DER COMPUTER UND DIE ZAHLEN

WIE EIN COMPUTER POSITIVE, NEGATIVE UND GLEITKOMMAZAHLEN DARSTELLT

GANZE POSITIVE ZAHLEN

WAS WIR SCHON GELERNT HABEN:

- Zahlen in der Münzendarstellung und in der Stellenwertdarstellung
- Zählen und Addieren im 2-adischen Zahlensystem („binäres Zahlensystem“)

WIE GROSS SIND EIGENTLICH DIE ZAHLEN, MIT DENEN EIN PROZESSOR RECHNEN KANN?

Die Antwort auf diese Frage hängt, wie du dir wahrscheinlich vorstellen kannst, davon ab, wie der Prozessor aufgebaut ist, mit anderen Worten: welche **Architektur** der Prozessor hat. Konkret geht es um die Frage, wie gross seine Speicherzellen („Register“) sind. **16 Bits** waren es bis in die frühen 1990er Jahre, **32 Bits** waren es bis zum Anfang der 2000er Jahre und zu Beginn der 2000er-Jahre kam die **64-Bit**-Architektur auf, die heute Standard ist.

Dementsprechend gross sind auch die Zahlen, die ein Prozessor **in einem Arbeitsschritt** verarbeiten kann, die er also z.B. addieren kann. Ohne zusätzliche Programme für die Darstellung grösserer Zahlen ist im Prozessor die Anzahl sowie die Grösse der Zahlen, die verarbeitet werden können, begrenzt.

Wie viele **unterschiedliche Zahlen** sind das nun, wenn wir davon ausgehen, dass jede Bitfolge genau eine Zahl darstellt? Das siehst du in der folgenden Tabelle.

Architektur	Anzahl der ganzen Zahlen		Datentyp
	in Potenzschreibweise	als Dezimalzahl	
16 Bits		65'536	**`short`**
32 Bits		4'294'967'296	**`int`**
64 Bits		1'844'674'473'709'551'616	**`long`**

Aufgabe 1: *Ergänze „zum Aufwärmen“ in dieser Tabelle die Werte in der Spalte „Anzahl der ganzen Zahlen in Potenzschreibweise“ (also in der Form a^n)!*

Den jeweiligen Längen der binären Zahlen sind in vielen Programmiersprachen bestimmte **Datentypen** zugeordnet (z.B. `short` für ganze Zahlen der Länge 16 Bit, `int` für ganze Zahlen der Länge 32 Bit, `long` für ganze Zahlen der Länge 64 Bit). Der weitaus am häufigsten verwendete Datentyp ist **`int`**, steht für „Integer“ (integer (engl.) = ganze Zahl (dt.)). Der Grund dafür ist, dass ganze Zahlen, die über die Milliardengrenze hinausgehen, verhältnismässig selten gebraucht werden und 32 Bit daher fast immer ausreichen.

ADDITION VON GANZEN ZAHLEN IN EINEM PROZESSOR

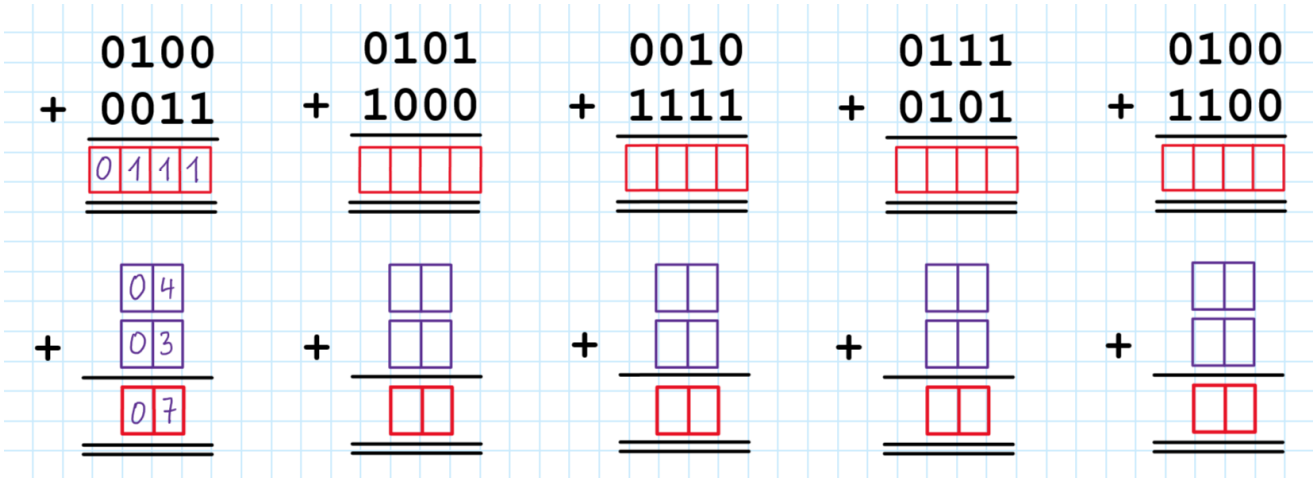
Der Einfachheit halber stellen wir uns nun einen Prozessor vor, der lediglich eine 4-Bit-Architektur hat, d.h., die Zahlen, die er verarbeiten kann, haben **genau** 4 Bits. (Jede Speicherzelle unseres Prozessors hat genau 4 Bits, daher wird jede Zahl mit 4 Bits dargestellt, d.h., dass auch führende Nullen gespeichert werden.)

Bei einer Addition addiert unser Prozessor nun Schritt für Schritt von dem Bit mit dem kleinsten Wert beginnend (in unserer Schreibweise also von rechts beginnend) jedes Bit, macht – falls nötig – einen Übertrag und addiert weiter. Er macht also z.B. sowohl bei der Addition von `0000` und `0001` als auch bei der Addition von `1111` und `1110` genau vier Berechnungsschritte und speichert eine Zahl mit genau 4 Bits als Ergebnis ab.[1]

Aufgabe 2:

a. *Simuliere nun als „Fingerübung" die Arbeit unseres 4-Bit-Prozessors und führe die hier dargestellten Additionen durch!*
Aber Achtung: Nicht alle Additionen werden ein sinnvolles Ergebnis liefern!

b. *Übertrage danach die vierstelligen Binärzahlen in zweistellige Dezimalzahlen und trage diese in die direkt darunter stehenden Additionen ein! Markiere anschliessend diejenigen Rechnungen, die kein sinnvolles Ergebnis liefern!*



Offensichtlich gibt es Additionen in einem Prozessor, bei denen er im Vergleich mit dem ersten Summanden vor der Addition als Ergebnis eine kleinere Zahl liefert. Das ist einerseits natürlich alles andere als ideal. Andererseits können wir diese Gegebenheit auch nutzen, um eine Subtraktion „getarnt" als Addition in unserem Prozessor durchzuführen und so auch eine Darstellung für negative ganze Zahlen zu finden.

[1] Aufgrund dieser Tatsache wird deutlich, dass es durchaus sinnvoll ist, in Programmen die Grösse von Zahlen durch die Verwendung z.B. des Datentyps `int` auf 32 Bits zu beschränken, damit ein Prozessor mit einer 64-Bit-Architektur etwa bei einer Addition darauf verzichten kann, die ersten 32 führenden Nullen zu berechnen, weshalb er in der Hälfte der Zeit zu einem Ergebnis kommt.

NEGATIVE GANZE ZAHLEN

Wie stellt man nun im Computer negative ganze Zahlen dar? Ein **Teil der Antwort** auf diese Frage lautet: **Man reserviert das erste (das am weitesten links stehende) Bit für das Vorzeichen**; steht hier eine **0**, so handelt es sich um eine positive Zahl (**+**), steht hier eine **1**, so handelt es sich um eine negative Zahl (**–**).

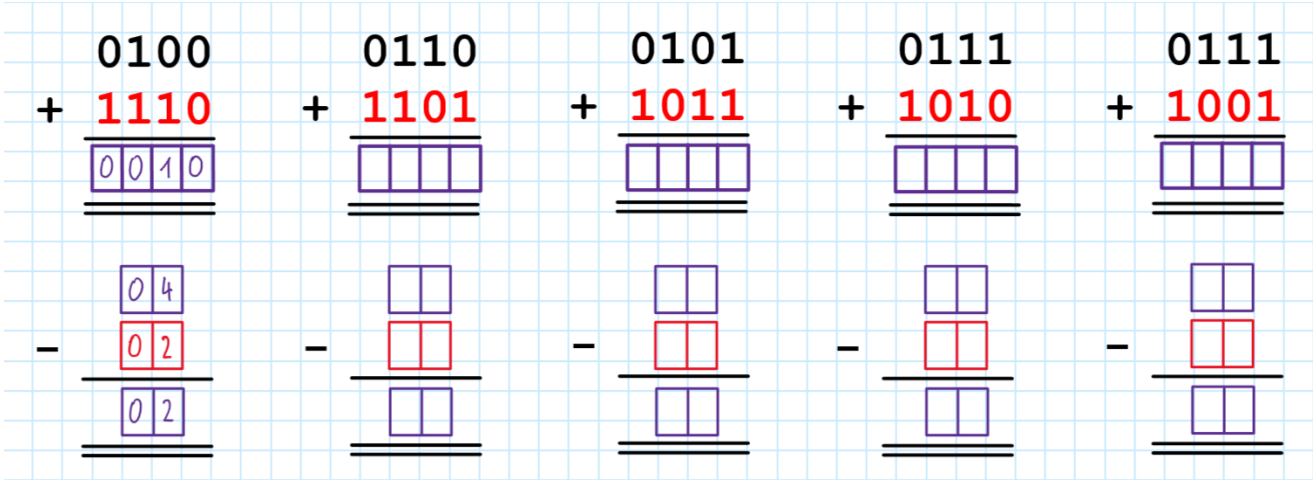
Die Frage, die sich daraus ergibt, lautet: Wie müssen die restlichen Bits aussehen, damit aus einer Zahl `b` (die mit 0 beginnt) die Zahl `–b` (die mit 1 beginnt) wird, sodass ein Addierwerk („Addition Gatter") in einem Prozessor die Subtraktion `a – b` als Addition von `a + (–b)` berechnen kann und dabei das richtige Resultat herauskommt?

Aufgabe 3:

a. Führe die hier dargestellten Additionen einer positiven (schwarz gedruckt) und einer negativen Binärzahl (rot gedruckt) wie in Aufgabe 2a durch!

b. Übertrage danach den ersten Summanden (schwarze Binärzahl) und das Ergebnis (blaue Kästchen) in Dezimalzahlen und trage diese Zahlen in die direkt darunter stehenden Additionen ein (blaue Kästchen)! Berechne anschliessend, welche Dezimalzahl die negativen (roten) Binärzahlen jeweils darstellen (rote Kästchen)!

c. Trage die negativen (roten) Binärzahlen in die unten angefügte Tabelle ein! Kannst du ein Muster erkennen? Versuche Regeln zu beschreiben, nach denen in diesem 4-Bit-Prozessor negative Zahlen gebildet werden!



Dezimalzahl	entsprechende Binärzahl		entsprechende Binärzahl
–1		**–5**	
–2	1110	**–6**	
–3		**–7**	
–4		**–8**	

Wenn du meinst, die Regeln für die Bildung von negativen Zahlen gefunden zu haben, melde dich bei deiner Lehrperson und erkläre ihr deinen Lösungsansatz!

Aufgabe 4: *Besprich mit einer Mitschülerin oder einem Mitschüler folgende Fragen!*

Wann kommt es bei einer Addition von **zwei positiven Binärzahlen** (die mit einer 0 beginnen) dazu, dass eine **negative Binärzahl** (die mit einer 1 beginnt) **als Ergebnis** herauskommt?
Wie würden die kleinsten/grössten **negativen** Binärzahlen in einer Tabelle bei einem Prozessor mit 8-Bit-Architektur aussehen? Wie sehen sie bei einer 16-Bit-Architektur aus?
Wie sieht die **kleinste** darstellbare negative Binärzahl in den verschiedenen Prozessorarchitekturen aus? Wie sieht die Dezimalzahl **-1** jeweils als Binärzahl aus?
Wie könnte ein Algorithmus aussehen, um aus einer positiven Binärzahl eine negative Binärzahl zu machen, also praktisch einen **Vorzeichenwechsel** von + zu - vorzunehmen?

DAS ZWEIERKOMPLEMENT: DER TRICK ZUR DARSTELLUNG VON NEGATIVEN ZAHLEN

Um positive von negativen Binärzahlen zu unterscheiden, hat man die Vereinbarung getroffen, dass bei einer positiven Zahl das **am weitesten links stehende Bit** eine **Null** und bei einer negativen Zahl das am weitesten links stehende Bit eine **Eins** ist.

Um eine **positive Binärzahl n** zur entsprechenden **negativen Binärzahl -n** (oder umgekehrt eine negative Binärzahl -n zur positiven Binärzahl n) umzuwandeln, muss ein Prozessor nur den folgenden sehr einfachen Algorithmus anwenden – unabhängig davon, nach welcher Architektur er gebaut ist.

ALGORITHMUS „ZWEIERKOMPLEMENT“

1. **Invertiere alle Bits!**
   Aus jeder Null wird eine Eins, aus jeder Eins wird eine Null. (→ „Einerkomplement“)
2. **Addiere zu dem Ergebnis eine 1!**
   (→ „Zweierkomplement“)

Mit diesem „Trick“ kann der Prozessor die arithmetische Operation „Subtraktion“ auf die Operation „Addition“ reduzieren. Weil wir schon gelernt haben, dass die Multiplikation nur eine verkürzte Schreibweise von mehreren Additionen ist und in weiterer Folge Potenzen nur die verkürzte Schreibweise von mehreren Multiplikationen sind, und weil auch die Division auf die Subtraktion zurückgeführt werden kann, wissen wir nun, dass **die Addition die einzige arithmetische Operation** ist, die wir unbedingt benötigen, um einen Prozessor zu bauen.

GLEITKOMMAZAHLEN

Bis jetzt haben wir uns nur über ganze Zahlen Gedanken gemacht. Leider sind nicht alle Zahlen ganze Zahlen. Das Gegenteil ist der Fall: Zwischen jeder ganzen Zahl und der ihr direkt nachfolgenden ganzen Zahl liegen unendlich viele nichtganze Zahlen.[2]

Beginnen wir ganz unkompliziert: In der goldenen Mitte zwischen den ganzen Dezimalzahlen 1 und 2 liegt die nichtganze Dezimalzahl 1.5. Wie können wir diese Zahl binär darstellen? In der goldenen Mitte zwischen 1.5 und 2 liegt 1.75. Wie können wir 1.75 binär darstellen? Wie können wir z.B. die Dezimalzahl -1234.567 binär darstellen?

WIE KANN MAN NICHTGANZE DEZIMALZAHLEN IN NICHTGANZE BINÄRZAHLEN UMWANDELN?

Als wir über die Stellenwertdarstellung von natürlichen Zahlen gesprochen haben, haben wir gesehen, dass jede Stelle einer Zahl $a_n a_{n-1} \ldots a_3 a_2 a_1 a_0$ eine Potenz der Basis des Zahlensystems darstellt. Für das Dezimalsystem ergeben sich somit folgende Werte für die einzelnen Stellen:

a_n	a_{n-1}	…	a_3	a_2	a_1	a_0
…			1000	100	10	1
10^n	10^{n-1}	…	10^3	10^2	10^1	10^0

[2] Warum ist das eigentlich so?

Ähnliches gilt nun für die Stellen hinter dem Komma. Auch hier stellt jede Stelle einer Zahl $a.b_1b_2b_3\ldots b_nb_{n+1}\ldots$ eine Potenz der Basis des Zahlensystems dar. Für das Dezimalsystem sind dies folgende Werte:

a			.	b_1	b_2	b_3	…	b_n	b_{n+1}	…
…	10	1	.	0.1	0.01	0.001	…			
…	10^1	10^0	.	10^{-1}	10^{-2}	10^{-3}	…	b^{-n}	$b^{-(n+1)}$	…

Für das Binärsystem besitzen die Stellen hinter dem Komma die Werte:

$2^{-1} = \frac{1}{2^1} = \frac{1}{2} = 0.5$	$2^{-2} = \frac{1}{2^2} = \frac{1}{4} = 0.25$	$2^{-3} = \frac{1}{2^3} = \frac{1}{8} = 0.125$	$2^{-4} = \frac{1}{2^4} = \frac{1}{16} = 0.0625$	…

Mit diesen Informationen ist es uns möglich, dezimale Kommazahlen in binäre Kommazahlen umzuwandeln und umgekehrt, wie folgende Beispiele zeigen. Wir nutzen dazu die **Greedy-Methode**, wie wir sie von der Münzendarstellung von natürlichen Zahlen her kennen.

$$
\begin{aligned}
10.10101_2 &= 1\cdot 2^1 + 0\cdot 2^0 + 1\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + 0\cdot 2^{-4} + 1\cdot 2^{-5} \\
&= 2 + 0 + 0.5 + 0 + 0.125 + 0 + 0.03125 \\
&= 2.65625_{10}
\end{aligned}
$$

$$
\begin{aligned}
3.1875_{10} &= \mathbf{1}\cdot 2^1 + 1.1875 \\
&= 1\cdot 2^1 + \mathbf{1}\cdot 2^0 + 0.1875 \\
&= 1\cdot 2^1 + 1\cdot 2^0 + \mathbf{0}\cdot 2^{-1} + 0.1875 \\
&= 1\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + \mathbf{0}\cdot 2^{-2} + 0.1875 \\
&= 1\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 0\cdot 2^{-2} + \mathbf{1}\cdot 2^{-3} + 0.0625 \\
&= 1\cdot 2^1 + 1\cdot 2^0 + 0\cdot 2^{-1} + 0\cdot 2^{-2} + 1\cdot 2^{-3} + \mathbf{1}\cdot 2^{-4} + 0 \\
&= 11.0011_2
\end{aligned}
$$

Aufgabe 5: *a. Wandle die binären Kommazahlen 101.111_2, 110.011_2 und 1001.1001_2 in dezimale Kommazahlen um!*

b. Wandle die dezimalen Kommazahlen 13.78125_{10}, 1.90625_{10} und 0.1_{10} in binäre Kommazahlen um! (Sechs Stellen hinter dem Komma sind genug …)

Bei den negativen ganzen Zahlen haben wir gesehen, dass das am weitesten links stehende Bit dafür reserviert ist, die Information zu speichern, ob es sich um eine positive oder um eine negative Zahl handelt. Das gilt auch für negative nichtganze Zahlen. Vielleicht könnte man einige andere Bits ja dafür reservieren, um Informationen über die Ziffern hinter dem Komma zu speichern …? Wieso ist in der Zwischenüberschrift auf der letzten Seite von „**Gleit**kommazahlen" die Rede …?

Aufgabe 6: *Bestimme und beschreibe die Informationen, die man benötigt, um die nichtganzen Zahlen 1.5, 1.75 und -1234.567 binär abspeichern zu können!*

Wenn du meinst, die notwendigen Informationen für die Speicherung von Kommazahlen gefunden zu haben, melde dich bei deiner Lehrperson und erkläre ihr deinen Lösungsansatz!

DIE ANNÄHERNDE DARSTELLUNG VON NICHTGANZEN SOWIE VON SEHR GROSSEN ZAHLEN

Wir suchen also eine Darstellung von nichtganzen Zahlen innerhalb der Grösse der Speicherzellen („Register"), die uns die Architektur unseres Prozessors vorgibt. Überdies wäre es schön, wenn wir auch viel grössere (positive) und viel kleinere (negative) Zahlen speichern könnten, als es uns die Registergrösse erlaubt. Tatsächlich können wir diese „zwei Fliegen mit einer Klatsche schlagen".

Allerdings hat es **zwei Folgen**, wenn wir das tun: Wir müssen erstens einige Bits für einen Exponenten reservieren, um durch ihn die Grössenordnung der Zahl zu bestimmen; wir müssen zweitens in Kauf nehmen, dass wir die Zahlen nicht mehr genau, sondern nur ungefähr bzw. annähernd genau darstellen können, weil wir durch diesen Trick ja viel mehr Zahlen darstellen können, als es uns die Anzahl der zur Verfügung stehenden Bitfolgen (= alle möglichen Registerinhalte) erlaubt.

Um derartige Zahlen abzuspeichern, benötigen wir zusammengefasst genau drei Informationen:

1. Das **Vorzeichen**: Es bestimmt, ob es sich um eine positive oder um eine negative Zahl handelt.
2. Der **Exponent**: Er bestimmt, an welcher Stelle der Zahl sich das Komma befindet, bzw. bei sehr grossen Zahlen, wie viele Nullen (also: Stellen) den abgespeicherten Ziffern noch angehängt werden müssen.
3. Die **Mantisse**: Sie bestimmt die abgespeicherten ersten n Ziffern der darzustellenden Zahl.

Damit **eindeutig** bestimmt werden kann, wo sich das Komma befindet, hat man sich auf eine **normalisierte Darstellung** geeinigt. Mithilfe der Wahl des Exponenten lässt man das Komma durch die Zahl „gleiten", sodass es sich in der dargestellten Zahl **immer nach der ersten Ziffer** der Mantisse befindet.[3]

ZWEI BEISPIELE ZUR VERANSCHAULICHUNG

Sehen wir uns gleich ein Beispiel an, um etwas deutlicher zu machen, was mit diesen drei Informationen gemeint ist. Der Einfachheit halber und für eine bessere Anschaulichkeit stellen wir uns einen fiktiven Prozessor vor, der mit Dezimalzahlen rechnen kann. Seine Speicherzellen haben Platz für genau 6 Zeichen. Wir reservieren nun:

- das **erste** Zeichen **A** für das **Vorzeichen** (**+** oder **–**)
- das **zweite** Zeichen **B** für das **Vorzeichen des Exponenten** (**+** oder **–**)
- das **dritte** Zeichen **C** für den **Exponenten**
- das **vierte, fünfte und sechste** Zeichen **D**, **E** und **F** für die **Mantisse**

Somit können wir mithilfe von 6 Zeichen z.B. folgende Zahlen darstellen:

ABCDEF	**=**	**A BC DEF**	**=**	**A D EF**		**BC**	
012123	**=**	**+ −2 123**	**=**	**+ 1.23 · 10^{-2}**	**=**	**0.0123**
011234	**=**	**+ −1 234**	**=**	**+ 2.34 · 10^{-1}**	**=**	**0.234**
000345	**=**	**+ +0 345**	**=**	**+ 3.45 · 10^{0}**	**=**	**3.45**
001456	**=**	**+ +1 456**	**=**	**+ 4.56 · 10^{1}**	**=**	**45.6**
002567	**=**	**+ +2 567**	**=**	**+ 5.67 · 10^{2}**	**=**	**567**
114678	**=**	**− −4 678**	**=**	**− 6.78 · 10^{-4}**	**=**	**− 0.000678**
012321	**=**	**+ −2 321**	**=**	**+ 3.21 · 10^{-2}**	**=**	**0.0321**
107234	**=**	**− +7 234**	**=**	**− 2.34 · 10^{7}**	**=**	**− 23'400'000**

[3] Daher kommt auch der Name „Gleitkommazahl".

Welches sind nun die grössten und kleinsten positiven Zahlen?

```
009999   =   + +9 999   =   + 9.99 · 10⁹    =   9'990'000'000
019001   =   + -9 100   =   + 1.00 · 10⁻⁹   =   0.000000001
```

Die kleinste negative Zahl wäre dementsprechend – 9.99 · 10^9 = - 9'990'000'000 und die grösste negative Zahl – 1.00 * 10^{-9} = - 0.000000001.

Das ist doch schon einmal gar nicht so schlecht. Wir können mit lediglich 6 Ziffern bis zu zehnstellige ganze Zahlen und nichtganze Zahlen mit bis zu 11 Stellen hinter dem Komma darstellen. Aber richtig sensationell ist es auch nicht. Wenn wir aber die Speicherzellen unseres fiktiven Prozessors lediglich um zwei Stellen auf 8 Zeichen erweitern (1 Zeichen für das Vorzeichen, 1 Zeichen für das Vorzeichen des Exponenten, 2 Zeichen für den Exponenten und vier Zeichen für die Mantisse), zeigt sich die Macht, die eine derartige Zahlendarstellung hat:

```
00101234 = + +10 1234 =  1.234 · 10¹⁰  =  12'340'000'000

00999999 = + +99 9999 =  9.999 · 10⁹⁹  =   9'999'000'000'000'000'000'
                                              000'000'000'000'000'000'
                                              000'000'000'000'000'000'
                                              000'000'000'000'000'000'
                                              000'000'000'000'000'000'
                                              000'000'000

01990001 = + -99 1000 =  1.000 · 10⁻⁹⁹  =  0.000000000000000000000000
                                           00000000000000000000000000
                                           00000000000000000000000000
                                           0000000000000000000000001
```

Wir halten fest:

Je mehr Zeichen wir für den Exponenten reservieren, umso grössere positive und kleinere negative Zahlen (positiver Exponent) bzw. umso mehr Stellen hinter dem Komma (negativer Exponent) können wir darstellen!

Das Problem allerdings ist die Genauigkeit dieser Zahlendarstellungen. Sie sind nicht exakt, sondern nur **Annäherungen**. Wenn für die Mantisse in unserem Beispiel nur vier Zeichen zur Verfügung stehen, dann werden alle Stellen ab der vierten Ziffer abgeschnitten. Wenn zusätzlich gerundet wird, werden alle Zahlen von `12'335'000'000` bis `12'344'999'999.999`... in der Darstellung `+ +10 1234` zusammengefasst. Die maximale Abweichung in dieser Beispielzahl beträgt also `5'000'000`. Das klingt nach viel, ist aber tatsächlich „nur" etwa `0.04 %` der dargestellten Zahl. Würden wir zwei zusätzliche Zeichen für die Mantisse reservieren, wäre die maximale Abweichung `50'000`. Das wären nur etwa `0.0004 %` der dargestellten Zahl.

Wir halten also fest:

Je mehr Zeichen wir für die Mantisse reservieren, umso genauer wird die annähernde Darstellung von nichtganzen bzw. sehr grossen Zahlen.

Die Kunst ist es nun, bei einer vorgegebenen Grösse der Register eines Prozessors (= Anzahl der Zeichen) ein gutes Verhältnis zwischen den Zeichen, die für den Exponenten reserviert sind, und den Zeichen, die für die Mantisse reserviert sind, zu finden.

DIE BINÄRE DARSTELLUNG VON GLEITKOMMAZAHLEN

Kommen wir nun wieder von den Dezimalzahlen zurück zu den binären Zahlen und von den fiktiven Prozessoren mit Speicherzellen von 6 oder 8 Zeichen zu den Architekturen realer Prozessoren mit 32 Bits oder 64 Bits.[4]

Hier ist das Verhältnis der jeweils zur Verfügung stehenden Bits folgendermassen:

Architektur	Vorzeichen	Exponent	Mantisse	Datentyp
32 Bits	1 Bit	8 Bits	23 Bits	**`float`**
64 Bits	1 Bit	11 Bits	52 Bits	**`double`**

Ein Beispiel einer binär dargestellten Gleitkommazahl mit 32 Bits:

```
ABCDEFGHIJKLMN...
00011001111010000000000000000000   =
```
→ So steht die Zahl in der Speicherzelle.

```
A BCDEFGHI JKLMN...
0 00110011 11010000000000000000000 =
```
→ So wird diese Binärfolge aufgeteilt.

```
+   51₁₀              1.625₁₀      =
```
→ Das sind die Teile in Dezimalzahlen umgerechnet.

```
+  1.625 · 2⁵¹                     ≈
```

```
+  3.65917 · 10¹⁵                  =
```
→ Diese Umrechnung hat Excel für mich gemacht. 😉

```
+  3'659'170'000'000'000
```

Wie du siehst, gibt es auch hier **Datentypen**, die mit den jeweiligen Bitlängen der Zahlendarstellungen zusammenhängen. Der Typ **`float`**, der 32 Bits verwendet, hat seinen Namen daher, dass neben „Gleitkommazahl“ auch der gleichbedeutende Ausdruck „Fliesskommazahl“ verwendet wird. Der Typ **`double`**, der 64 Bits verwendet, heisst deshalb so, weil damit die „doppelte Genauigkeit“ bezeichnet wird.

Aufgabe 7: *Beantworte die folgenden Fragen in der Tabelle auf der Rückseite dieses Blattes!*

a. *Berechne (eventuell im Kopf?), welches die jeweils höchsten und niedrigsten Exponenten sind, die bei den Datentypen* `float` *und* `double` *dargestellt werden können! Notiere sie als Dezimalzahlen und stelle sie auch als Bitfolge dar!*

b. *Stelle die grösstmögliche Zahl, die annähernd mit den Datentypen* `float` *und* `double` *dargestellt werden kann, in der binären Darstellung von Gleitkommazahlen dar und berechne (mit Taschenrechner oder Computer) diese Zahl als Dezimalzahl*

[4] 16-Bit-Architekturen können wir hier ausblenden, weil damit nur entweder nicht ausreichend grosse oder nicht ausreichend genaue Zahlen dargestellt werden können.

(dargestellt als dezimale Gleitkommazahl)!
Tipp: $1.11111111111\ldots_2$ kannst du aufrunden auf 10.0_2.

c. *Berechne (mit Taschenrechner oder Computer) die höchste maximale Abweichung (dargestellt als dezimale Gleitkommazahl) von der exakten Darstellung einer Zahl, wenn die Mantisse nach 23 Bits (`float`) oder nach 52 Bits (`double`) abgeschnitten wird!*

`float`	**`double`**
a. höchster und niedrigster Exponent	
b. grösste darstellbare Zahl (dargestellt als Binärfolge und als Dezimalzahl in der Form $n \cdot 10^m$)	
c. höchste maximale Abweichung von der exakten Darstellung einer Zahl (dargestellt als $n \cdot 10^m$)	