl}
 & 9.51 \times 10^2 \\
+ & 0.642 \times 10^2 \\
\hline
 & 10.152 \times 10^2
\end{array}
$$

rounded to 1.02×10^3

For each of these examples we have to use “internally” more than 3 significant digits to achieve correct rounding.

There are also situations where more than 3 significant digits are needed even without rounding.

Example d: subtraction – 147 - 87.6 = 59.4

$$
\begin{array}{r}
1.47 \times 10^2 \\
-0.876 \times 10^2 \\
\hline
0.594 \times 10^2
\end{array}
$$

In this example one additional „internal“ digit is sufficient. However, there are cases where this is not sufficient.

Example e: 101 - 3.76 = 97.2

	1.01×10^2	1.01×10^2
	-0.0376×10^2	-0.037×10^2
	0.9724×10^2	0.973×10^2
rounded to	0.972×10^2	0.973×10^2

Deleting the least significant digit („6“) from 0.0376 results in 0.973 instead of 0.972.

Thus, here we need more than 3 significant digits for correct rounding!

Round and guard

If we ignore the “round to even” rule it can be shown that two additional digits are sufficient for correct rounding.

- Guard **g** - tells us if we have to take a closer look at r. If g=5, then we could be in the middle between two numbers (example: base = 10).
- Round **r** - if r > 0 then rounding is simple, as we are not in the middle – but what if r = 0? Are we exactly in the middle (i.e. can we apply round-to-even)?

However, "round-to-even“ requires some more effort.

Example f: 4.5674 + 0.00025001 = 4.5677 (and not 4.5676)

We have 5 significant digits:

4.5674×10^{0} 4.5674

2.5001×10^{-4} + 0.00025001

4.5676501

gr

rounded to 4.5677

Round r and **guard g** are not sufficient –
there are some more digits important, but how many?

Idea: Did we drop *something* while truncating the number to the significant digits?
If yes, then set a sticky-bit

Sticky bit

For a correct rounding it is sufficient to know, if all digits less significant than the round digit are equal to zero.

A single bit is enough to store this information: "sticky"-bit

If we drop a digit during alignment that was not equal to zero we set the sticky bit (if we have bits, i.e., base = 2, then the sticky bit is simply the OR over all bits less significant than the round bit).

Thus, the sticky bit tells us if we dropped *something* during alignment (and the necessary truncation due to the limitation of significant digits).

If the result ties, i.e., the distance to the next lower and higher floating point number is the same, then the sticky bit decides if we have to apply round-to-even or we are simply closer to one number.

Questions & Tasks

- Can IEEE P 754 represent numbers smaller than minreal? What about zero? What is the price to pay?
- Can IEEE P 754 represent numbers larger than maxreal?
- What does “correct rounding” mean – are there no more errors?
- How many digits do we need to get correct results?
- If base = 2, how many extra bits do we need to perform “correct rounding” according to IEEE P 745?
- What is the idea of a sticky-bit?

ADDITION AND SUBTRACTION

Addition and subtraction

Circuits for the addition of integers (fixed point, base 2, two's complement):

- The base of all arithmetic operations

Very simple:

- Subtraction ≙ addition of the negative value
- $X - Y = X + (-Y)$

We can also describe multiplication and division based on addition (however, more efficient circuits are known)

- For floating point numbers:
  - Separate processing of significand/mantissa/fraction and characteristic/exponent
  - Again, integer addition forms the base

Thus, basic types of adders are important (https://en.wikipedia.org/wiki/Adder_(electronics))

From half adder to full adder

Addition of two binary digits **a** and **b** results in a sum **s** and a carry **c**.

Truth table:

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

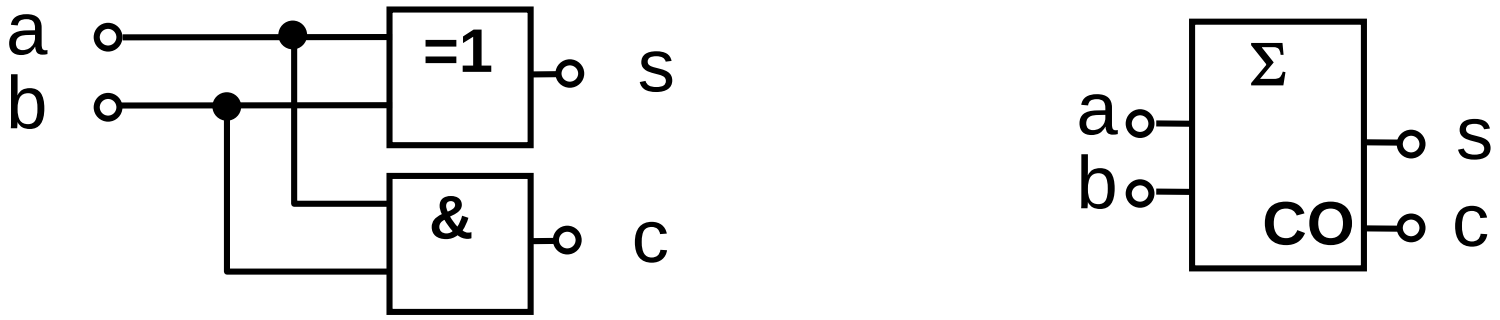
This is called a half adder

Half adder

Equations: $s = a \bar{b} \vee \bar{a} b = a \oplus b$

$c = a b$

Logic diagram and logic symbol (according to IEC):



1 bit half adder

Addition of multiple digits

Additional input for the carry of less significant digits necessary.

a_i	b_i	c_i	s_i	c_{i+1}
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

c_i is also called carry in **CI**

c_{i+1} is also called carry out **CO**

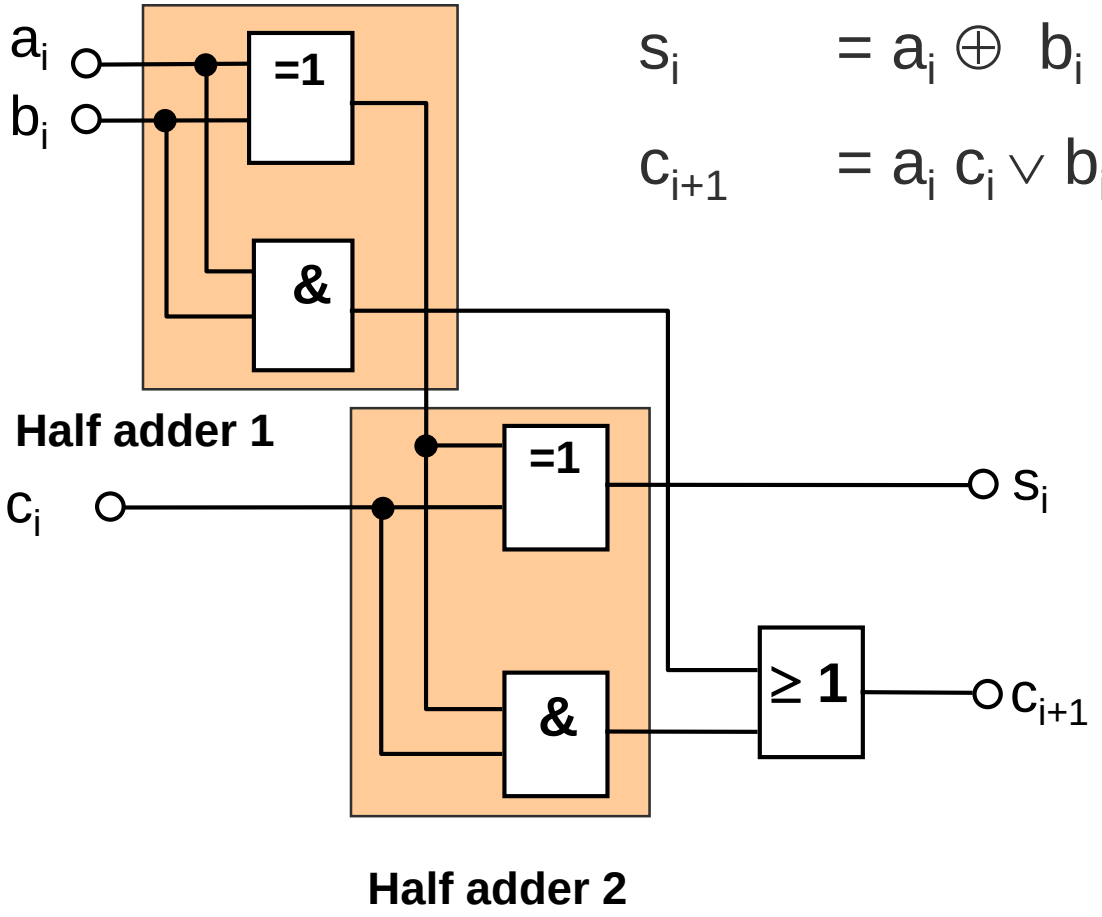
The truth table describes a full adder

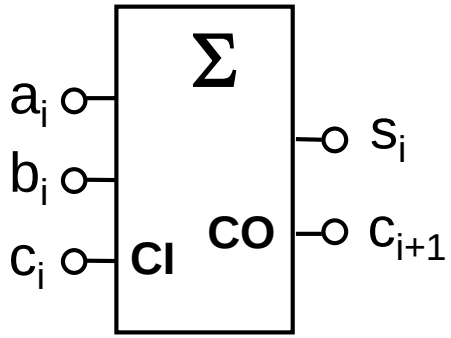
Equations, logic diagram and symbol

Equations:

$$s_i = a_i \oplus b_i \oplus c_i$$

$$c_{i+1} = a_i c_i \vee b_i c_i \vee a_i b_i = (a_i \oplus b_i) c_i \vee a_i b_i$$





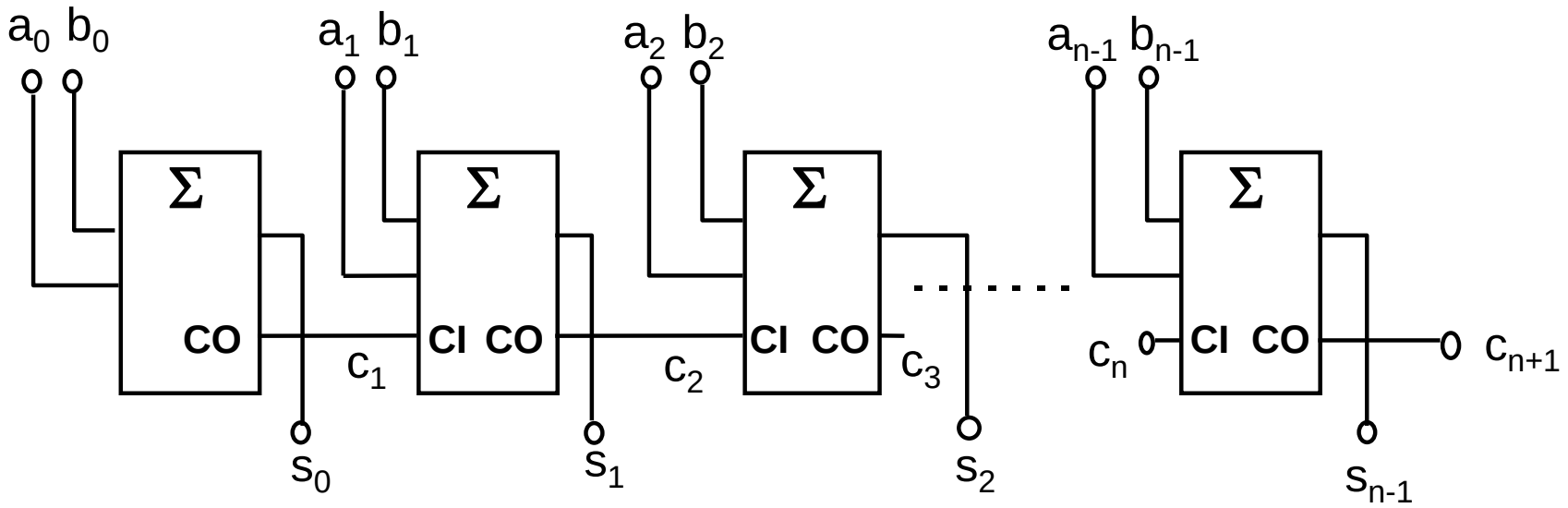
Full adder

Ripple-carry adder

How to add two integers with n bits?

Simple solution:

- Use a full adder for each bit plus take the carry of the less significant bit as carry in to generate the sum plus the carry out.
- The LSB (least significant bit) needs only a half adder.



Problem

The result of a single bit position is valid only if the carry in based on less significant bit positions is computed.

Worst case: the valid values for the carries ripple through all bit positions (thus the name ripple-carry adder).

The time needed to get a stable result is proportional to the bits added.

Therefore, the ripple-carry adder is sometimes also called an asynchronous parallel adder as it takes all bits of the operands in parallel. (asynchronous, because it depends on the value of the operand how long it takes before the output is stable)

Carry-lookahead adder

In order to avoid the disadvantage of long delays during addition using a carry-ripple adder, the carry-lookahead adder directly determines all carries based on the input operands

Equations:

- $c_{i+1} = a_i\, b_i \vee (a_i \oplus b_i)\, c_i = g_i \vee p_i\, c_i$
- $s_i = (a_i \oplus b_i) \oplus c_i = p_i \oplus c_i$

Let's define:

- $g_i = a_i\, b_i$ (generate carry) and
- $p_i = (a_i \oplus b_i)$ (propagate carry)

We can derive g_i und p_i directly from the input bits of the two operands a and b.

Direct computation of the carries based on the inputs

We can recursively solve the computation of $\mathbf{c_{i+1}}$ by using the terms for $\mathbf{c_i}$.

$c_{i+1} = g_i \vee p_i\, c_i$

This results in

$c_1 = g_0 \vee p_0\, c_0$

$c_2 = g_1 \vee p_1\, g_0 \vee p_1\, p_0\, c_0$

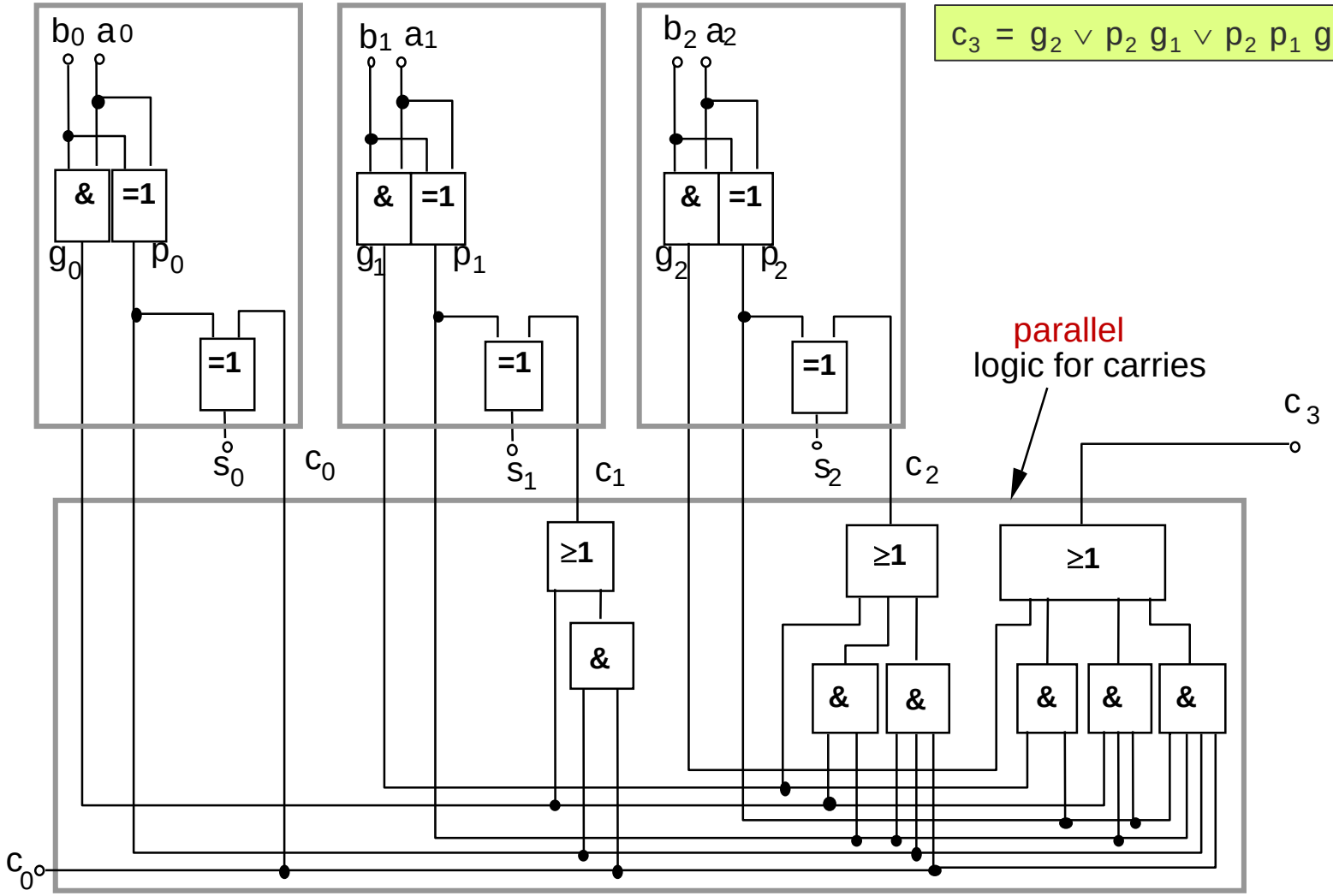
$c_3 = g_2 \vee p_2\, g_1 \vee p_2\, p_1\, g_0 \vee p_2\, p_1\, p_0\, c_0$

etc.

The time for addition is now (almost…) independent of the number of input bits as the computation of all carries can start immediately by “looking ahead” over all input bit positions – thus the name carry-lookahead adder.

Logic diagram of a 3 bit carry-lookahead addierer

$c_3 = g_2 \vee p_2\, g_1 \vee p_2\, p_1\, g_0 \vee p_2\, p_1\, p_0\, c_0$



Carry-lookahead-Addierer

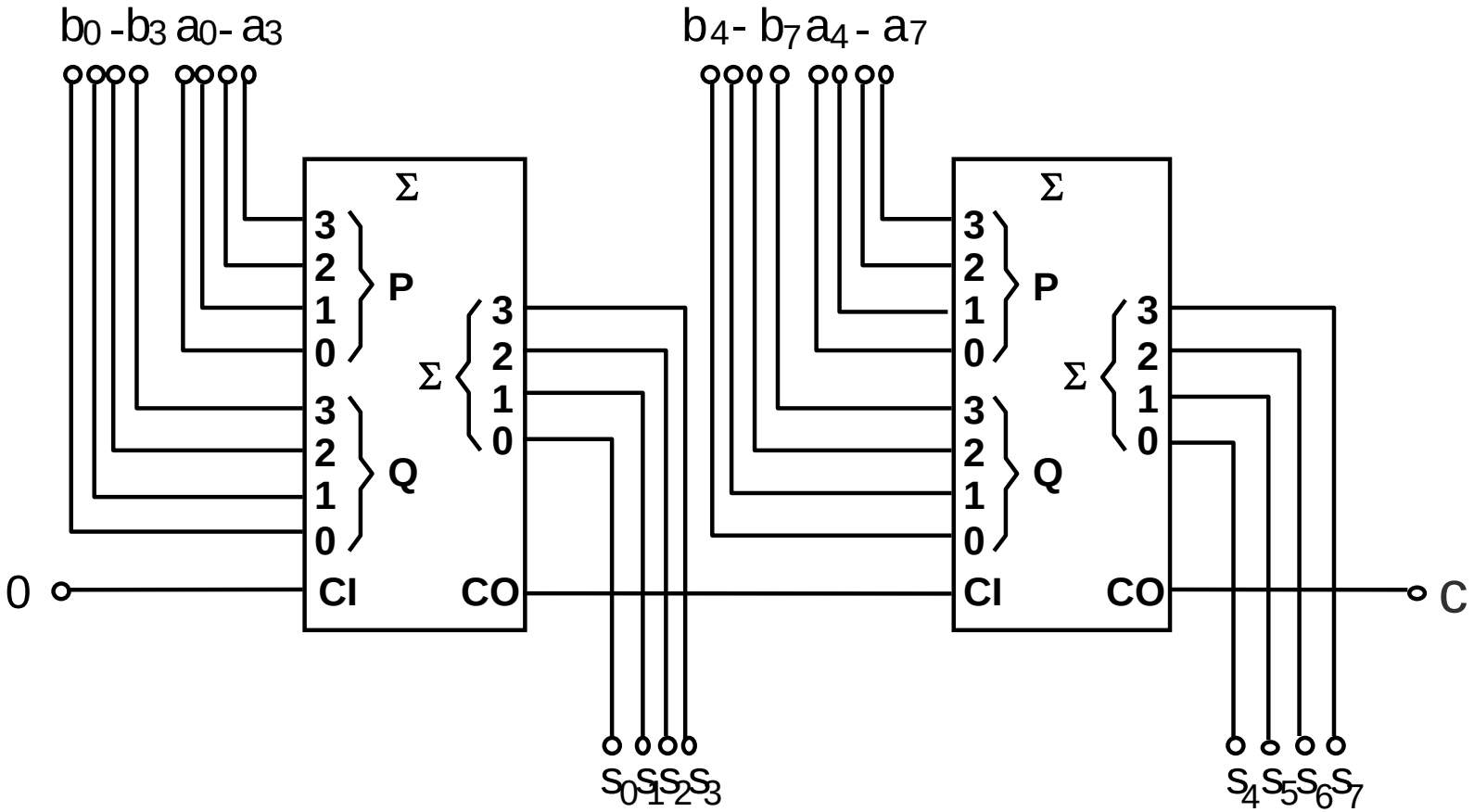
Problem:

- The number of logic gates increases with the number of input bits.
- This increases the delay of the gates plus requires more space.

Solution:

- Cascading smaller carry-lookahead adders (carry-ripple between the carry-lookahead adders)
- carry-select adder, conditional sum adder, carry skip-adder, …
  (https://en.wikipedia.org/wiki/Adder_(electronics))

Carry-lookahead adder



Cascading two 4 bit carry-lookahead adders to add 8 bit integers

Addition and subtraction

SUBTRACTION

Subtraction

Subtraction by addition of the two's complement.

Remember: Two's complement was flipping all bits and then adding 1.

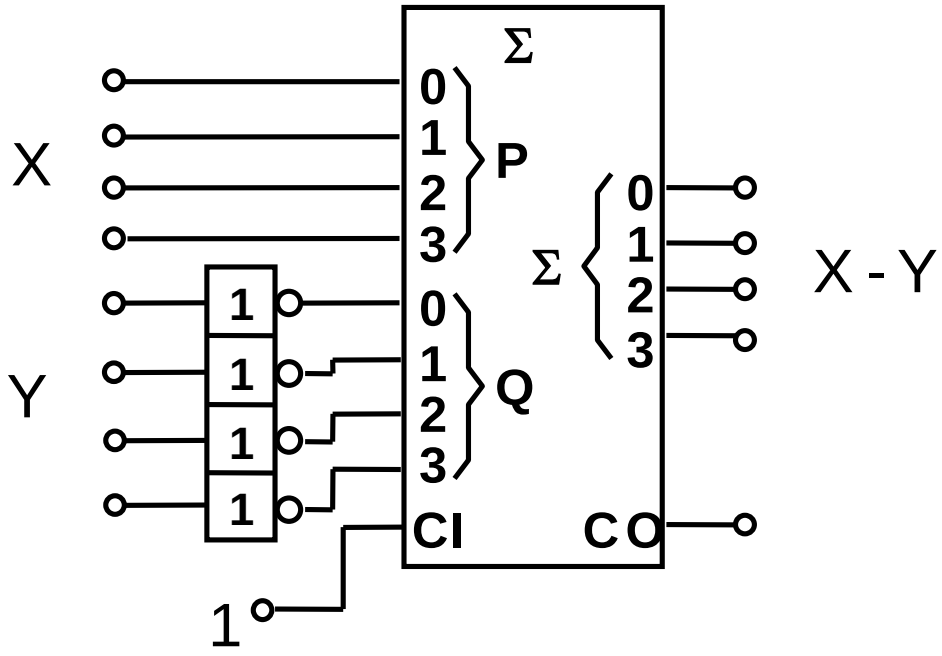
$$X - Y = X + (\overline{Y} + 1) = X + \overline{Y} + 1$$

Please note:

- We assume that both operands X and Y are in the format two's complement.
- The result is again in the format two's complement.

Subtractor

We can use an adder, take the minuend X as it is, invert the single bits of the subtrahend Y, and set the carry in CI.

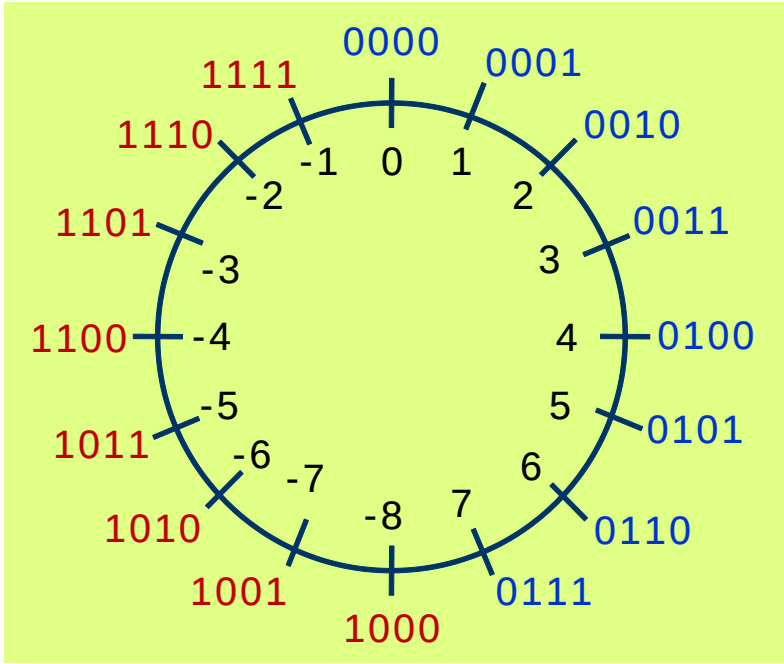


Subtraction of two numbers in two‘s complement format.

Exception 1

We can distinguish three different cases for an addition

1) Both summands are positive
   - the sign bits of both summands are 0
   - the result must be positive
   - the result is correct only if its sign bit is 0, otherwise we have an overflow




```
  5          0 1 0 1
+ 2        +   0 0 1 0
-----      -----------
= 7          0 0 0 0
             0 1 1 1
```

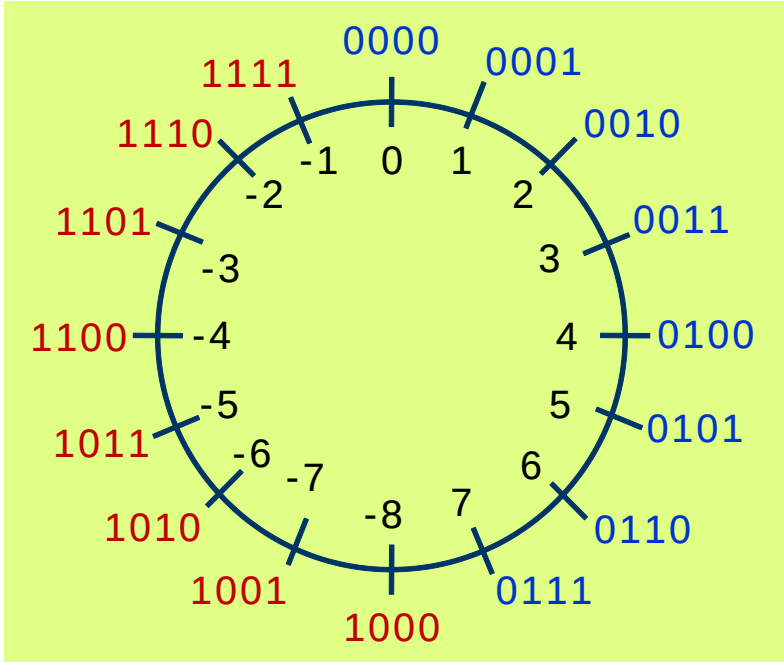
both carries are equal, thus the result is correct

```
  5          0 1 0 1
+ 6        +   0 1 1 0
-----      -----------
= 11         0 1 0 0
             1 0 1 1
```

the carries are not equal, thus the result is not correct

Exception 2

2) Both summands are negative
- the sign bits of both summands are 1
- the result must be negative
- the result is correct only if its sign bit is 1, otherwise we have an (negative) overflow



$$\begin{array}{r} -5 \\ +\ (-2) \\ \hline =\ -7 \end{array} \qquad \begin{array}{r} 1011 \\ +\ 1110 \\ \hline 1\,1\,1\,0 \\ \hline 1001 \end{array}$$

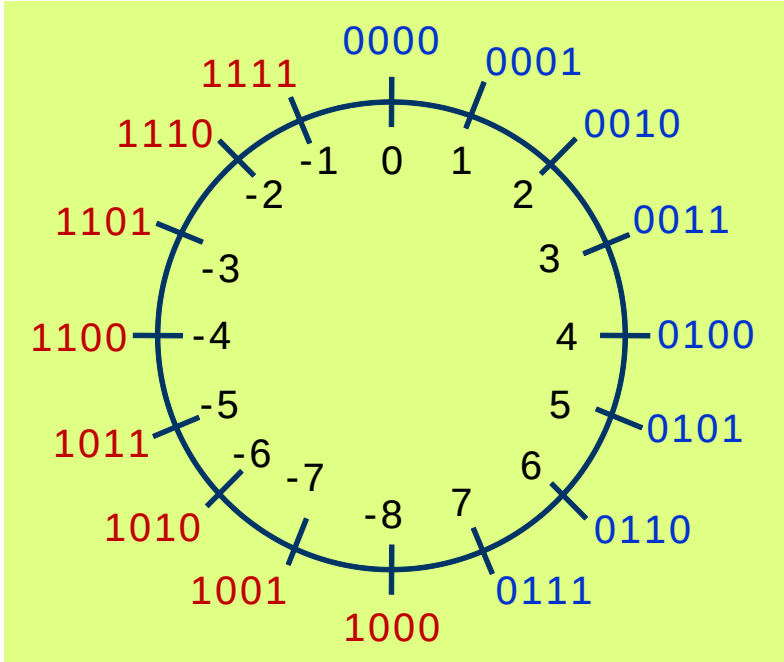
both carries are equal, thus the result is correct

$$\begin{array}{r} -5 \\ +\ (-6) \\ \hline =\ -11 \end{array} \qquad \begin{array}{r} 1011 \\ 1010 \\ \hline 1\,0\,1\,0 \\ \hline 0101 \end{array}$$

the carries are not equal, thus the result is not correct

Exception 3

3) Both summands have different signs
- the result is always correct
- the sign depends on the absolute value of the subtrahend or minuend
- we can ignore the carry generated by the MSBs




```
    5          0 1 0 1
+ (- 6)      + 1 0 1 0
-------      ---------
=  - 1       0 0 0 0
              1 1 1 1
```

both carries are equal, thus the result is correct

```
  - 5          1 0 1 1
+   6        + 0 1 1 0
-------      ---------
=   1        1 1 1 0
              0 0 0 1
```

both carries are equal, thus the result is correct

Overflow detection

It is very simple to detect an overflow during the addition/subtraction of integers in two's complement:

- correct addition: the carries are equal

- overflow: the carries are not equal

We can generate an overflow bit using an XOR (exclusive or).

Questions & Tasks

- We could use a half adder for the LSB. However, n-bit carry ripple adders can use a full adder for the LSB as well. What for?
- How does an adder react if an overflow occurs? What about the result? What about a subtractor?

FLOATING POINT ADDITION

Floating point addition

How to add two floating point numbers a_1 and a_2?

$$a_1 = s_1 \times b^{e_1} \qquad a_2 = s_2 \times b^{e_2}$$

Example: $a_1 = 3.21 \times 10^{2}$

$a_2 = 8.43 \times 10^{-1}$

3 significant digits.

Remember: Computation is done with two additional digits (guard und round) plus the sticky bit.

significand	G	R	S

Floating point addition – step 1

1. Exponent alignment

- We can add floating point numbers only if the exponents are equal

First: If $e_1 < e_2$, swap the operands to get: $d = e_1 - e_2 \geq 0$

Second: Shift right the significand s_2 by d positions

- If d > 2, set the sticky bit, if any of the dropped digits had a value $\neq 0$ (in case of binary digits, the sticky bit is the OR over all dropped bits).

Example: d = 2 - (-1) = 3

3.2100×10^2

0.0084×10^2 set the sticky bit because we dropped the 3 of 8.43

Floating point addition – steps 2 and 3

2. Add the significands

Example:

$$
\begin{array}{rl}
 & 3.2100 \times 10^2 \\
 & 0.0084 \times 10^2 \\
\text{----------} & \\
 & 3.2184 \times 10^2
\end{array}
$$

3. Normalization

- Normalize the sum by shifting the significand and adjusting the exponent/characteristic.

Floating point addition – step 4

4. Rounding

Round using one of the rounding modes while taking the digits g, r and the sticky bit into consideration

- most common: binary digits (bits) and round-to-even

Example:

$$\begin{array}{rl} & 3.2100 \times 10^2 \\ & 0.0084 \times 10^2 \\ \text{----------} & \\ & 3.2184 \times 10^2 \end{array}$$

Rounded result: 3.22×10^2

Example 1: 32 - 2.25 = 30 using 4 significands, base = 2

$1.000 * 2^5 - 1.001 * 2^1$

"Infinite" internal precision

```
 1.000 0000 * 2^5
-0.000 1001 * 2^5   align, keep all the bits
 0.111 0111 * 2^5   precise result (= 29.75)
 1.110 1110 * 2^4   normalized
 1.111      * 2^4   round up (= 30)
```

Using g(uard), r(ound), s(ticky) bits

plus 4 significands

```
 1.000 000 * 2^5
-0.000 101 * 2^5   align, drop bit, set sticky
 0.111 011 * 2^5
 1.110 11  * 2^4   normalized
 1.111     * 2^4   round up (= 30)
```

Example 2: 32 - 3.75 = 28 using 4 significands , base = 2

$1.000 * 2^5 - 1.111 * 2^1$

"Infinite" internal precision

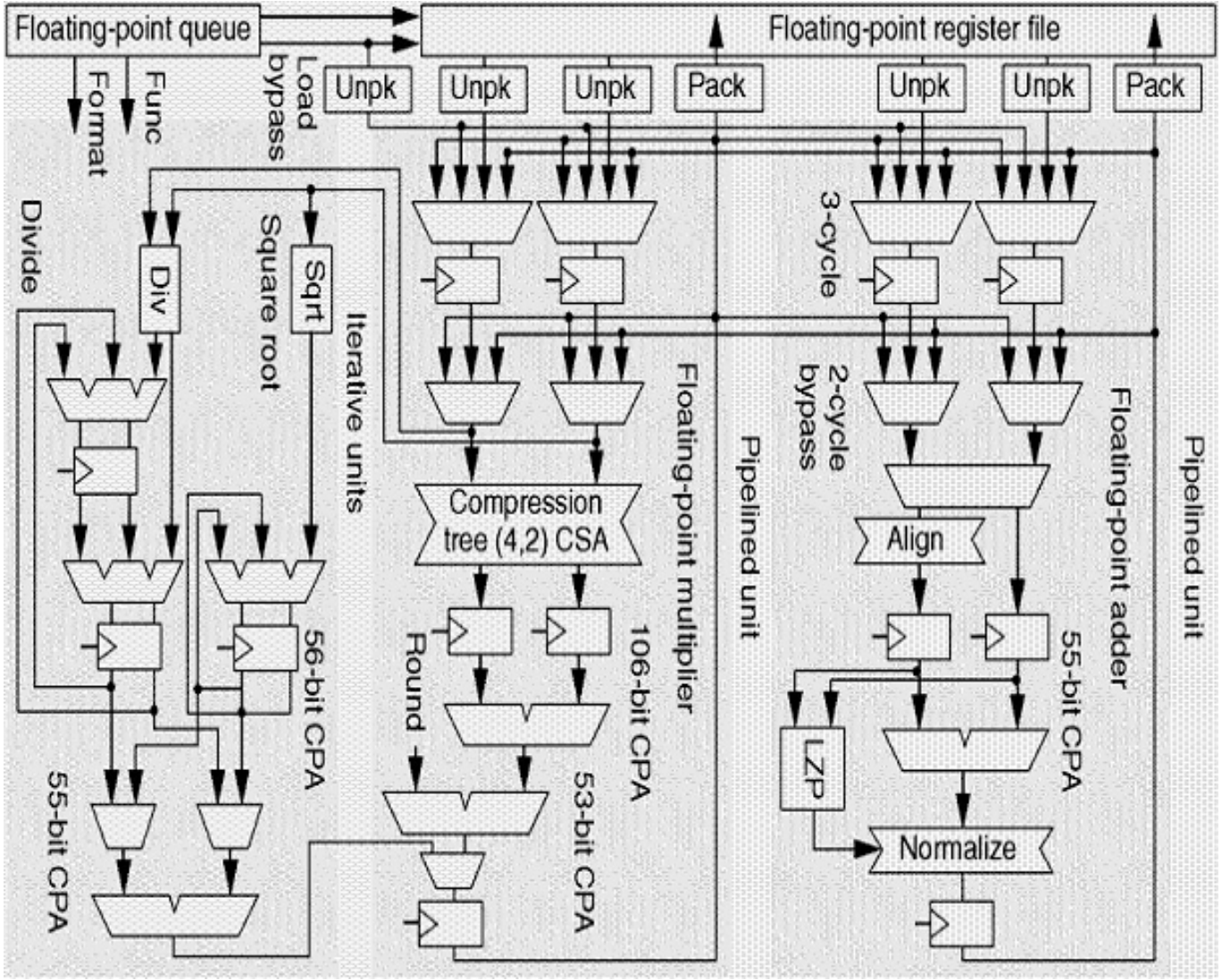
```
 1.000 0000 * 2^5
-0.000 1111 * 2^5   align, keep all the bits
 0.111 0001 * 2^5   precise result (= 28.25)
 1.110 0010 * 2^4   normalized
 1.110      * 2^4   round down (= 28)
```

Using g(uard), r(ound), s(ticky) bits

plus 4 significands

```
 1.000 000 * 2^5
-0.000 111 * 2^5   align, drop bit, set sticky
 0.111 001 * 2^5
 1.110 01  * 2^4   normalized
 1.110     * 2^4   round down (= 28)
```

MIPS R10000 Floating Point Unit



See literature for multiplication and division!

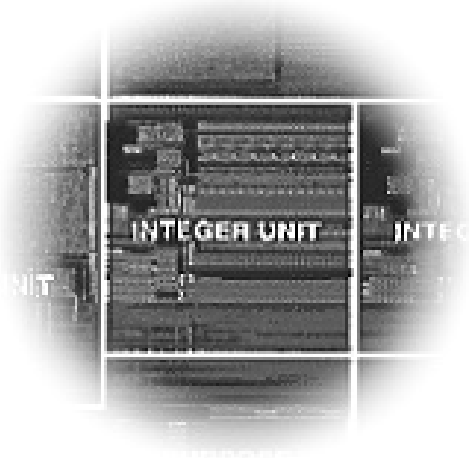
ARITHMETIC LOGIC UNIT (ALU)

Arithmetic logic unit

ALU (arithmetic logic unit):
- Part of the execution unit of a processor
- Performs logic and arithmetic operations

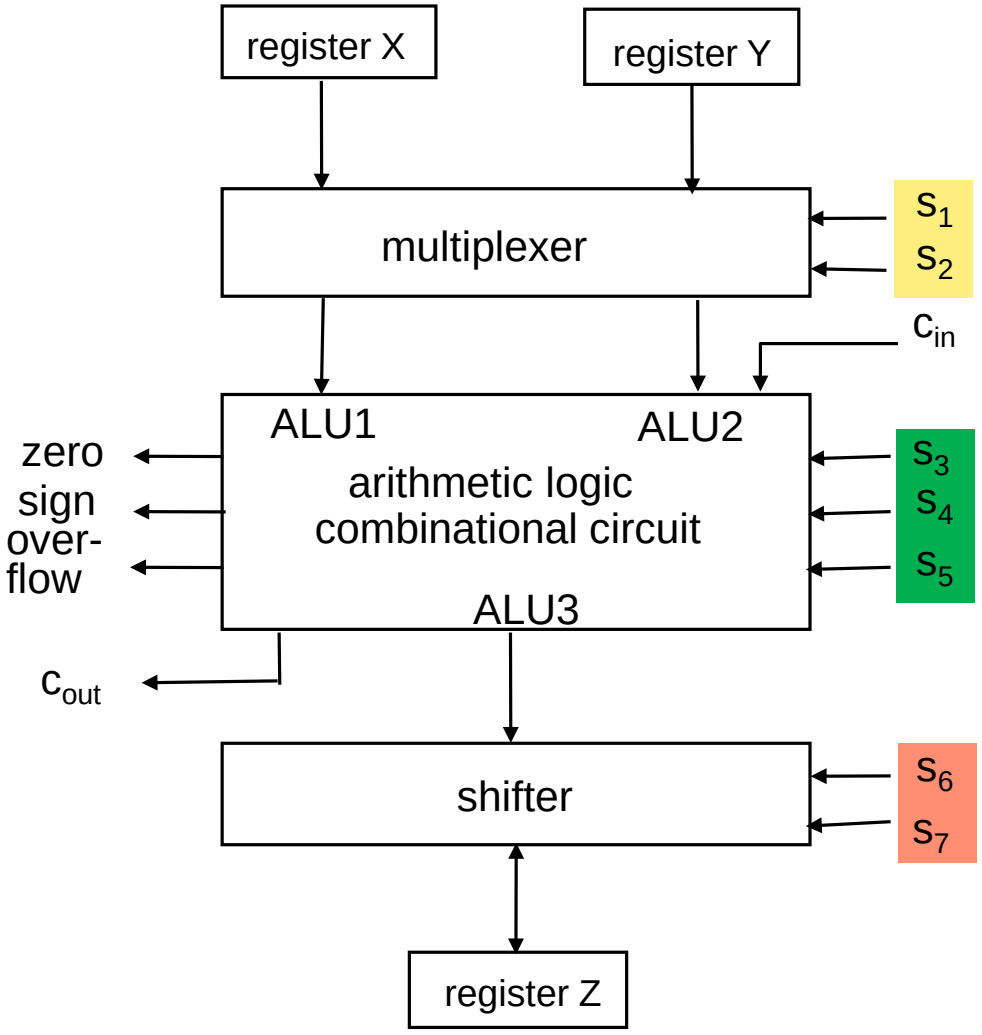
Inputs of an ALU:
- Operands and control signals of the control unit

Outputs of an ALU:
- Results and status signals to the control unit



Quite often ALUs in simple processors can operate on integers only. Floating point operations are off-loaded to an FPU (floating point unit) or emulated in software as a sequence of fixed point operations.

Diagram of a simple ALU



s_1	s_2	ALU1	ALU2
0	0	X	Y
0	1	X	0
1	0	Y	0
1	1	Y	X

s_3	s_4	s_5	ALU3
0	0	0	ALU1 + ALU2 + c_{in}
0	0	1	ALU1 – ALU2 – Not(c_{in})
0	1	0	ALU2 – ALU1 – Not(c_{in})
0	1	1	ALU1 $\vee$ ALU2
1	0	0	ALU1 $\wedge$ ALU2
1	0	1	Not(ALU1) $\wedge$ ALU2
1	1	0	ALU1 $\oplus$ ALU2
1	1	1	ALU1 $\leftrightarrow$ ALU2

s_6	s_7	Z
0	0	ALU3
0	1	ALU3 ÷ 2
1	0	ALU3 × 2
1	1	store Z

Example operations of the ALU

Left shift of the ones' complement of Y stored in Z:

- Control signals: $s_1 \ldots s_7 = 10\ 111\ 10$
  - 10 : ALU1 = Y
  - 111: ALU3 = ALU1 ↔ ALU2
  - 10 : Z = ALU3 × 2

Is $X > Y$?

- Check status signal "sign" after the operation $Y - X$.
- Control signals: $s_1 \ldots s_7 = 00\ 010\ 00$ and $c_{in} = 1$
  - 00 : ALU1 = X und ALU2 = Y
  - 010: ALU3 = ALU2 – ALU1 – not(c_{in})
  - 00 : Z = ALU3

Questions & Tasks

- When do we set the sticky bit?
- Why is it enough to store a single sticky bit instead of a value?
- Does the ALU support loops, if-then-else etc.?

Overview

Numbering systems
- Positive numbers
- Negative numbers
- Number conversion
- Real numbers (fixed/floating point)
  - Real systems use IEEE!
- Rounding of numbers

Simple circuits
- Addition / Subtraction
- See literature for Multiplication / Division

Arithmetic logic unit (ALU)
- Components of a simple ALU