Dr. Robin Bergenthum

Grundlagen der Informatik I

Modul 65001

Leseprobe

Fakultät für
Mathematik und Informatik

FernUniversität in Hagen

Das Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung und des Nachdrucks, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Kein Teil des Werkes darf in irgendeiner Form (Druck, Fotokopie, Mikrofilm oder ein anderes Verfahren) ohne schriftliche Genehmigung der FernUniversität reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Inhaltsverzeichnis

1 Einführung in die Informatik

Hallo! Schön, dass Sie sich für das **Modul 65001 Grundlagen der Informatik I** entschieden haben. In diesem Lehrtext finden Sie eine Einführung in die **Technische** und die **Praktische Informatik**. Als Teil der Praktischen Informatik werden wir uns auch mit dem imperativen Programmieren beschäftigen. Insgesamt habe ich mir viel Mühe gegeben, Ihnen eine spannende Auswahl an Themen zusammenzustellen. Aufbauend auf die Inhalte dieses Moduls können Sie dann gerne auch den zweiten Teil dieser Einführung, das **Modul 65002 Grundlagen der Informatik II** von Sebastian Küpper, belegen. Dort werden Sie die Einführung in die Praktische Informatik vertiefen und sich mit dem Objektorientierten Programmieren auseinandersetzen, bevor Sie eine Einführung in die Theoretische Informatik erhalten.

Viel Spaß beim Lesen!

Robin Bergenthum

https://www.fernuni-hagen.de/mi/fakultaet/lehrende/bergenthum/index.shtml

1.1 Was ist Informatik?

Informatik ist die Wissenschaft von allem, was man mit einem Computer machen kann. Aus diesem Grund heißt Informatik im Englischen **Computer Science**, die Wissenschaft der Computer. Doch was machen wir mit Computern? Nehmen wir an, wir haben ein konkretes, reales Problem. Dieses Problem beschreiben wir in einer dem Computer lesbaren Form. Dann geben wir unser Problem als Eingabe an den Computer. Der Computer „computert“, er führt also einen Algorithmus aus, und wir erhalten eine Ausgabe. Diese Ausgabe beschreibt dann die Lösung unseres Problems. Damit das alles funktioniert, benötigen wir Informatik, um den Computer zu entwerfen, unser Problem zu formalisieren, den Algorithmus zu schreiben, und um das Ergebnis zu nutzen.

In der deutschen Sprache spricht man nicht von Computerwissenschaft oder Rechnerwissenschaft, sondern hat das Kunstwort **Informatik** geschaffen. Ähnlich wie bei Mathematik wird Informatik aus dem Wort Information abgeleitet und bezeichnet die Wissenschaft der Verarbeitung von Informationen. Dass diese Verarbeitung in der Regel mithilfe von Computern erfolgt, nehmen wir dabei einfach als gegeben an.

Wie in jedem Einführungstext zur Informatik üblich wollen wir zuerst, drei zentrale Grundbegriffe aus dem Umfeld der Informatik beschreiben: **Nachricht**, **Information** und **Datum**. Mit diesen Begriffen werden wir dann eine etwas präzisere Definition des Begriffs Informatik versuchen.

Eine **Nachricht** ist eine vom konkreten Medium abstrahierte, endliche Folge von Symbolen zur Speicherung und Übertragung von Informationen. Dabei folgt jede Nachricht einer bestimmten Syntax. Die **Syntax** ist das Regelsystem, nach dem wir einzelne Zeichen und Symbole zu einer Nachricht zusammensetzen dürfen. Eine **Information** ist die einer Nachricht zugeordnete Bedeutung. Die Nachricht folgt einer Syntax, die Bedeutung wird jedoch über die Semantik definiert. **Semantik** ist die Wissenschaft der Bedeutungslehre. Ein **Datum** ist ein Paar aus **Nachricht** und zugehöriger **Information**. Der Plural von Datum ist **Daten**. Innerhalb der Informatik geben wir Informationen als Daten in einen Computer ein. Auch die Ausgabe des Computers sind Daten. Dem Ergebnis können wir dann die gewünschten Informationen entnehmen und das Ergebnis nutzen. Durch die Verarbeitung und Analysen von Daten generieren wir also Lösungen und sind nach Syntax und Semantik, bei der **Pragmatik** angekommen.

Das Positionspapier: *Was ist Informatik?* der *Gesellschaft für Informatik* (https://gi.de/) definiert Informatik wie folgt.

Informatik ist die Wissenschaft von der systematischen Darstellung, Speicherung, Verarbeitung und Übertragung von **Daten**, besonders der automatischen Verarbeitung mithilfe von Computern. **Informatik** ist zugleich Grundlagenwissenschaft, Formalwissenschaft als auch Ingenieursdisziplin.

[von Gesellschaft für Informatik e.V. (GI), https://gi.de/fileadmin/GI/Hauptseite/Themen/was-ist-informatik-lang.pdf]

Damit ist die Informatik ein sehr breites Feld. Aus diesem Grund gliedert sich die Informatik mittlerweile in eine Vielzahl von spezialisierten Teilgebieten.

Die **Theoretische Informatik** ist ein Teilgebiet der Informatik, das sich mit der formalen Untersuchung von Algorithmen, Berechnungen und Problemlösungen befasst. Sie konzentriert sich auf die grundlegenden Konzepte und Prinzipien, die der Informatik zugrunde liegen, und untersucht abstrakte Modelle von Berechnungen und deren Eigenschaften. Zu den zentralen Fragen der Theoretischen Informatik gehören: Wie formalisiere ich Probleme, die mein Computer lösen soll? Wie schwer ist ein solches Problem? Gibt es verschiedene Klassen von Problemen, die ähnlich schwer sind? Was ist das abstrakte Modell eines Computers? Welche Klassen von Problemen kann ein Computer lösen? Wie viel Speicher oder Zeit braucht ein Computer für das Lösen verschiedene Probleme innerhalb verschiedener Problemklassen? Wann sind Algorithmen für Probleme effizient und korrekt? Damit ist die Theoretische Informatik die Grundlage für viele andere Teilgebiete der Informatik und trägt wesentlich zum Verständnis der Grenzen und Möglichkeiten von Computern bei. Sie werden die Theoretische Informatik nicht in diesem Kurs, aber im nächsten Semester, im **Modul 65002 Grundlagen der Informatik II** von Sebastian Küpper, kennen lernen.

Die **Technische Informatik** ist ein Teilgebiet der Informatik, das sich mit der Entwicklung und dem Entwurf von Komponenten und Bausteinen moderner Computer befasst. Im Gegensatz zur Theoretischen Informatik, die sich auf abstrakte Konzepte und Algorithmen konzentriert, beschäftigt sich die Technische Informatik mit der Hardwareseite von Computersystemen. Zu den zentralen Fragen der Technischen Informatik gehören: Wie realisiere und optimiere ich Logikschaltungen auf Hardwareebene? Was ist die grundlegende Architektur eines Computers? Wie baue und verbinde ich Speicher und Prozessoren? Wie kommunizieren Hardwarekomponenten, um eine

effiziente Interaktion zu ermöglichen? Damit bildet die Technische Informatik eine Brücke zwischen der Theoretischen Informatik und der praktischen Anwendung eines Computers. Ziel der Technischen Informatik ist es, Computer zu optimieren und ihre Leistung, Zuverlässigkeit und Energieeffizienz zu erhöhen. Wir werden die Grundlagen der Technischen Informatik in den Kapiteln 2, 3 und 4 dieses Lehrtextes kennen lernen.

Die **Praktische Informatik** ist ein Teilgebiet der Informatik, das sich mit der Anwendung von theoretischen Konzepten und Methoden auf konkrete Probleme und Anwendungen befasst. Im Gegensatz zur Theoretischen Informatik und zur Technischen Informatik umfasst die Praktische Informatik die Entwicklung, Implementierung und Anwendung von Software und Systemen. Zu den zentralen Fragen der Technischen Informatik gehören: Wie entwerfe, analysiere, implementiere und teste ich Software? Was sind geeignete Prinzipien, Methoden und Werkzeuge, um Software hochwertig und effizient zu entwickeln? Wie unterscheidet sich Software in verschiedenen Anwendungsbereichen wie Desktopanwendungen, Webanwendungen, mobile Anwendungen, Datenbankanwendungen? Damit verbindet die Praktische Informatik die Konzepte der theoretischen und der Technischen Informatik, um Lösungen zu entwickeln und umzusetzen. In diesem Modul werden wir uns in den Lektionen 5, 6 und 7 damit befassen, Algorithmen zu entwerfen und zu implementieren.

Die theoretische, Technische und Praktische Informatik bilden zusammen die sogenannte **Kerninformatik** – die eigentliche Wissenschaft des Computers. Im Gegensatz dazu befasst sich die Angewandte Informatik mit den möglichen Anwendungen der Informatik.

Damit ist die **Angewandte Informatik** ein Sammelbegriff für viele anwendungsbezogene Teildisziplinen. Bestes Beispiel innerhalb dieses Moduls ist für uns wohl die Wirtschaftsinformatik. Die **Wirtschaftsinformatik** ist ein interdisziplinäres Fachgebiet, das die Bereiche Informatik und Betriebswirtschaftslehre miteinander verbindet. Es beschäftigt sich mit der Anwendung von Informatik zur Unterstützung von Geschäftsprozessen, zur Optimierung von betrieblichen Abläufen und zur Erreichung von Unternehmenszielen. Studierende der Wirtschaftsinformatik lernen sowohl betriebswirtschaftliche als auch informationstechnologische Konzepte, um komplexe Geschäftsprobleme zu lösen. Neben der Wirtschaftsinformatik gibt es noch viele andere Teildisziplinen der Angewandten Informatik, die sich jeweils mit spezifischen Aspekten der praktischen Anwendung befassen. Die Liste ist lang, einige der prominenteren Teilbereiche sind sicherlich die Medizinische Informatik, Robotik, Geoinformatik, Simulation und Modellierung, Datenbankmanagement, Netzwerktechnik, Human-Computer Interaction, IT-Sicherheit, Künstliche Intelligenz, Spieleentwicklung, E-Learning und Bildungstechnologie und viele andere mehr.

Neben den Teilbereichen theoretische, Technische, Praktische und Angewandte Informatik existieren mittlerweile weitere Teilbereiche, wie zum Beispiel **Informatik und Gesellschaft** und die **Didaktik der Informatik**. Gerne können sie sich auf der Seite der *Gesellschaft für Informatik* unter den verschiedenen Fachbereichen und Fachgruppen einmal umsehen (https://gi.de/netzwerk/fachbereiche).

1.2 Was ist ein Computer?

Wir wollen uns jetzt kurz mit der Geschichte und den Anfängen der Informatik beschäftigen. Das gehört einfach zu jedem Modul über die Grundlagen der Informatik dazu. Insgesamt hilft uns der Einblick dabei, die verschiedenen Fragestellungen innerhalb der Informatik besser zu verstehen.

Die Informatik ist im Vergleich zu anderen Wissenschaften eher jung und beginnt ihre rasche Entwicklung erst mit der Erfindung des Computers. Für dieses Modul gehen wir trotzdem erst einmal viel weiter zurück, denn Informatiker beginnen das Zählen gerne bei der Null. Im Jahr 0 herrschte in Europa das römische Reich und somit war während dieser Zeitperiode die Verwendung der römischen Zahlen weit verbreitet. Nach dem Zusammenbruch des römischen Reiches wurden sie jedoch allmählich von arabischen Ziffern verdrängt. Arabische Zahlen wurden durch den Handel mit arabischen Ländern und die Übernahme von mathematischen und wissenschaftlichen Werken aus dem Nahen Osten bekannt. Sie wurden im Laufe der Zeit immer populärer, da sie effizienter und einfacher zu verwenden waren als römische Zahlen. Die Verwendung arabischer Zahlen in Europa begann im Mittelalter, insbesondere im 10. Jahrhundert. Bis zum 15. Jahrhundert hatten arabische Zahlen die römischen Zahlen in Europa weitgehend verdrängt. Heute sind arabische Zahlen die Standardzahlen in fast allen Teilen der Welt.

Der Grund dafür, dass die arabischen Zahlen die römischen Zahlen abgelöst haben, liegt einfach daran, dass man mit arabischen Zahlen viel besser „rechnen“ kann.

Die römischen Zahlen bestehen aus den Symbolen I, V, X, L, C, D und M. Diese Symbole lassen sich gut in Stein meißeln. Durch Folgen dieser Symbole können wir jetzt Zahlen darstellen. So ist zum Beispiel die Folge von Symbolen II die Zahl 2, die Folge $VIII$ die Zahl 8 und die Folge XIV die Zahl 14.

Die **Syntax** und die **Semantik** der römischen Zahlen lassen sich durch eine Menge von Regeln beschreiben. Zunächst hat jedes Symbol einen Wert:

I hat den Wert 1,
V hat den Wert 5,
X hat den Wert 10,
L hat den Wert 50,
C hat den Wert 100,
D hat den Wert 500,
M hat den Wert 1000.

Folgen dieser Symbole werden von links nach rechts gelesen, und alle Symbole dem Wert nach absteigend sortiert. Zusätzlich dürfen alle Symbole, außer dem Symbol M, maximal viermal hintereinander in der Folge auftauchen. Nach diesen Regeln ist $MMMMMCCCCLXVIIII$ eine römische Zahl. Aber die Folgen $IIVXM$, IVI und $CCCCCVIII$ sind keine römischen Zahlen.

Für eine römische Zahl bestimmen wir den Wert der Zahl, indem wir die Werte der Symbole addieren. Damit hat die Folge $MMMMMCCCCCLXVIIII$ den Wert $1000 + 1000 + 1000 + 1000 + 1000 + 100 + 100 + 100 + 100 + 50 + 10 + 5 + 1 + 1 + 1 + 1 = 5469$.

Jetzt können wir bereits alle Zahlen darstellen, weil wir immer dann, wenn wir eins der Symbole fünf Mal addieren wollen, einfach das nächste, höherwertige Symbol einsetzen dürfen. Aus $IIIII$ wird V, aus $VIIIII$ wird X, aus $XIIIII$ wird XV und so weiter.

Um die Lesbarkeit zu erhöhen, versuchen wir auch immer mit möglichst wenigen Symbolen auszukommen. Wir schreiben X statt VV oder M statt DD. Wir sehen damit, dass in jeder römischen Zahl die Symbole V, L und D immer nur maximal einmal vorkommen.

Um die Länge einer römischen Zahl weiter zu verkürzen, gibt es zusätzlich die sogenannte Subtraktionsregel. Diese Regel besagt, dass wir die Symbole I, X und C einem ihrer beiden nächstgrößeren Symbole voranstellen dürfen und dann ihren Wert von dessen Wert subtrahieren können. Mit dieser Regel wird aus $IIII$ die Folge IV und aus $VIIII$ die Folge IX. Die Zahl 99 ist aber nicht IC sondern $XCIX$. Mit der Subtraktionsregel wird aus unserer Zahl 5469 die Folge $MMMMMCDLXIX$.

Damit haben wir die grundlegende Syntax und die Semantik der römischen Zahlen kennen gelernt. Bereits an dieser Stelle sehen wir, dass wir Zahlen damit einigermaßen ordentlich darstellen können, dass es aber überhaupt keinen Spaß macht mit dieser Zahlendarstellung zu rechnen. Denn manchmal können wir Folgen einfach konkatenieren. So ergibt $M + MM = MMM$. Dabei kann es notwendig sein Teilfolgen zu ersetzen. So ergibt $III + III = IIIIII = VI$. Oft müssen wir auch die Subtraktionsregel auf das Ergebnis anwenden. Wir bekommen zum Beispiel $VII + VII = VVIIII = XIIII = XIV$. Oder wir müssen die Subtraktionsregel in der Eingabe auflösen, bevor wir sie auf das Ergebnis erneut anwenden. Wir bekommen zum Beispiel $XCIX + XCIX = CXCVIII$.

Arabische Zahlen lassen sich viel einfacher verrechnen. In Europa veröffentlichte im Jahr 1525 der Mathematiker Adam Ries das Buch „Rechnung auff der linihen und federn“ als eins der frühesten deutschen Lehrbücher über Arithmetik und Algebra. Es behandelt verschiedene mathematische Themen, darunter die vier Grundrechenarten Addition, Subtraktion, Multiplikation und Division. Darüber hinaus behandelte das Buch die Bruchrechnung, Maßeinheiten, sowie praktische Probleme aus dem Alltag, wie Handel, Landvermessung und Zeitrechnung. Adam Ries trug mit diesem Buch maßgeblich zur Popularisierung des Dezimalsystems und der grundlegenden Rechenvorschriften in Europa bei. Bis heute geht auf Adam Ries die Redewendung: „Das macht nach Adam Riese …“ zurück.

[von Adam Ries, Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=3790545]

Die Rechenvorschriften arabischer Zahlen lernen wir bereits in der Grundschule. So addieren wir zwei Zahlen, indem wir diese untereinanderschreiben und alle Ziffern stellenweise von hinten nach vorne, mit Übertrag, addieren.

$$\begin{array}{cccccc} & & 1 & 3 & 3 & 7 \\ + & & & 9 & 4 & 2 \\ & {\scriptstyle 1} & & & & \\ \hline & & 2 & 2 & 7 & 9 \end{array}$$

Diese systematische Methode der Addition zweier Zahlen können wir als eine Folge von Schritten beschreiben.

Wenn du zwei Zahlen addieren willst, schreibe beide untereinander, so dass Einer, Zehner, Hunderter, Tausender, usw. jeweils untereinanderstehen. Führe dann die folgenden Schritte durch.

(Schritt 1) Addiere zuerst beide Einer-Ziffern. Ist das Ergebnis kleiner als 10 ist dies die Einer-Ziffer der Summe beider Eingaben. Ist das Ergebnis größer oder gleich 10, so ist die Einer-Ziffer des Ergebnisses die Einer-Ziffer der Summe beider Eingaben und wir bekommen einen Übertrag an der Zehner-Ziffer.

(Schritt 2) Addiere jetzt die beiden Zehner-Ziffern. Ist ein Übertrag an der Zehner-Ziffer erhöhe das Ergebnis um 1. Ist das Ergebnis kleiner als 10 ist dies die Zehner-Ziffer der Summe beider Eingaben. Ist das Ergebnis größer oder gleich 10, so ist die Einer-Ziffer des Ergebnisses die Zehner-Ziffer der Summe beider Eingaben und wir bekommen einen Übertrag an der Hunderter-Ziffer.

(Schritt 3) Addiere jetzt die beiden Hunderter-Ziffern. Ist ein Übertrag an der Hunderter-Ziffer erhöhe das Ergebnis um 1. Ist das Ergebnis kleiner als 10 ist dies die Hunderter-Ziffer der Summe beider Eingaben. Ist das Ergebnis größer oder gleich 10, so ist die Einer-Ziffer des Ergebnisses die Hunderter-Ziffer der Summe beider Eingaben und wir bekommen einen Übertrag an der Tausender-Ziffer.

usw.

Wir führen diese Schritte für jede Ziffer der Eingaben durch. Haben wir alle Stellen der Eingaben betrachtet, ist das Ergebnis die Summe unserer Addition.

Mit dieser systematischen Methode können wir, selbst wenn wir nur einfache Ziffern addieren können, Additionen viel größerer Zahlen sicher und schnell durchführen. Diese Art von **systematischen Methoden zur Berechnung eines Ergebnisses** ist Kern der Informatik. In der Informatik nennen wir ein Verfahren, was für eine ganze Klasse verschiedenster Eingaben je die zugehörige, passende Ausgabe systematisch berechnen kann, einen **Algorithmus**.

Der Begriff **Algorithmus** leitet sich dabei vom Namen des persischen Mathematikers Muhammad ibn Musa al-Chwarizmi ab. In seinem im 9. Jahrhundert geschriebenen Buch, das übersetzt „Regeln der Wiedereinsetzung und Reduktion“ heißt, entwickelte er eine systematische Methode zur Lösung von Gleichungen, die als al-Chwarismi-Methode bekannt wurde. Seine Arbeit war ein bedeutender Beitrag zur Entwicklung der Algebra und hatte einen großen Einfluss auf die Mathematik im Mittelalter und darüber hinaus. Im Laufe der Zeit wurde dann aus dem Namen al-Chwarismi das Wort Algorithmus, um systematische Methoden zur Lösung von Problemen zu beschreiben.

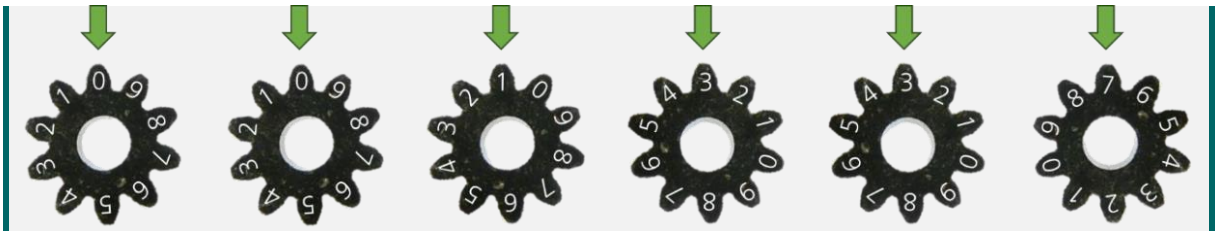
[von Unknown, Gemeinfrei, https://commons.wikimedia.org/w/index.php?curid=2652443]

Nachdem sich nun also das Dezimalsystem in Europa durchgesetzt hatte, lag auch der nächste Schritt, diese systematischen Methoden zu automatisieren, quasi auf der Hand. In Europa wurden jetzt die ersten **mechanischen Rechenmaschienen** erfunden, die dann je die speziellen Arbeitsschritte eines Algorithmus mechanisch durchführen konnten.

Wilhelm Schickard, ein deutscher Astronom, Mathematiker und Erfinder, entwarf und baute 1623 eine mechanische Rechenuhr, die als die erste bekannte mechanische Rechenmaschine gilt. Diese Maschine wurde entwickelt, um die vier Grundrechenarten auszuführen. Schickard konzipierte seine Rechenuhr für seine Zeit sehr fortschrittlich, und sie stellte den ersten großen Schritt in der Entwicklung zahlreicher weiterer Rechenmaschinen dar.

[von Herbert Klaeren, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8159979]

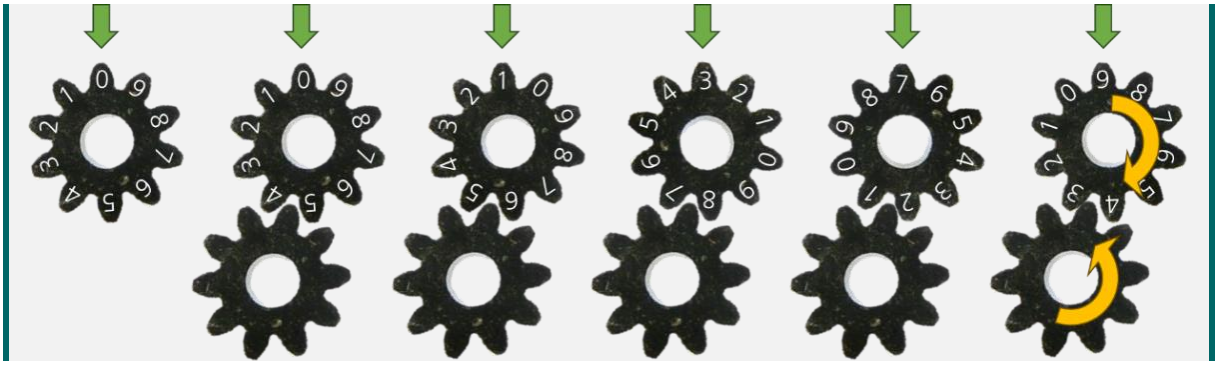
Um zu verstehen, wie man einen Algorithmus mechanisch umsetzen kann, schauen wir uns die Funktionsweise von Schickards Rechenuhr einmal genauer an. Dazu betrachten wir zunächst, wie die Maschine Zahlen, zuerst noch ohne Übertrag, addiert. Die folgende Abbildung zeigt sechs in einer Reihe angeordnete Zahnräder, die wir mit den Ziffern 0 bis 9 beschriften. Wir können die Zahnräder drehen und lesen immer die obenstehende Ziffer ab. So können wir alle Zahlen von 0 bis 999999 sehr einfach darstellen. Die folgende Abbildung zeigt die Zahl 1337.



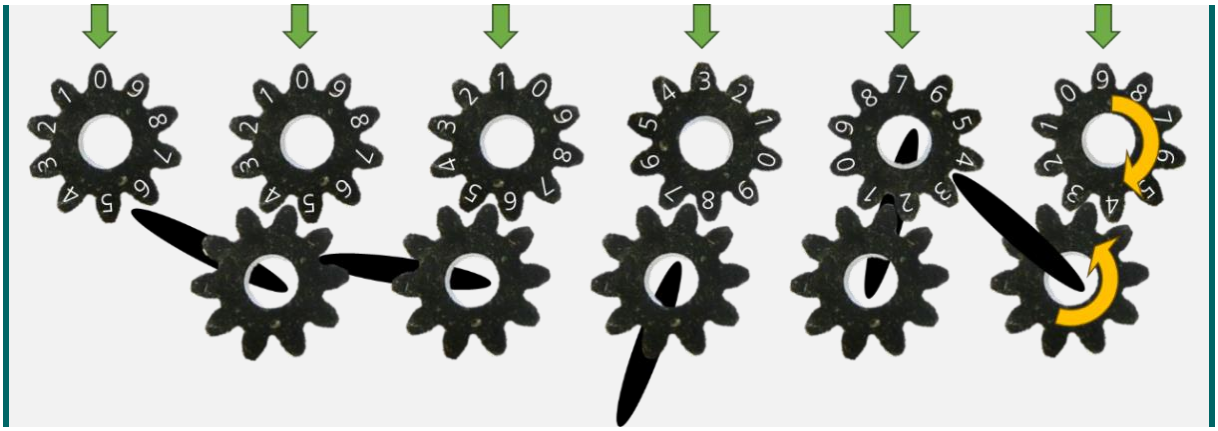
Jetzt fangen wir mit dem Addieren an. Betrachten wir als Beispiel, dass wir zu 1337 die Zahl 42 addieren wollen. Das schaffen wir, indem wir die auf 001337 eingestellten Räder entsprechend der 000042 im Uhrzeigersinn „weiterdrehen“. Das bedeutet, wir drehen die ersten vier Räder nicht, das fünfte Rad um 4 Positionen und das letzte Rad um 2 Positionen weiter nach rechts. Dieses Weiterdrehen können wir durch einfaches Abzählen leicht bewerkstelligen, wirklich etwas

rechnen müssen wir dazu nicht. In diesem Beispiel stehen die Räder nach dem Drehen auf 001379 und zeigen damit die Zahl 1379 als richtiges Ergebnis unserer Summe.

Um jetzt auch Überträge zu berücksichtigen, erweitern wir unsere Rechenuhr. Zunächst benötigen wir für jedes der fünf hinteren Zahnräder, ein zweites Rad. Jedes dieser zusätzlichen Zahnräder dreht sich mit seinen Partner aus der ersten Reihe, nur eben andersherum, gegen den Uhrzeigersinn. Die folgende Abbildung zeigt diese einfache Erweiterung.



Jetzt befestigen wir unter jedem unteren Zahnrad einen langen Hebel, der sich mit seinem Rad dreht. Die Aufgabe des Hebels ist es, das nächste Zahnrad, um genau eine Position zu drehen, wenn das eigene Partnerrad von der 9 auf die 0 wechselt. Für unsere Rechenuhr müssen wir jetzt lediglich alle Räder und Hebel so anordnen, so dass die Hebel immer nur genau das nächste Rad treffen, ohne mit anderen Rädern zu verhaken.



In der Abbildung unserer Maschine sehen wir jetzt, dass der Hebel des Rads der Einer-Stelle das Rad der Zehner-Stelle um eine Position schubsen wird, wenn wir das Rad der Einer-Stelle von der 9 auf die 0 drehen. Das würde automatisch an der Zehner-Stelle die Ziffer 7 auf die Ziffer 8 verstellen. Der Hebel der Zehner-Stelle kommt dadurch eine Position näher an das Rad der Hunderter-Stelle. Addieren wir noch 20 indem wir das fünfte Ziffernrad um 2 Positionen weiterdrehen, bekommen wir dann auch einen Übertrag an der Hunderter-Stelle.

Damit ist unsere Addiermaschine fertig! Mit dieser Maschine können wir Zahlen bis zu einem Ergebnis von 999999 addieren. Wollen wir noch größere Zahlen addieren, spendieren wir mehr Zahnräder. Wollen wir subtrahieren, drehen wir die entsprechenden Zahnräder einfach gegen den

Uhrzeigersinn. Die Rechenuhr von Wilhelm Schickard hat damit das schriftliche Addieren und Subtrahieren mechanisch umgesetzt. Aus dem Algorithmus ist ein Automat geworden.

Die Rechenuhr kann aber nicht nur addieren und subtrahieren, sie kann auch multiplizieren. Hier ist die grundsätzliche Idee die Multiplikation, genau wie beim schriftlichen Multiplizieren, aus vielen Additionen zusammenzusetzen. Auch hier sehen wir eine der zentralen Ideen der Informatik in Aktion: Wir haben bereits eine Maschine, die addieren kann. Wenn es uns gelingt, die Multiplikation als Folge von Additionen darzustellen, können wir unsere Addiermaschine auch zum Multiplizieren verwenden.

Die folgende Abbildung zeigt wie sich die Multiplikation der Zahl 1337 mit der Zahl 4 aus einer Summe von vier Teilergebnissen zusammensetzen lässt. 1 mal 4 ist 4, 3 mal 4 ist 12, 3 mal 4 ist 12 und 7 mal 4 ist 28. Rücken wir die Teilergebnisse an die richtige Stelle, bekommen wir $4000 + 1200 + 120 + 28 = 5348$ als Ergebnis der Multiplikation.

1	3	3	7	*	4
4					
1	2				
	1	2			
		2	8		
5	3	4	8		

Wir sehen, an diesem einfachen Beispiel, dass wir, um die Teilergebnisse zu erhalten, immer nur zwei Ziffern multiplizieren müssen. Um das zu ermöglichen, besitzt Schickards Rechenuhr sechs Walzen. Jede dieser Walzen ist mit der folgenden Multiplikationstabelle beklebt. In der Rechenuhr kann man jede der sechs Walzen auf genau eine Spalte der Multiplikationstabelle einstellen und dann über alle Walzen hinweg alle, bis auf eine Zeile, mit Schiebern ausblenden.

	0	1	2	3	4	5	6	7	8	9
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Betrachten wir das Ganze an unserem Beispiel und multiplizieren 1337 mit der Zahl 4. Wir gehen jetzt davon aus, dass wir sechs Walzen haben, die alle mit der Multiplikationstabelle beklebt sind. Jetzt stellen wir zunächst die sechs Walzen auf 001337 ein. Dadurch erhalten wir die folgende Ansicht.

Die Walzen Eins und Zwei zeigen die 0er-Spalte, Walze Drei zeigt die 1er-Spalte, Walze Vier und Fünf zeigen die 3er-Spalte und Walze Sechs zeigt die 7er-Spalte der Multiplikationstabelle.

0	**0**	**1**	**3**	**3**	**7**
0	0	1	3	3	7
0	0	2	6	6	14
0	0	3	9	9	21
0	0	4	12	12	28
0	0	5	15	15	35
0	0	6	18	18	42
0	0	7	21	21	49
0	0	8	24	24	56
0	0	9	27	27	63

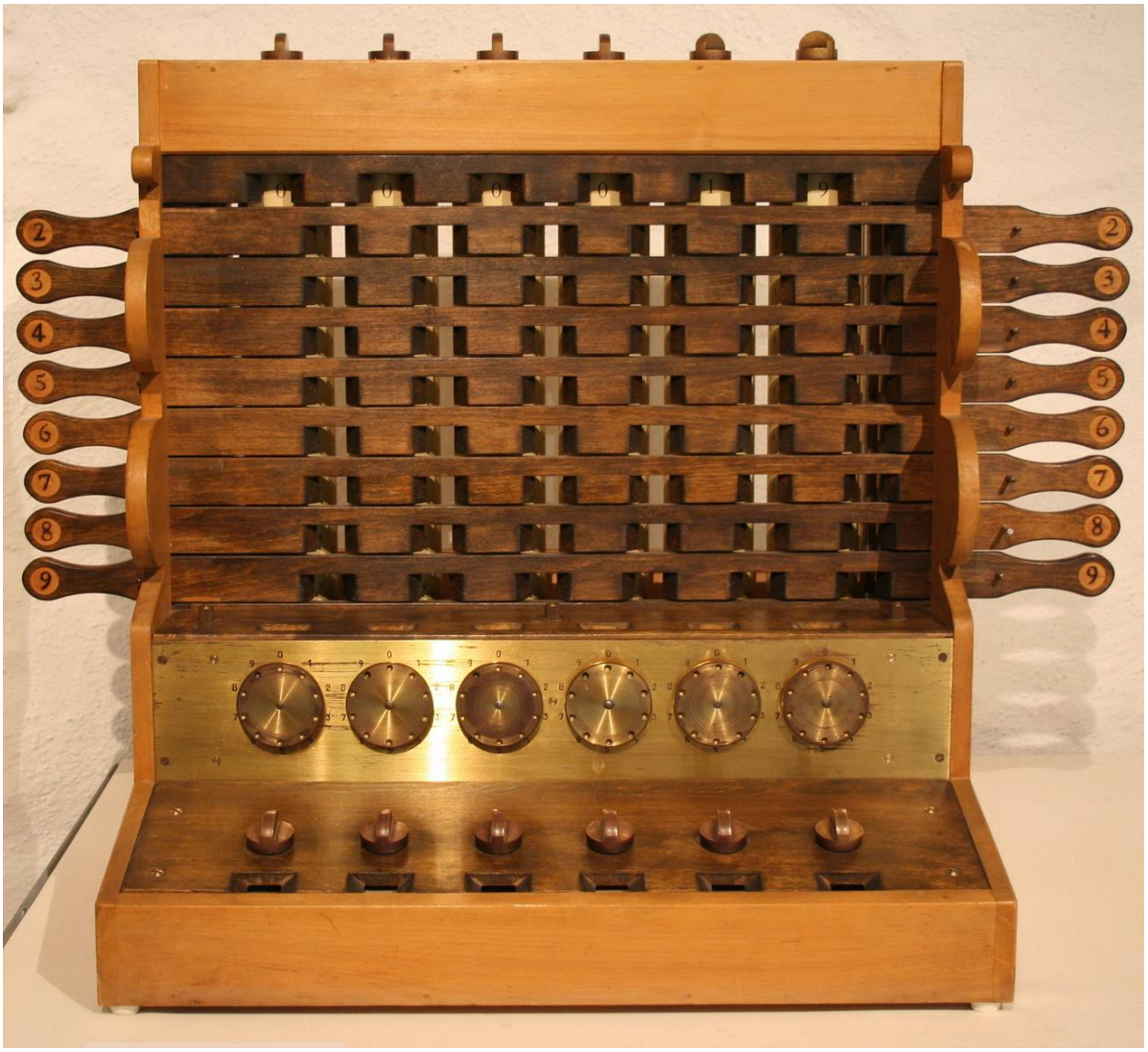
In Schickards Rechenuhr sind alle Walzen zunächst abgedeckt. Da wir unsere Eingabe mit 4 multiplizieren wollen, existiert ein mit 4 beschrifteter Schieber, der lediglich die 4er-Zeilen unserer sechs Walzen aufdeckt. Die Idee ist es genau die Zahlen aufzudecken, die wir im nächsten Schritt addieren müssen. Ziehen wir den Schieber erhalten wir die folgende Ansicht.

0	**0**	**1**	**3**	**3**	**7**
0	0	4	12	12	28

Die Walzen sind so im oberen Teil der Rechenuhr angeordnet, dass wir jetzt die Zahlen 4, 12, 12 und 28 einfach ablesen und per Hand in unsere Addiermaschine übertragen können. Dabei übertragen wir jede der vier Zahlen an der Position, an der sie angezeigt wird. Für die 4 drehen wir die Tausender-Stelle der Addiermaschine um 4 Positionen. Für die erste 12 drehen wir die Hunderter-Stelle um 2 und die Tausender-Stelle um eine weitere Position, usw.

Wollen wir unsere Zahl 1337 mit der Zahl 42 multiplizieren, führen wir einfach zwei Multiplikationen durch. Zuerst multiplizieren wir, genau wie eben beschrieben, 1337 mit 4, tragen die Zahlen der Walzen jetzt aber um eine Stelle nach links verschoben in die Addiermaschine ein. Dadurch bekommen wir 053480 auf der Addiermaschine. Jetzt decken wir die 4er-Stellen der Walzen zu und öffnen die 2er-Stellen. Dort erscheint jetzt die Folge 0 0 2 6 6 14, was wir, ohne zu verschieben, in die Addiermaschine übertragen. Insgesamt haben wir dann mit 42 multipliziert.

Die folgende Abbildung zeigt noch einmal Schickards Rechenuhr. Die sechs Walzen sind senkrecht im oberen Teil der Rechenuhr angeordnet. Oben können wir die Walzen auf die richtige Eingabe drehen. Die Schieber auf der linken und rechten Seite decken die Walzen mit den Multiplikationstabellen an der richtigen Stelle auf. Die Einstellräder der Addiermaschine sehen wir unten, auf dem Fuß der Maschine. So lassen sich die aufgedeckten Zahlen gut übertragen. Im Internet können Sie kurze Videos finden, die Nachbauten dieser Maschine in Aktion zeigen.

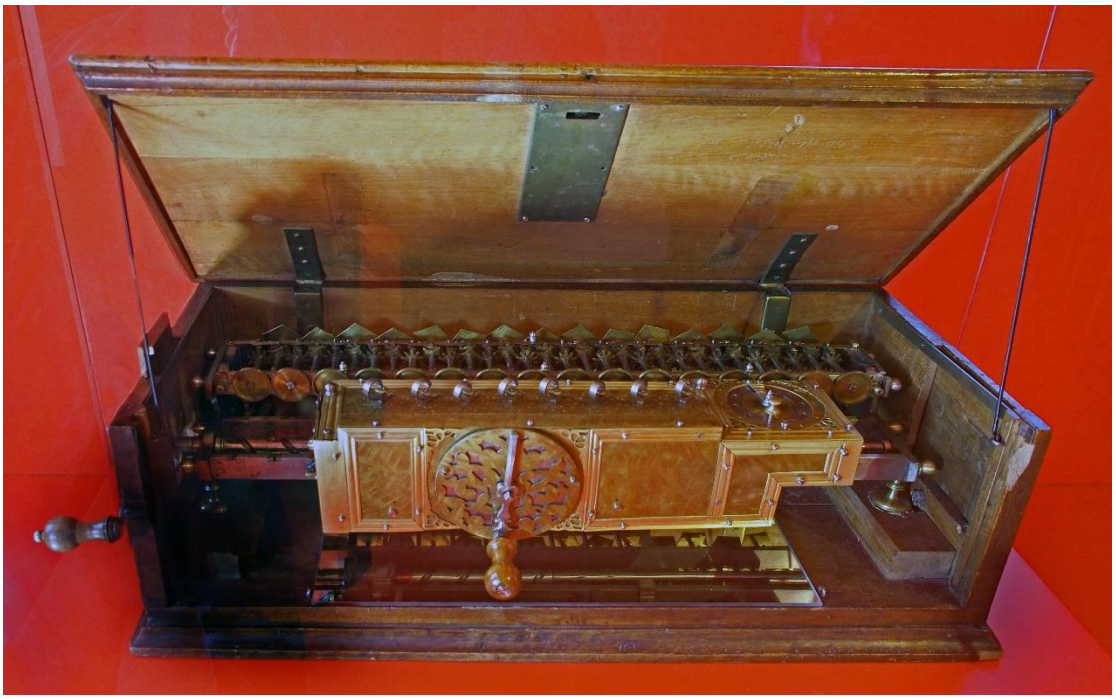


[von Herbert Klaeren, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=8159979]

Neben Schickards Rechenuhr entwickelten sich dann weitere mechanische Maschinen. **Blaise Pascal**, ein französischer Mathematiker, Physiker, Erfinder und Philosoph, entwarf 1645 eine Addiermaschine, die als **Pascaline** bekannt wurde. Die Pascaline addiert nach der gleichen Idee wie Schickards Rechenuhr, ordnet die Zahnräder und Hebel lediglich etwas anders an. Außerdem besitzt die Pascaline Wählscheiben, die man mit einem Stift leicht drehen kann, um Zahlen in die Maschine einzugeben.

[von David.Monniaux, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=186079]

Gottfried Wilhelm Leibniz, ein deutscher Philosoph, Mathematiker und Erfinder, entwarf im Jahr 1690 eine dezimale Multiplikationsmaschine, die als **Stepped Reckoner** bekannt wurde. Der Stepped Reckoner benutzt aber keine Multiplikationstabelle, sondern basiert auf dem Prinzip der Leibnizschen Zahnräder, bei dem Walzen mit unterschiedlich vielen Zähnen verwendet werden, um Multiplikation und Division durchzuführen.



[von Hajotthu, Museum Herrenhausen Palace, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=28141911]

Die **Rechenuhr**, die **Pascaline** und der **Stepped Reckoner** konnten uns also dabei helfen, die Grundrechenarten durchzuführen. Leider hatten diese Automaten drei große Probleme. Erstens war die Feinmechanik noch nicht so weit entwickelt, dass man die Maschinen über längere Zeiträume zuverlässig nutzen konnte. Zahnräder und Hebel gingen einfach zu schnell kaputt. Zweitens konnte jede dieser Maschinen nur genau das ausführen, für den Zweck, für den sie gebaut wurde. Eine Addiermaschine konnte eben nur addieren. Die mechanischen Maschinen waren in diesem Sinne nicht programmierbar. Drittens waren die Maschinen sehr langsam, da jede Eingabe manuell und durch Drehen oder Kurbeln von Rädern vorgenommen werden musste.

Im Jahr 1934 hatte der deutsche Ingenieur **Konrad Zuse** die Idee, das duale Zahlensystem für das maschinelle Rechnen zu verwenden. Bis zu diesem Zeitpunkt benutzten alle Automaten das Dezimalsystem zum Rechnen, wobei Zahnräder und Walzen auf zehn verschiedene Positionen eingestellt werden mussten. Das duale Zahlensystem kennt hingegen nur die Ziffern 0 und 1. Eine von Zuses zentralen Ideen war es, jede Stelle einer Dualzahl durch einen Schalter darzustellen. Ist der Schalter offen, entspricht das der Ziffer 0, ist der Schalter geschlossen, entspricht das der Ziffer 1. Warum dieser Schritt entscheidend für die Entwicklung des modernen Computers war, wollen wir uns nun überlegen.

[von Wolfgang Hunscher, CC BY 3.0, https://commons.wikimedia.org/w/index.php?curid=620905]

Wir kennen die Dualzahlen aus der Schule. Wir wissen, dass man mit Dualzahlen praktischerweise genauso gut rechnen kann wie mit Dezimalzahlen. Trotzdem wollen wir das an dieser Stelle an einem Beispiel kurz noch einmal wiederholen. Die Dezimalzahl 1337 ist die Dualzahl 10100111001. Das können wir leicht nachrechnen.

$$
\begin{aligned}
&(10100111001)_{bin} \\
&= 1 \cdot 2^{10} + 0 \cdot 2^9 + 1 \cdot 2^8 + 0 \cdot 2^7 + 0 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1024 + 256 + 32 + 16 + 8 + 1 \\
&= 1337
\end{aligned}
$$

Die Dezimalzahl 42 ist die Dualzahl 101010.

$$
\begin{aligned}
&(101010)_{bin} \\
&= 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \\
&= 32 + 8 + 2 \\
&= 42
\end{aligned}
$$

Genau wie Dezimalzahlen, können wir zwei Dualzahlen addieren, indem wir sie untereinanderschreiben und stellenweise betrachten.

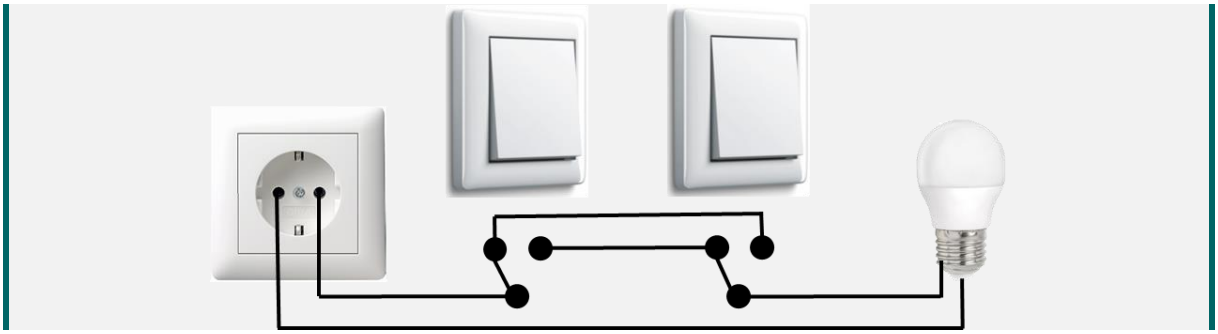
$$
\begin{array}{cccccccccccc}
 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 1 \\
+ & & & & & & 1 & 0 & 1 & 0 & 1 & 0 \\
 & & & & & 1 & 1 & 1 & & & & \\
\hline
 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1
\end{array}
$$

Und, nur zur Kontrolle, rechnen wir die Zahl 10101100011 wieder zurück ins Dezimalsystem.

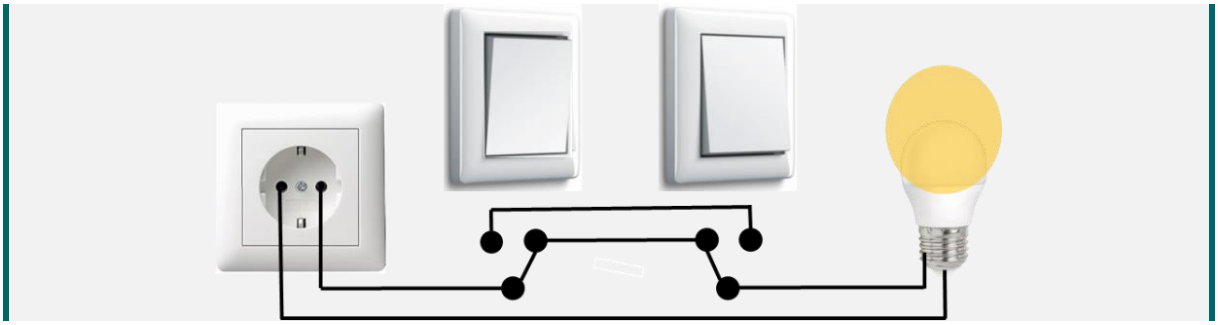
$$
\begin{aligned}
&(10101100011)_{bin} \\
&= 1 \cdot 2^{10} + 0 \cdot 2^9 + 1 \cdot 2^8 + 0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 \\
&= 1024 + 256 + 64 + 32 + 2 + 1 \\
&= 1379
\end{aligned}
$$

Auch die Subtraktion, Multiplikation und Division können wir mit Dualzahlen wie gewohnt schriftlich durchführen. Aber was ist jetzt der Vorteil dieser binären Zahlendarstellung? Um das zu erkennen, bauen wir uns jetzt eine **elektromechanische Addiermaschine**.

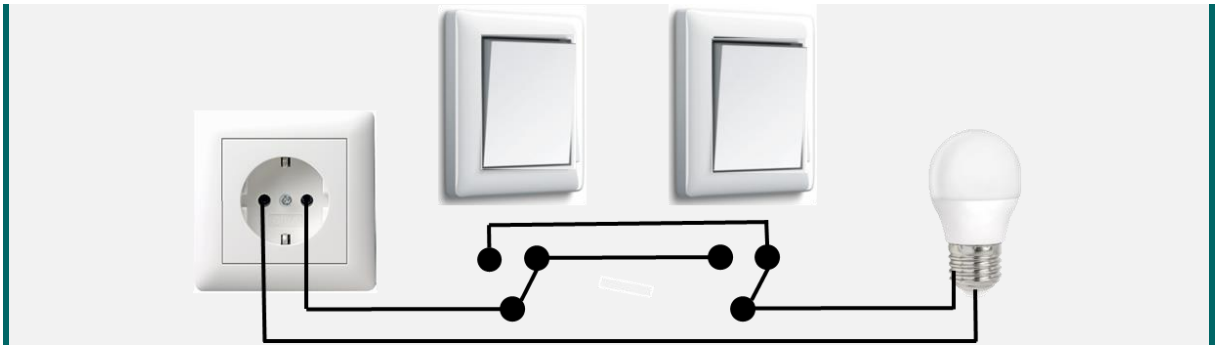
Die folgende Abbildung zeigt einen Aufbau, mit dem wir versuchen wollen, zwei binäre Ziffern zu addieren. Wir sehen eine Steckdose, eine Lampe und zwei Schalter in einer sogenannten **Wechselschaltung**. Wir brauchen eine Wechselschaltung immer dann, wenn wir eine Lampe von mehreren Schaltern aus bedienen wollen.



Ich habe eine Wechselschaltung in meinem Schlafzimmer. Das ist sehr praktisch. Der eine Schalter ist an meinem Bett, der zweite Schalter ist an der Tür. Wenn ich schlafe, ist das Licht aus und auch in der Abbildung ist der Stromkreis nicht geschlossen. Ich kann jetzt den einen oder den anderen Schalter benutzen, um das Licht einzuschalten. Denn verändere ich die Position eines der beiden Schalter, schließt sich der Stromkreis. Entweder der erste Schalter bewegt sich und der Strom fließt „unten entlang“, oder der zweite Schalter bewegt sich und der Strom fließt „oben entlang“. Sagen wir ich wache auf und drücke den ersten Schalter neben meinem Bett, das Ergebnis sehen wir in der folgenden Abbildung. Das Licht geht an.



Jetzt kann ich entweder den Schalter neben meinem Bett nochmal drücken, oder ich kann den Schalter neben der Tür benutzen, um das Licht wieder auszuschalten. Wenn ich das Schlafzimmer verlasse, kann ich also einfach den Schalter an der Tür benutzen, um die Lampe wieder auszuschalten. Beide Schalter sind „an“, der Stromkreis ist nicht geschlossen und das Licht ist aus.



Eine Wechselschaltung mit zwei Schaltern hat also 4 Zustände. Aus und aus ergibt aus. An und aus ergibt an. Aus und an ergibt an. An und an ergibt aus. Ist die Lampe an oder aus, kann ich einen der zwei Schalter drücken, um den Zustand der Lampe zu verändern.

Mit dieser Wechselschaltung können wir jetzt, genau wie mit unseren Zahnrädern, zwei Binärzahlen, zunächst ohne Übertrag addieren. Dazu codieren wir ganz einfach:

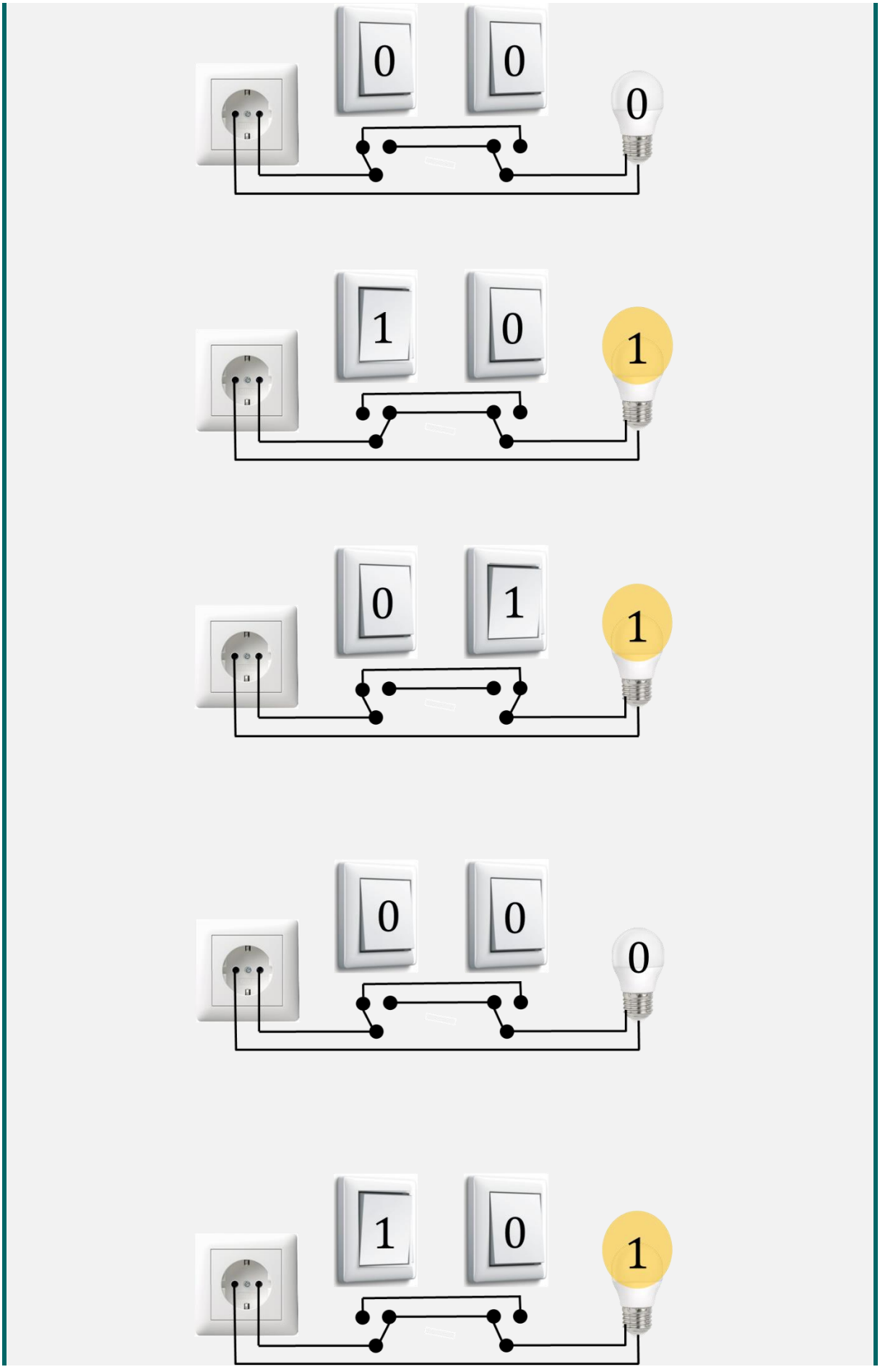
Schalter gedrückt ist 1,
Schalter nicht gedrückt ist 0,
Lampe an ist 1,
Lampe aus ist 0.

Betrachten wir damit jetzt noch einmal unsere Wechselschaltung, bekommen wir

$$0 + 0 = 0,$$
$$1 + 0 = 1,$$
$$0 + 1 = 1,$$
$$1 + 1 = 0.$$

Das ist genau die „Rechnung“, die wir für das ziffernweise Addieren zweier Binärzahlen brauchen.

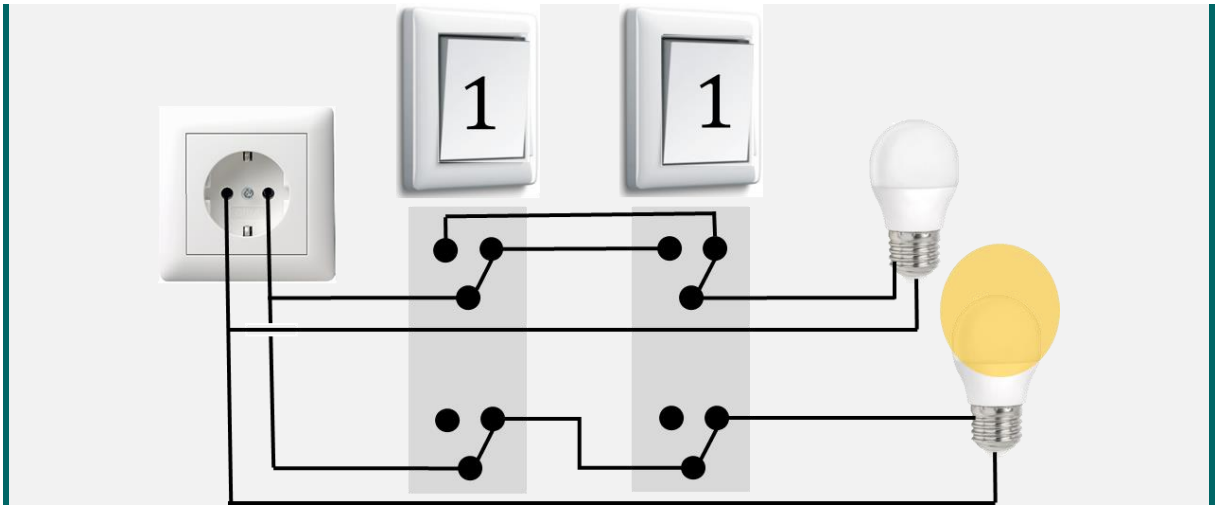
Auf der nächsten Seite sehen wir fünf unserer Wechselschaltungen. Lesen wir die ersten Schalter von unten nach oben, bekommen wir die Binärzahl 10010 und damit die Dezimalzahl 18. Lesen wir die zweiten Schalter von unten nach oben, bekommen wir die Binärzahlzahl 100 und damit die Dezimalzahl 4. Das sind die zwei Eingaben in unsere Addiermaschine. Haben wir die Schalter entsprechend gedrückt, zeigen die Lampen unser Ergebnis. Wieder lesen wir dazu die Lampen von unten nach oben und bekommen die Binärzahl 10110, die Dezimalzahl 22. Unsere elektronische Addiermaschine hat hier $18 + 4 = 22$ gerechnet.


0
0
0
1
0
1
0
1
1
0
0
0
1
0
1

Solang bei der Addition kein Übertrag vorkommt, können wir jetzt also Binärzahlen elektronisch addieren. Werden die Zahlen länger, müssen wir wieder für jede der Ziffernpaare eine weitere Lampe mit zwei Schaltern spendieren. Wenn wir Strom zur Verfügung haben, geht die Rechnung rasend schnell. Die Bauteile sind viel unaufwendiger als Zahnräder. Zusätzlich können wir diese sehr flexibel anordnen. Insgesamt bekommen wir so eine viel kompaktere und langlebigere Addiermaschine. Jetzt müssen wir unsere Maschine nur noch so erweitern, dass diese auch mit Überträgen umgehen kann.

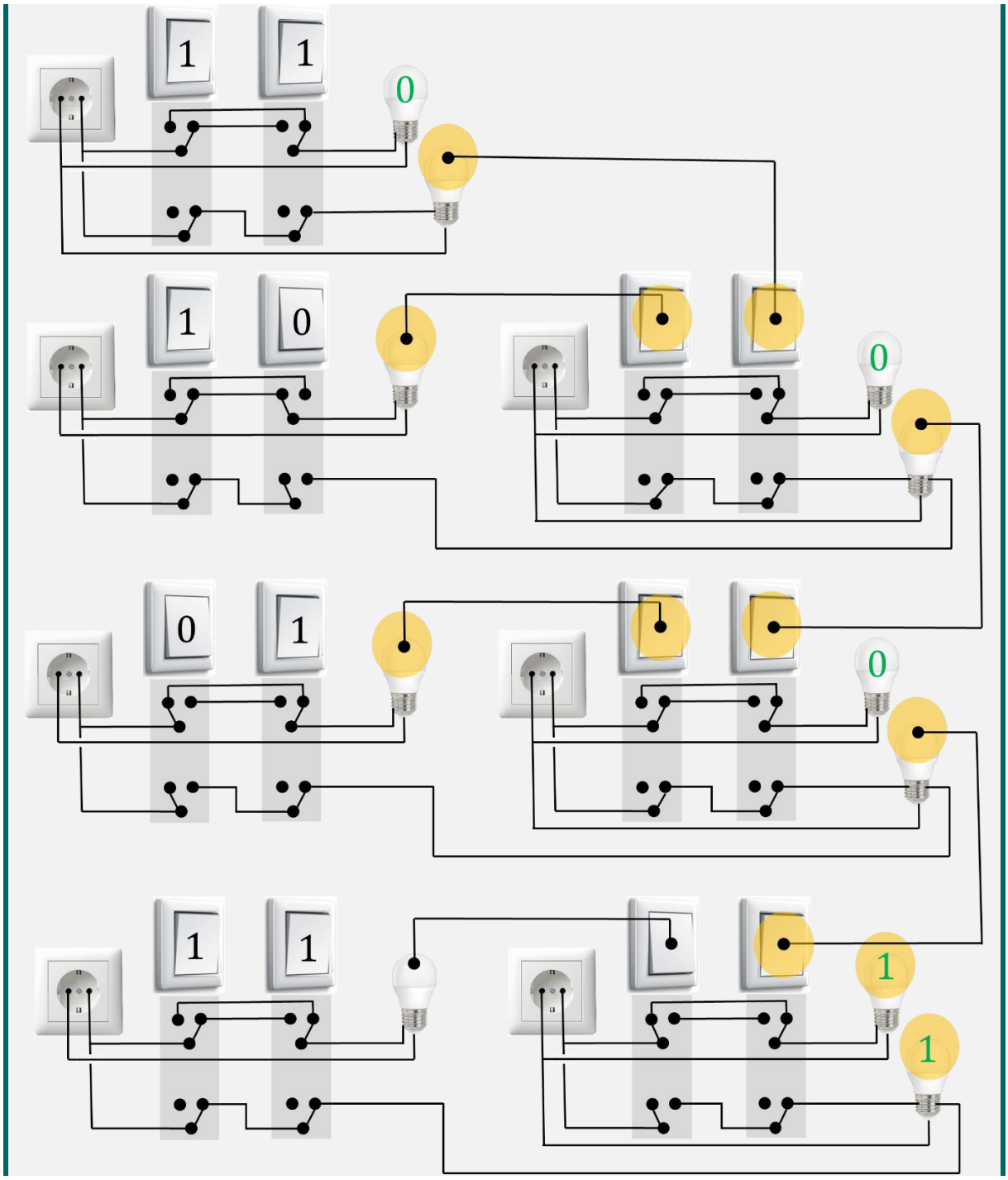
Wenn wir zwei binäre Ziffern addieren, bekommen wir immer genau dann einen **Übertrag**, wenn beide Ziffern 1 sind. Wenn wir das in Lichtschaltern denken, entspricht das einer **Reihenschaltung**. Der Strom geht durch den ersten Schalter zum zweiten Schalter und von dort aus zur Lampe. Nur wenn beide Schalter an sind, ist die Lampe an.

Das folgende Bild zeigt eine Erweiterung einer Reihe unserer elektromechanische Addiermaschine. In dieser Erweiterung benutzen wir jede Eingabe zweimal. Das ist hier so abgebildet, dass jeder Schalter über einem grauen Kasten liegt. Die zwei Weichen innerhalb jedes Kastens werden immer nur gleichzeitig und immer vom Schalter über dem Kasten bewegt. Oben, genau wie bisher, benutzen wir beide Eingaben in einer Wechselschaltung. Unten, das ist neu, benutzen wir beide Eingaben in einer Reihenschaltung. Die **Wechselschaltung** gibt uns die Ziffer der **Addition**. Die **Reihenschaltung** bestimmt, ob es einen **Übertrag** gibt, oder nicht. In diesem Beispiel sind beide Eingaben 1. Das Ergebnis der Addition ist 0, der Übertrag ist 1. Unsere Maschine kann jetzt Binärziffern addieren. In der Informatik nennen wir eine solche Maschine einen **Halbaddierer**.



Doch damit sind wir noch nicht fertig. Um nun ganze Binärzahlen addieren zu können, brauchen wir eine weitere Idee. Denn bei der Addition von zwei Ziffern müssen wir auch den Übertrag aus der vorherigen Rechnung berücksichtigen. Im Grunde addieren wir den Übertrag zum Ergebnis der Summe der nächsten beiden Ziffern. Dadurch wird das Ergebnis jeder Addition auch Teil der Eingabe der nächsten Addition. Es reicht nun nicht mehr aus, nur eine Lampe für den Übertrag anzuschalten, der Übertrag muss einen Schalter für die nächste Addition betätigen. Ein solcher elektromechanischer Schalter ist ein **Relais**. Bekommt ein Relais Strom, schließt es den Schalter. Bekommt ein Relais keinen Strom, bleibt der Schalter offen. Als Konrad Zuse seinen Rechner baute, wurden solche Relais für die Telegraphie bereits verwendet.

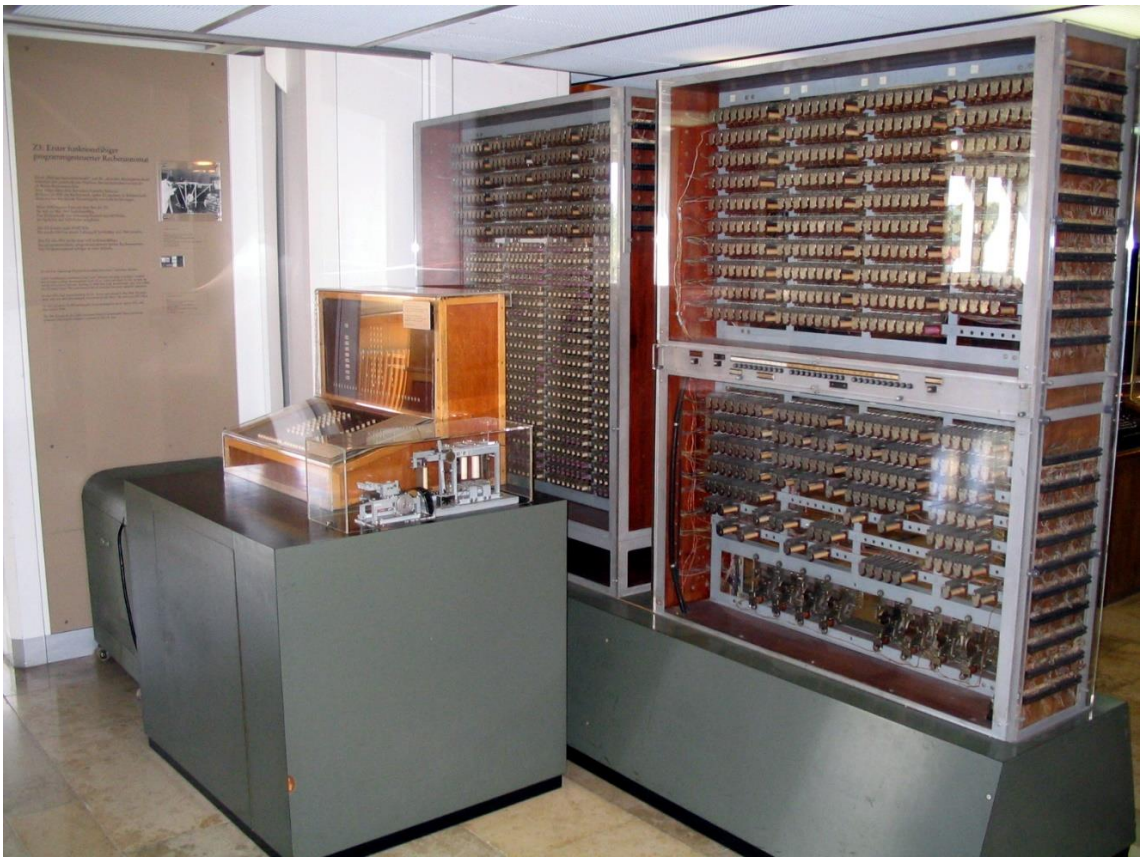
Jetzt bauen wir mit Hilfe von Relais unsere vollständige elektromechanische Addiermaschine. Auf dieser Abbildung sehen wir, wie wir zwei vierstellige Binärzahlen addieren können. Die Relais werden hier so dargestellt, dass Lampen Schalter betätigen. Wenn Lampe und Schalter miteinander verbunden sind, bedeutet das, dass der Schalter geschlossen ist, genau dann, wenn die Lampe leuchtet.



Wir sehen hier unsere fertige Addiermaschine wie sie die zwei Binärzahlen 1011 und 1101 addiert. Wieder geben wir die Zahlen über die zwei Reihen von Schaltern ein, die wir jeweils von unten nach oben lesen. Ganz oben, in der ersten Reihe der Maschine, sehen wir einen Halbaddierer, der die hinteren beiden Ziffern der Zahlen verrechnet, denn für die ersten zwei Ziffern gibt es nie einen

Übertrag. In der nächsten Reihe sehen wir zwei Halbaddierer. Der erste verrechnet die nächsten zwei Ziffern. Der zweite addiert auf dieses Ergebnis den Übertrag der vorherigen Rechnung. In der Abbildung werden die Schalter des zweiten Halbaddierers der zweiten Reihe von den Lampen, also von den Ergebnissen, der vorherigen zwei Rechnungen gedrückt. Zusätzlich bekommen wir in der zweiten Zeile einen Übertrag, wenn der erste Halbaddierer oder der zweite Halbaddierer einen Übertrag generiert. Das stellen wir dar, indem die Übertrag-Lampe von links oder von rechts Strom beziehen kann. Zeile drei und vier unserer Maschine funktionieren genau wie die zweite Zeile. Immer wird der Übertrag der vorherigen Rechnung mit einbezogen und evtl. ein neuer Übertrag generiert. Wir lesen die Lampen ab und bekommen das Ergebnis 11000 als Ergebnis unser Summe. Damit wissen wir jetzt, wie wir Binärzahlen elektromechanisch addieren können.

Konrad Zuse baute 1941 die erste **elektromechanische Rechenanlage Z3**. Das folgende Bild zeigt den Nachbau seiner Z3 im Deutschen Museum in München. Die erste, eigentliche Z3 wurde leider bei einem Bombenangriff auf Berlin zerstört.



[von Venusianer, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=3632073]

Die Z3 kann mit Hilfe von Relais schalten und addieren. Das geschieht in der einen Hälfte der Maschine, dem sogenannten **Rechenwerk**. Die Relais werden aber nicht nur verwendet, um die Logik- und Berechnungsfunktionen der Z3 auszuführen. Relais behalten ihren Zustand bei, solange keine äußeren Einflüsse auf sie einwirken. Daher können die Relais in der Z3 den Zustand ihrer Schaltkreise speichern, was es dem Computer ermöglicht, Ergebnisse für spätere Rechnungen abzulegen und wiederzuverwenden. Das **Speicherwerk**, die zweite Hälfte der Z3, kann mit dieser Technik 64 Zahlen speichern.

Die Z3 verfügt über ein Pult, über das man eine **Eingabe** machen kann. Diese Eingabe wird dann in eine Dualzahl umgerechnet und in den Speicher geladen. Eingaben und Zahlen aus dem Speicher können ins Rechenwerk geladen und dort addiert werden. Und wir wissen ja bereits: Wenn wir Zahlen speichern und addieren können, können wir auch die Multiplikation und andere Rechnungen als eine Folge von Additionen darstellen. Die Z3 kann addieren, subtrahieren, multiplizieren, dividieren und die Wurzel ziehen. Dabei müssen die verschiedenen Schritte der Rechnungen nicht mehr stückweise oder manuell übertragen werden. Die Z3 kann die Multiplikation zweier Zahlen direkt in mehrere Schritte zerlegen und eigenständig durchführen. Wenn wir aber eine Multiplikation aus mehreren Additionen zusammensetzen wollen, müssen wir festlegen, wann eine Addition fertig ist und wann die nächste beginnen kann. Bei unseren mechanischen Maschinen war das klar, weil jemand Zahnräder mit der Hand drehte oder Reihe für Reihe Zwischenergebnisse übertrug. Dadurch wurde die Geschwindigkeit festgelegt. In Zuses Z3 wird der **Takt** über einen kleinen Elektromotor gesteuert. Dieser gibt das Tempo vor, mit dem sich alle Schaltungen bewegen. Der Takt sorgt dafür, dass immer nach 0,2 Sekunden eine neue Rechnung beginnt. Innerhalb dieser Zeit können alle Relais der Z3 einmal ihre Position verändern. Die Z3 kann damit fünf Additionen pro Sekunde durchführen. Die Taktfrequenz der Z3 ist also genau 5 Hz. Eine Multiplikation dauert 16 dieser Takte. Heute haben gängige Prozessoren eine Taktfrequenz von 2,5 GHz bis 5 GHz. Damit kann solch ein Prozessor 2,5 bis 5 Milliarden Rechenoperationen pro Sekunde durchführen.

Addition, Subtraktion, Multiplikation, Division, Radizieren, Speichern, Laden, Einlesen und Ausgeben sind in die Z3 „eingebaut". Das sind die neun Befehle, die die Z3 direkt ausführen kann. In der Informatik nennen wir das die **Maschinensprache** eines Computers. Um jetzt komplexere Rechnungen und Aufgaben durchführen zu können, verfügt die Z3 über einen zusätzlichen **Lochstreifenleser**. Dieser kann eine Reihe von vorgestanzten Befehlen der Maschinensprache lesen und abarbeiten. Das bedeutet, dass keine Eingaben über den Streifen gelesen werden, sondern eine Liste von Befehlen, die dann beliebige Eingaben verrechnen kann. Die Z3 kann also Algorithmen, die sich aus ihren neun Befehlen zusammensetzten, vom Lochstreifen ausführen und für beliebige Eingaben abarbeiten. So benutzte Konrad Zuse die Z3 zum Beispiel für Berechnungen über die Festigkeit von Flugzeugflügeln. Deshalb spricht man bei der Z3 vom ersten **programmierbaren Rechner** – ein Meilenstein in der Geschichte der Informatik!

Auf YouTube finden Sie ein kurzes Video des Deutschen Museums, das den Nachbau der Z3 in Aktion zeigt. Es trägt den Titel „Die Z3 von Konrad Zuse im Deutschen Museum". Wenn Sie wollen, schauen Sie sich das gerne einmal an.

Aber ist die Z3 der erste Computer? Diese Frage lässt sich nicht eindeutig beantworten. Wie moderne Computer nutzt die Z3 das duale Zahlensystem und ist programmierbar. Allerdings liest sie das Programm nur sequenziell von einem Lochstreifen ab. Moderne Computer laden das Programm in den Speicher und können beim Abarbeiten von Befehl zu Befehl springen. Dadurch lassen sich Schleifen und Verzweigungen zwischen den Befehlen während des Programmdurchlaufs umsetzen. Außerdem ist die Z3 elektromechanisch. Die Relais sind Schalter, die per Spannung geschaltet werden und hörbar klappern. Moderne Computer sind vollständig elektronisch, und die Schaltungen werden durch Transistoren realisiert. Wenn Sie Ihrem Computer beim

Rechnen zuhören, hören Sie nur den Lüfter – und die Festplatte nur, wenn Sie keine Solid State Disk (SSD) besitzen. Im Prozessor, also im Rechenwerk, bewegt sich heute nichts mehr.

Ab 1943 entwickeln **John Presper Eckert** und **John William Mauchly** an der University of Pennsylvania den **ENIAC** (Electronic Numerical Integrator and Computer). Statt Relais verwendet der ENIAC Elektronenröhren als Schaltelemente und ist damit der weltweit erste elektronische Rechner. Der ENIAC ist ebenfalls programmierbar, aber auf eine andere Weise als moderne Computer. Er wird durch das Umstecken von Kabeln und das Einstellen von Schaltern programmiert, anstatt durch die Eingabe von Befehlen. Diese Methode erfordert physische Änderungen an der Verdrahtung des Computers, um verschiedene Berechnungen ausführen zu können. Der ENIAC verwendet auch das alte dezimale Zahlensystem. Trotzdem kann der ENIAC beliebige Programme voll elektronisch ausführen. Je nachdem, ob wir die Z3 zählen oder nicht, ist der ENIAC der erste oder der zweite Computer in unserer Geschichte. Leider wird der ENIAC hauptsächlich für ballistische Berechnungen im Zusammenhang mit dem US-Militär eingesetzt.

Im Jahr 1949 entwickelt **Maurice Vincent Wilkes** an der University of Cambridge den **EDSAC** (Electronic Delay Storage Automatic Calculator). Dieser Rechner basiert auf dem von **John von Neumann** vorgeschlagenen Konzept, dass der Speicher nicht nur die Daten, sondern auch die Programme enthalten soll. Programme werden auf Lochkarten geschrieben und dann zunächst von der Lochkarte in den Speicher geladen. Die gespeicherten Programme werden dann Befehl für Befehl ins Rechenwerk transferiert und ausgeführt. Den aktuellen Befehl erkennt man anhand eines Befehlszählers. Am Ende jedes Befehls wird dieser um eins erhöht. Es gibt aber auch Sprungbefehle, die den Inhalt des Befehlszählers ändern. Ebenso gibt es Verzweigungsbefehle, die in Abhängigkeit vom Wert eines Entscheidungsbits den Befehlszähler entweder um eins erhöhen oder einen Sprungbefehl ausführen.

[von Computer Laboratory University of Cambridge, CC BY 2.0, https://commons.wikimedia.org/w/index.php?curid=432919]

Der EDSAC ist damit der weltweit erste Computer, der gespeicherte Programme nutzt. Ab dem Jahr 1950 gibt es in der Informatik somit zwei Entwicklungsstränge: **Hardware** und **Software**. Hardware bezeichnet die physische, greifbare Ausrüstung eines Computers, wie Prozessoren, Speicher und Eingabegeräte, die miteinander verbunden sind, um Rechenoperationen auszuführen. Software hingegen besteht aus den Programmen und Anweisungen, die auf der Hardware laufen, und steuert die Funktionsweise des Computers, indem sie ihm spezifische Aufgaben und Prozesse vorgibt.

Insgesamt wissen wir jetzt, was ein moderner Computer ist und wie er sich von seinen Vorgängern unterscheidet. Ein moderner Computer verwendet das duale Zahlensystem, setzt auf Elektronik statt Mechanik, ist nach der von-Neumann-Architektur aufgebaut und kann damit Programme laden, speichern und ausführen. Mittlerweile nutzen wir hochintegrierte Schaltkreise und haben mehrere Prozessoren auf einem Chip. Heute haben wir selbstverständlich Zugriff auf PCs (Personal Computers), Laptops, Tablets und Smartphones. Computer führen Milliarden von Operationen pro Sekunde aus, Programme laden wir über das Internet aus dem App-Store in den Speicher, und tragen Smartphones, kleine mobile Computer, in unseren Hosentaschen mit uns herum.

1.3 Was sind Daten?

Wir haben gerade gesehen und gelernt, wie elektronische Rechenmaschinen natürliche Zahlen als Binärzahlen darstellen und mit diesen Rechnungen ausführen. Wir wissen aber, dass Computer nicht nur in der Lage sind Zahlen zu verarbeiten. Computer speichern Texte, Listen, Adressen, Termine, Bilder, Fotos, Musik, Videos und vieles mehr.

In der Informatik repräsentieren wir die Aspekte unserer realen Welt, die wir für die jeweilige Anwendung oder Rechnung benötigen, als **Daten** in den Speichern unserer Computer. Der Computer führt dann auf diesen Daten Berechnungen aus, um die Daten zu manipulieren und Ergebnisse zu erzeugen. In diesem Abschnitt überlegen wir uns, wie wir die Dinge dieser Welt in den Speicher unseres Computers bekommen.

1.3.1 Zahlen

Wenn wir zwei Zahlen mit einem Rechner addieren wollen, müssen wir die Eingabe zunächst in unserem Rechner speichern. Dazu brauchen wir Platz. Umso größer die Eingabe, umso mehr Platz benötigen wir. Ein Programm im Rechner soll dann für zwei beliebige Eingaben die Summe berechnen, doch ist das technisch wirklich möglich? Denn physischer Speicher in unserem Computer steht immer nur begrenzt zur Verfügung. Da es unendlich viele natürliche Zahlen gibt, müssen wir uns bei den möglichen Eingaben immer einschränken. Das ist im Grunde genauso wie mit den Zahnrädern der ersten Rechenmaschinen. Wir können immer mehr Zahnräder hinzufügen, aber nicht unendlich viele. Je mehr Speicher wir haben, desto größere Zahlen können wir darstellen. Da wir mit der Darstellung immer nur einen begrenzten Wertebereich abdecken können und Speicher Geld kostet, macht es Sinn, sich zu überlegen, wie wir Zahlen möglichst effizient speichern. In diesem Abschnitt lernen wir deshalb verschiedene Darstellungen von Zahlen kennen.

Unsere Addiermaschine aus dem letzten Abschnitt hat sechs Zahnräder mit den Ziffern 0 bis 9. Damit kann man die natürlichen Zahlen 0 bis 999999 darstellen. Um den gleichen Wertebereich als Dualzahl abzubilden müssen wir die Zahl $999999 = (11110100001000111111)_{bin}$ in unseren Speicher bekommen. Das bedeutet wir benötigen 20 Dualziffern, um eine natürliche Zahl zwischen 0 und 999999 abzuspeichern. Jede Dualziffer benötigt eine Speicherzelle. In der Informatik nennen wir das, was man mit einer Speicherzelle mit zwei Zuständen speichern kann ein **Bit**. Wir müssen also 20 Bit Speicher reservieren, um natürliche Zahlen bis zu einer Größe von 999999 darstellen zu können.

Nur damit keine Missverständnisse aufkommen: Das „B“ in MB (Megabyte) oder GB (Gigabyte) steht für **Byte**, nicht für Bit. In unseren Computern sind 8 Speicherzellen zu einem Byte zusammengefasst. Ein Byte besteht also aus 8 Bits. 1024 Byte ergeben ein Kilobyte, 1024 Kilobyte ein Megabyte, 1024 Megabyte ein Gigabyte und so weiter. Damit ergibt sich

$$1\,GB = 1.024\,MB = 1.048.576\,KB = 1.073.741.824\,Bytes = 8.589.934.592\,Bits$$

Natürliche Zahlen können wir einfach in Binärzahlen umrechnen, um diese zu speichern. Leider kommen wir in den wenigsten Anwendungen mit den natürlichen Zahlen aus. Überlegen wir uns zunächst, wie wir mit negativen Zahlen umgehen können. Also, wie speichern wir einen Bereich der ganzen Zahlen? Im Grunde gibt es bei den ganzen Zahlen zwei Arten: positive und negative ganze Zahlen. Wenn wir eine ganze Zahl schreiben, setzen wir entweder ein Minuszeichen oder nicht. Denken wir in Bit, benötigen wir also genau ein Bit, um das Vorzeichen zu speichern.

Wenn wir ein Bit spendieren, um eine natürliche Zahl zusammen mit einem Vorzeichen zu speichern, nennen wir das die **Vorzeichendarstellung** einer ganzen Zahl. Wir verabreden, dass 0 für eine positive Zahl und 1 für eine negative Zahl steht. Die folgende Tabelle zeigt den Wertebereich -7 bis 7 und die Darstellung jeder dieser Zahlen in 4-Bit Vorzeichendarstellung.

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

Schauen wir uns nun an, wie weit wir mit 16 Bits kommen, um ganze Zahlen zu speichern.

$$[0\ 111\ 1111\ 1111\ 1111]_{\text{vz}} = +32767$$

$$[1\ 111\ 1111\ 1111\ 1111]_{\text{vz}} = -32767$$

Mit der 16-Bit-Vorzeichendarstellung decken wir den Wertebereich $\{-32767, \dots, 32767\}$ ab.

Doch ist die Vorzeichendarstellung wirklich praktisch? Um das zu überprüfen, schauen wir uns an, wie wir die wichtigste Operation, die Addition, in der Vorzeichendarstellung durchführen können. Dazu verwenden wir in diesem Abschnitt eine weitere Schreibweise. Wir benutzen den Operator $\oplus$, um zu zeigen das zwei Bitketten durch eine passende Schaltung addiert werden. Im Grunde steht $\oplus$ für die Schaltung auf Seite 22 dieses Lehrtextes.

Addieren wir jetzt zwei Zahlen in der Vorzeichendarstellung. Zuerst prüfen wir die zwei ersten Bits unserer Eingaben. Sind beide Bits 0, haben wir zwei positive Zahlen und können die folgenden Bits wie gewohnt addieren.

$$3 + 2 = [0011]_{vz} + [0010]_{vz} = 0\,[011 \oplus 010] = 0\,[101] = [0101]_{vz} = 5$$

Sind beide ersten Bits 1, haben wir zwei negative Zahlen. Auch hier können wir einfach addieren.

$$(-3) + (-2) = [1011]_{vz} + [1010]_{vz} = 1\,[011 \oplus 010] = 1\,[101] = [1101]_{vz} = -5$$

Sind die Vorzeichen gleich und verlassen wir den Wertebereich nicht, ist die Addition sehr leicht umsetzbar.

Was passiert, wenn die beiden Zahlen unterschiedliche Vorzeichen haben? Sagen wir die erste Zahl ist negativ und die zweite Zahl ist positiv. Wir können dann nicht einfach addieren. Was machen wir stattdessen? Wir betrachten zuerst die Beträge der zu addierenden Zahlen. Die vom Betrag her größere Zahl kommt nach vorne. Dann ziehen wir die zweite Zahl von der ersten ab. Mussten wir die Zahlen tauschen, ist das Vorzeichen des Ergebnis positiv. Stand die vom Betrag her größere Zahl bereits vorne, ist das Ergebnis negativ. Wir müssen die Vorzeichen prüfen, dann Beträge vergleichen, tauschen und das Vorzeichen des Ergebnis anpassen. Insbesondere brauchen wir eine weitere Schaltung $\ominus$, die Bitketten subtrahiert.

$$(-3) + 2 = [1011]_{vz} + [0010]_{vz} = 1\,[011 \ominus 010] = 1\,[001] = [1001]_{vz} = -1$$

$$(-3) + 4 = [1011]_{vz} + [1100]_{vz} = 0\,[100 \ominus 011] = 0\,[001] = [0001]_{vz} = 1$$

Was passiert, wenn die erste Zahl positiv ist und die zweite Zahl negativ? Auch hier müssen wir vergleichen, tauschen und $\ominus$ benutzen.

$$2 + (-3) = [0010]_{vz} + [1011]_{vz} = 1\,[011 \ominus 010] = 1\,[001] = [1001]_{vz} = -1$$

$$4 + (-3) = [0100]_{vz} + [1011]_{vz} = 0\,[100 \ominus 011] = 0\,[001] = [0001]_{vz} = 1$$

Wenn man sich das so überlegt, erscheint es seltsam, wie schwierig die Addition von ganzen Zahlen eigentlich ist. Das gilt übrigens unabhängig davon, ob wir Binärzahlen oder Dezimalzahlen addieren. Allerdings machen wir uns darüber in der Regel keine Gedanken, da wir das seit der Grundschule kennen und unterstützt durch den Zahlenstrahl, irgendwann einfach akzeptiert haben.

Für uns Menschen ist die Vorzeichendarstellung sehr intuitiv zu lesen. Die natürliche Zahl beschreibt den Abstand zur Null und das Vorzeichen gibt die Richtung an in der wir auf dem Zahlenstrahl unterwegs sind.

Innerhalb eines Rechners ist gute Lesbarkeit kein sinnvolles Argument. Hier geht es lediglich darum, dass wir Zahlen effizient verarbeiten können. Um hier die zentralen Ideen zu verstehen, betrachten wir jetzt eine alternative mögliche Darstellung der ganzen Zahlen, mit der sich sehr effizient rechnen lässt.

In der **Zweierkomplementdarstellung** stellen wir alle positiven Zahlen wie in der Vorzeichendarstellung als Binärzahlen mit einer führenden 0 dar. An dieser Stelle ändert sich also nichts. Etwas Neues kommt bei der Darstellung der negativen Zahlen hinzu. Um eine negative Zahl im Zweierkomplement darzustellen, berechnen wir zunächst die zugehörige positive Zahl und **kippen** dann jedes Bit. Kippen bedeutet, dass wir jede 1 durch eine 0 und jede 0 durch eine 1 ersetzen. Wir nennen die gekippte Bitkette das **Komplement**. Schließlich **addieren** wir noch **Eins** auf das Komplement, um die Zweierkomplementdarstellung zu erhalten.

Die folgende Tabelle zeigt den Wertebereich -8 bis 7 und die Darstellung jeder dieser Zahlen in 4-Bit Zweierkomplementdarstellung. Die Zahl 7 ist immer noch die Bitkette 0111, hier ändert sich nichts. Kippen wir die 0111, erhalten wir 1000. Damit ist die Bitkette $1000 + 1$, also die 1001, die Zahl -7. Genau wie bei der Vorzeichendarstellung gibt das erste Bit wieder das Vorzeichen an. Der abgedeckte Wertebereich ist aber um eine Zahl gewachsen, da wir die 0 nicht mehr doppelt codieren, können wir die Zahl -8 als die Bitkette 1000 darstellen. Die Reihenfolge der positiven Zahlen bleibt unverändert, die der negativen Zahlen ist anders.

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

Die Sortierung der negativen Zahlen in der Zweierkomplementdarstellung sieht zuerst ungewohnt aus, ist aber sehr praktisch, wenn wir Zahlen addieren wollen. Schauen wir uns dazu die untere Zeile der Tabelle der 4-Bit Zweierkomplementdarstellung genauer an. Beginnen wir ganz Links und laufen nach rechts, sehen wir, dass wir die je um Eins größere Zahl bekommen, wenn wir auf die Bitkette 1 addieren. Aus 1000 wird 1001, aus 1001 wird 1010, aus 1010 wird 1011, und so weiter. Die 1 addieren, ist ein Feld nach rechts gehen, völlig unabhängig davon, ob wir uns links oder rechts der 0 befinden. Selbst wenn wir von der 1111 zur 0000 laufen, ist das die 1 addieren, wenn wir den Überlauf einfach ignorieren.

Betrachten wir nochmal die Beispielrechnungen, die wir für die Vorzeichendarstellung durchgeführt haben. Der leichteste Fall ist sicherlich, wenn beide Eingaben positiv sind. Hier stimmt die Zweierkomplementdarstellung mit der Vorzeichendarstellung überein. Hier ändert sich nichts. Um die 2 auf die 3 zu addieren, laufen wir in der Wertetabelle von der 0011 zwei Felder nach rechts. Zwei Felder nach rechts, das ist die 0010 addieren.

$$3 + 2 = [0011]_{2k} + [0010]_{2k} = [0011 \oplus 0010] = [0101]_{2k} = 5$$

Was ist, wenn die erste Zahl negativ ist? Um die 4 auf die -3 zu addieren, laufen wir in der Wertetabelle von der 1101 vier Felder nach rechts. Vier Felder nach rechts, das ist die 0100 addieren.

$$(-3) + 4 = [1101]_{2k} + [0100]_{2k} = [1101 \oplus 0100] = [0001]_{2k} = 1$$

Bei dieser Rechnung bekommen wir bei $[1101 \oplus 0100] = [0001]_{2k}$ einen Überlauf. Aber in der Zweierkomplementdarstellung kommen wir ja von der 1111 zur 0000. Wir können den Überlauf also einfach ignorieren. Wir können die letzte Lampe unserer Schaltung auf Seite 22 einfach abbauen um $\oplus$ zu realisieren. Dadurch wird alles nur noch einfacher.

Insgesamt können wir also ignorieren, dass die erste Zahl negativ ist und können auch unsere normale $\oplus$ Schaltung zum Addieren einer positiven Zahl auf eine negative Zahl verwenden.

Aber auch für das **Subtrahieren**, also das addieren von negativen Zahlen, in Zweierkomplementdarstellung haben wir noch eine wirklich gute Idee. Schauen wir noch einmal auf die Wertetabelle der Zweierkomplementdarstellung. Wenn wir auf die 0111 eine 1 addieren, springen wir zur 1000. Wir laufen im Kreis! Anstatt 1 Feld nach Links zu laufen, können wir auch 15 Felder nach rechts laufen. Anstatt eine 1 zu subtrahieren, addieren wir eine 15. Anstatt eine 3 zu subtrahieren, addieren wir eine 13.

1 Schritt nach links, sind 15 Schritte nach rechts. 15 Schritte nach rechts ist 15 mal die 1 addieren. Anstatt $a - 1$, rechnen wir $a + 15$.

3 Schritte nach links, sind 13 Schritte nach rechts. 13 Schritte nach rechts ist 13 mal die 1 addieren. Anstatt $a - 3$, rechnen wir $a + 13$.

b Schritte nach links, sind $2^n - b$ Schritte nach rechts. $2^n - b$ Schritte nach rechts ist $2^n - b$ mal die 1 addieren. Anstatt $a - b$, rechnen wir $a + (2^n - b)$.

Was ist jetzt dieses $2^n - b$? Also $2^n - 1$ ist die Bitkette die nur aus Einsern besteht. Wenn wir von dieser Bitkette b abziehen, bekommen wir das gekippte b und nennen es $\bar{b}$. Das können sie gerne ausprobieren, oder sich überlegen, dass $b + \bar{b}$ nur aus Einsern besteht. Insgesamt bekommen wir

$$a - b = a + (2^n - b) = a + (2^n - 1 - b) + 1 = a + \bar{b} + 1$$

Schauen wir uns jetzt den letzten Ausdruck $\bar{b} + 1$ noch einmal an. In der Zweierkomplementdarstellung ist dieses Kippen und Eins addieren einfach der Wechsel von der positiven auf die negative Zahl. Das heißt, wir können subtrahieren, indem wir die Zweierkomplementdarstellung addieren.

$$2 + (-3) = [0010]_{2k} + [1101]_{2k} = [0010 \oplus 1101] = [1111]_{2k} = -1$$

$$4 + (-3) = [0100]_{2k} + [1101]_{2k} = [0100 \oplus 1101] = [0001]_{2k} = 1$$

An diesen Beispielen sehen wir noch einmal ganz deutlich, dass wir jetzt keine $\ominus$ Schaltung mehr brauchen. Wir müssen auch die ersten Bits nicht prüfen und keine Fälle mehr unterscheiden. Wir ignorieren einfach, dass die zweite Zahl negativ ist und addieren wie gewohnt.

Auch im Bereich der negativen Zahlen funktioniert unsere Subtraktion.

$$(-2) + (-3) = [1110]_{2k} + [1101]_{2k} = [1110 \oplus 1101] = [1011]_{2k} = -5$$

Durch die Darstellung der negativen Zahlen in der Zweierkomplementdarstellung geht zwar die Intuition mit dem Abstand zur 0 verloren, aber alle Zahlen sind jetzt in einem Kreis angeordnet, in dem wir durch Addieren positiver Zahlen vorwärtslaufen können, egal wo wir uns befinden. Anstatt zu subtrahieren, nutzen wir aus, dass wir die notwendigen Rückwärtsschritte in Vorwärtsschritte übersetzen können, da wir mit 2^n Schritten genau einmal im Kreis laufen. Passenderweise wird die Übersetzung von Vorwärts auf Rückwärts genau durch das Kippen und das Addieren von Eins realisiert, was witzigerweise bereits in der Codierung enthalten ist.

Schauen wir uns das Rechnen mit der Zweierkomplementdarstellung noch einmal an etwas größeren Beispielen an. Für die folgenden Rechnungen übersetzen wir uns zunächst einige Zahlen in die Zweierkomplementdarstellung und stellen uns vor, dass diese in unserem Computer im Speicher liegen und warten. Zur besseren Lesbarkeit stellen wir die Folgen von Bits in Viererblöcken dar.

$$
\begin{aligned}
1337 &= [0000\ 0101\ 0011\ 1001]_{2k} \\
-1337 &= [1111\ 1010\ 1100\ 0111]_{2k} \\
\\
42 &= [0000\ 0000\ 0010\ 1010]_{2k} \\
-42 &= [1111\ 1111\ 1101\ 0110]_{2k} \\
\\
1379 &= [0000\ 0101\ 0110\ 0011]_{2k} \\
-1379 &= [1111\ 1010\ 1001\ 1101]_{2k} \\
\\
1295 &= [0000\ 0101\ 0000\ 1111]_{2k} \\
-1295 &= [1111\ 1010\ 1111\ 0001]_{2k}
\end{aligned}
$$

In der folgenden Tabelle sehen wir die Addition der Zahlen $1337 + 42$. Das Ergebnis ist die Darstellung der Zahl 1379.

	0	0	0	0	0	1	0	1	0	0	1	1	1	0	0	1
$\oplus$	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
										1	1	1				
	0	0	0	0	0	1	0	1	0	1	1	0	0	0	1	1

In der nächsten Tabelle sehen wir die Addition der Zahlen $-1337 + 42$. Das Ergebnis ist die Darstellung der Zahl -1295. Um das nachzuvollziehen, müssen sie von der Ergebnisbitkette 1 abziehen und dann alle Bits kippen. Wir können in der Zweierkomplementdarstellung auch im Bereich der negativen Zahlen addieren.

	1	1	1	1	1	0	1	0	1	1	0	0	0	1	1	1
$\oplus$	0	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0
												1	1	1		
	1	1	1	1	1	0	1	0	1	1	1	1	0	0	0	1

In der nächsten Tabelle sehen wir eine Subtraktion. Wir rechnen dort $1337 - 42$, indem wir die 1337 und die -42 addieren. Das Ergebnis ist die Darstellung der Zahl 1295. Da wir in der Addition einen Übertrag an der höchsten Stelle bekommen, sind wir über die 0 gelaufen. Wir können diesen Überlauf einfach ignorieren.

	0	0	0	0	0	1	0	1	0	0	1	1	1	0	0	1
$\oplus$	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0
1	1	1	1	1	1	1	1	1	1	1	1					
	0	0	0	0	0	1	0	1	0	0	0	0	1	1	1	1

Nur der Vollständigkeit halber, rechnen wir in der folgenden Tabelle noch einmal $-1337 - 42$, indem wir die -1337 und die -42 addieren.

	1	1	1	1	1	0	1	0	1	1	0	0	0	1	1	1
$\oplus$	1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0
1	1	1	1	1	1	1	1	1	1				1	1		
	1	1	1	1	1	0	1	0	1	0	0	1	1	1	0	1

In der Vorzeichendarstellung mussten wir bei der Addition ganzer Zahlen verschiedene Fälle unterscheiden, wobei wir manchmal addieren und manchmal subtrahieren mussten. In der Zweierkomplementdarstellung können wir einfach direkt addieren. Da wir dadurch nicht nur Zeit, sondern auch Kupfer, also eine Schaltung für das Subtrahieren, sparen, hat sich die Zweierkomplementdarstellung als Standard in unseren Rechnern durchgesetzt.

Nur der Vollständigkeit halber möchte ich noch die **Einerkomplementdarstellung** erwähnen, da sie häufig in Lehrbüchern zur Informatik zu finden ist. Bei dieser Methode wird die negative Zahl gebildet, indem wir einfach die Bits der positiven Zahl kippen, aber dann keine Eins addieren. Dadurch wird die Umrechnung von Dezimalzahlen in Einerkomplement schneller, allerdings wird dann die Zahl 0 im Wertebereich doppelt dargestellt, einmal als positive und einmal als negative Null. Dies führt dazu, dass wir Überträge beim Addieren nicht mehr einfach ignorieren können, und das insgesamt wieder mehr gerechnet werden muss.

Damit haben wir für die Darstellung natürlicher Zahlen erst einmal genügend Auswahl. Können wir mit dieser Technik auch die **rationalen Zahlen**, also die Menge aller Brüche, darstellen?

Eine rationale Zahl ist der Quotient zweier natürlicher Zahlen. Zusätzlich kann ein Bruch positiv oder negativ sein. Damit liegt es nahe eine rationale Zahl als Paar einer ganzen und einer natürlichen Zahl darzustellen.

Geben wir uns wieder 16-Bit Speicherplatz, um eine rationale Zahl darzustellen, können wir zum Beispiel die ersten 12-Bit verwenden, um den Zähler als ganze Zahl in Zweierkomplementdarstellung darzustellen, und die letzten 4-Bit verwenden, um den Nenner als Binärzahl darzustellen.

Nach diesen Spielregeln zeigt die folgende Bitkette die ganze Zahl $-1337/15$.

$$-1337/15 = [1010\ 1100\ 0111\ 1111]_{2k(12)/bin(4)}$$

Die ersten 12 Bits sind die $1010\ 1100\ 0111$, die -1337 in Zweierkomplementschreibweise, die letzten 4 Bits sind die binäre Zahl 15.

Schauen wir uns unsere Darstellung kurz genauer an. Berechnen wir die Größte und die Kleinste darstellbare Zahl und schauen wir in welchen Schritten wir über den abgedeckten Zahlenbereich laufen können.

$$[0111\ 1111\ 1111\ 0001]_{2k(12)/bin(4)} = 2047$$

$$[1000\ 0000\ 0000\ 0001]_{2k(12)/bin(4)} = -2048$$

$$[0000\ 0000\ 0000\ 1111]_{2k(12)/bin(4)} = 0$$

$$[0000\ 0000\ 0001\ 1111]_{2k(12)/bin(4)} = 1/15$$

$$[0111\ 1111\ 1111\ 1111]_{2k(12)/bin(4)} = 2047/15$$

$$[1111\ 1111\ 1111\ 1111]_{2k(12)/bin(4)} = -1/15$$

$$[0000\ 0000\ 0011\ 1110]_{2k(12)/bin(4)} = 3/14$$

Mit unsere Quotientendarstellung können wir also rationale Zahlen im Wertebereich von -2048 bis 2047 abdecken. Bis zur Zahl $2047/15$, ungefähr 136, können wir in $1/15$er Schritten gehen. Wir können aber auch einige Zahlen zwischen den $1/15$er Schritten abbilden. So liegt zum Beispiel $3/14$ zwischen $3/15$ und $4/15$. Das heißt die Abstände zwischen den Zahlen, die wir darstellen können, sind unterschiedlich groß.

Zusätzlich können wir viele der Zahlen durch verschiedene Brüche darstellen und verschwenden dadurch Speicherplatz.

$$[0000\ 0000\ 0001\ 0011]_{2k(12)/bin(4)} = 1/3$$

$$[0000\ 0000\ 0010\ 0110]_{2k(12)/bin(4)} = 2/6$$

$$[0000\ 0000\ 0011\ 1001]_{2k(12)/bin(4)} = 3/9$$

Natürlich können wir nicht alle rationalen Zahlen im Bereich von -2048 bis 2047 abdecken, da es unendlich viele davon gibt. Wenn wir mit dieser Zahlendarstellung arbeiten und rechnen, kann es leicht passieren, dass wir das Ergebnis einer Rechnung nicht exakt darstellen können. Das liegt natürlich wieder daran, dass wir mit endlichem Speicher nur endlich viele Zahlen darstellen können. Bei den rationalen Zahlen gibt es innerhalb jedes festgelegten Intervalls unendlich viele Werte. Wir werden oft **runden** müssen.

Wir runden, wenn wir den Quotient aus 3 und 2 als ganze Zahl abspeichern, oder wenn wir die Zahl $1/16$ in unserer $2k(12)/bin(4)$-Darstellung ablegen wollen. Wir runden immer dann, wenn der Wertebereich, mit dem wir arbeiten, nicht zu der Zahl passt, die wir speichern wollen.

In vielen Anwendungen ist das Runden unproblematisch, solange wir wissen, wie und um wie viel gerundet wird. So können wir mögliche Rundungsfehler in einer Lösung einkalkulieren und mit ihnen arbeiten. Banken berechnen Zinsen in der Regel mit hoher Präzision, oft auf viele Dezimalstellen genau. Der Endbetrag wird jedoch häufig nur auf zwei Dezimalstellen gerundet, wenn er dem Kunden angezeigt wird. Dieser Rundungsfehler bleibt in der Regel gering, kann aber bei langfristigen Berechnungen oder großen Beträgen signifikant werden. In der **Festkommadarstellung** rationaler Zahlen ist es besonders leicht Rundungsfehler zu verstehen und abzuschätzen.

Um die Idee der **Festkommadarstellung** zu verstehen, beschränken wir uns zunächst auf die positiven rationalen Zahlen und geben uns wieder 16-Bit Speicherplatz. Wir beschreiben mit den ersten 12-Bits eine Binärzahl, dann denken wir uns ein Komma und benutzen die letzten 4-Bits für Nachkommastellen. Bei Binärzahlen hat die erste Stelle nach dem Komma die Wertigkeit $1/2$. Die zweite Stelle nach dem Komma die Wertigkeit $1/4$. Die dritte Stelle die Wertigkeit $1/8$ und die letzte die Wertigkeit $1/16$.

Nach diesen Spielregeln zeigt die folgende Bitkette die rationale Zahl $1337{,}9375$.

$$1337{,}9375 = [0101\ 0011\ 1001\ 1111]_{bin(12)/bin(4)}$$

In der Festkommadarstellung stellen wir keine Zahlen doppelt dar. Dadurch dass wir in äquidistanten Schritten über einen Zahlenbereich laufen, können wir den Rundungsfehler, den wir innerhalb des darstellbaren Intervalls machen, immer wunderbar abschätzen.

In der $bin(12), bin(4)$-Darstellung machen wir zwischen den Zahlen 0 und 4095 immer Schritte der Länge $(0{,}0001)_{bin}$, also $1/16 = 0{,}0625$. Der maximale Rundungsfehler, falls wir immer nur abrunden, ist also immer echt kleiner als diese Zahl.

In der $bin(8), bin(8)$-Darstellung machen wir zwischen den Zahlen 0 und 255 immer Schritte der Länge $(0{,}00000001)_{bin}$, also $1/256 = 0{,}00390625$.

Rechnen können wir in der Festkommadarstellung genau wie mit binären Zahlen. Wollen wir negative rationale Zahlen darstellen, benutzen wir wieder ein Bit für das Vorzeichen, oder eine Komplementdarstellung. Wie wir bereits wissen, unterscheiden sich diese Darstellungen für positive Zahlen nicht.

$$1337{,}1337 \approx [0101\ 0011\ 1001\ 0010]_{2k(12)/2k(4)} = 1337{,}125$$

Wie bekommen wir $-1337{,}125$? Invertieren wir die Bitkette erhalten wir $1010\ 1100\ 0110\ 1101$. Für die Zweierkomplementdarstellung addieren wir jetzt wieder eine 1 an der letzten Stelle und erhalten demnach die folgende Darstellung.

$$-1337{,}125 = [1010\ 1100\ 0110\ 1110]_{2k(12)/2k(4)}$$

Insgesamt können wir mit einer Festkommadarstellung die rationalen Zahlen gut darstellen und mögliche Rundungsfehler immer leicht abschätzen.

Neben den Vorteilen der Festkommadarstellung gibt es jedoch einen gravierenden Nachteil. Für große Zahlen sind die Rundungsfehler relativ gering. Der Unterschied zwischen 1337 und $1337{,}0625$ mag für viele praktische Anwendungen nicht groß ins Gewicht fallen. Der gleiche Unterschied ist für kleine Zahlen relativ betrachtet jedoch viel höher. Die Zahl $0{,}125$ ist doppelt so groß wie die Zahl $0{,}0625$. Noch schlimmer ist es meist, wenn zum Beispiel die $0{,}05$ direkt zur 0 abgerundet wird. Wir können immer versuchen mehr Speicher und mehr Nachkommastellen zu spendieren, aber da wir dann auch für große Zahlen sehr kleine Schritte machen, wäre das meist Verschwendung von Speicherplatz.

In vielen Anwendungen hat es Sinn den **relativen Rundungsfehler** zu betrachten. Diesen definieren wir als Quotient des Rundungsfehlers und der gerundeten Zahl. Runden wir zum Beispiel in der $bin(12), bin(4)$-Darstellung die $0{,}1337$ auf die $0{,}125$ machen wir einen relativen Rundungsfehler von

$$\frac{0{,}1337 - 0{,}125}{0{,}1337} \approx 0{,}065$$

Runden wir von der $1337{,}1337$ auf die $1337{,}125$ machen wir einen relativen Rundungsfehler von

$$\frac{1337{,}1337 - 1337{,}125}{1337{,}1337} \approx 0{,}0000065$$

Um den relativen Rundungsfehler auszugleichen, betrachten wir jetzt die Idee, für kleine Zahlen mehr Nachkommastellen zu speichern als für große.

Bei der **Gleitkommadarstellung** speichern wir einfach die ersten Ziffern einer Zahl, ohne darauf zu achten, ob es sich dabei um Vorkommastellen oder um Nachkommastellen handelt. Diese einfache Idee führt dazu, dass wir den relativen Rundungsfehler über den Zahlenbereich ausgleichen.

Schauen wir uns die Gleitkommadarstellung kurz an zwei Beispielen und zunächst in Dezimalschreibweise an. Wenn wir die Zahl $1337{,}1337$ darstellen möchten, verschieben wir das Komma zunächst 4 Stellen nach vorne. Dadurch bekommen wir eine Zahl zwischen 0 und 1. In diesem Beispiel die Zahl $0{,}13371337$. Zusätzlich wissen wir, dass wir diese Zahl mit 10^4 multiplizieren müssen, um die ursprüngliche Zahl zu erhalten.

$$1337{,}1337 = 0{,}13371337 \cdot 10^4$$

Wenn wir die Zahl $0{,}0013371337$ darstellen möchten, verschieben wir das Komma zunächst 2 Stellen nach hinten. Wieder bekommen wir dadurch eine Zahl zwischen 0 und 1. In diesem Beispiel auch die Zahl $0{,}13371337$. Zusätzlich wissen wir, dass wir diese Zahl mit 10^{-2} multiplizieren müssen, um die ursprüngliche Zahl zu erhalten.

$$0{,}0013371337 = 0{,}13371337 \cdot 10^{-2}$$

Wir können jede rationale Zahl eindeutig auf diese Weise darstellen. Die einzige Ausnahme ist hier tatsächlich die Zahl 0. Wie wir mit der 0 umgehen, betrachten wir gleich noch in Ruhe. Alle anderen Zahlen können wir zunächst in ihre normierte Darstellung bringen und dann lediglich die Zahlen hinterm Komma, den Exponenten und ein Vorzeichen speichern.

Diese Art der normierten Darstellung funktioniert mit Binärzahlen genau so gut wie mit Dezimalzahlen. Eine Dezimalzahl verzehnfachen wir, um das Komma zu verschieben. Bei einer Binärzahl verdoppeln oder halbieren wir entsprechend.

$$101{,}101 = 0{,}101101 \cdot 2^3$$

$$0{,}00101101 = 0{,}101101 \cdot 2^{-2}$$

Hier sehen wir auch, dass bei Dezimalzahlen die erste Ziffer nach dem Komma eine Ziffer zwischen 1 und 9 sein kann. Denn wäre es die Ziffer 0, müssten wir das Komma weiter verschieben. Bei Binärzahlen bekommen wir mit dem gleichen Argument immer eine 1 als erste Nachkommastelle. Aus diesem Grund können wir jede normierte Binärzahl immer erst ab der zweiten Nachkommastelle speichern.

Fassen wir diese Ideen zusammen, sind wir jetzt bei der Standarddarstellung für Zahlen in unseren Computern angekommen. Diese Darstellung ist die **IEEE 754 Norm** mit dem sprechenden Namen **IEEE Standard for Floating-Point Arithmetic**. Dabei steht IEEE für Institute of Electrical and Electronic Engineers (www.ieee.org/).

Betrachten wir den **IEEE Standard for Floating-Point Arithmetic** in der 16-Bit Version. Das erste Bit ist das **Vorzeichen**. Dann haben wir 5 Bits, um einen **Exponenten** darzustellen. Danach bleiben uns 10 Bits, um die **Mantisse** unserer Zahl ab der zweiten Nachkommastelle abzuspeichern.

V	E	E	E	E	E	M	M	M	M	M	M	M	M	M	M

Bevor wir uns jetzt jedoch Beispiele ansehen, müssen wir kurz über die Zahl 0 sprechen. Wollen wir die 0 darstellen, bekommen wir diese niemals in unsere normalisierte Darstellung. Die folgende Gleichung hat keine Lösung, denn der erste und der zweite Faktor sind immer ungleich null.

$$0{,}1[Mantisse] \cdot 2^{[Exponet]} = 0$$

Jetzt ist aber gerade die 0 so wichtig, dass wir diese natürlich auch exakt darstellen können müssen. Also reservieren wir in der IEEE 754 Norm einen Sonderfall. Ist der Exponent die Bitkette 00000, ist das das Zeichen dafür, dass wir die gesamte Zahl **denormalisiert** darstellen. Denormalisiert bedeutet dabei, dass wir die Mantisse als Nachkommastellen einer Binärzahl verstehen die wir dann, per Definition, mit 2^{-14} multiplizieren.

$$0,[Mantisse] \cdot 2^{-14}$$

Das hat einen sehr schönen Effekt. Sind der Exponent und die Mantisse 0, bekommen wir durch die denormalisierte Darstellung wirklich unsere Zahl 0.

$$0,[0000000000] \cdot 2^{-14} = 0$$

Zusätzlich können wir in der denormalisierten Darstellung extra kleine Zahlen darstellen. So ist zum Beispiel die kleinste positive Zahl, die wir damit speichern können, die $(2^{-24})_{bin}$.

$$0,[0000000001] \cdot 2^{-14} = (2^{-24})_{bin}$$

Ähnlich zum Exponent 00000, ist auch der Exponent 11111 ein reservierter Sonderfall in der IEEE 754 Norm. Ist der Exponent 11111 und die Mantisse 0, steht das für die Zahl **unendlich**. Ist der Exponent 11111 und die Mantisse nicht 0, ist der Wert der Zahl **NaN**. Das ist die englische Abkürzung für „not a number“. In Computern benutzen wir diesen Wert, um das Ergebnis einer ungültigen Operation als ungültig zu kennzeichnen. Damit haben wir jetzt aber die Sonderfälle abgehakt und können uns der normalisierten Darstellung widmen.

In der **normalisierten** Darstellung bleiben uns demnach die Bitketten 00001 bis 11110 für die Darstellung des Exponenten der Gleitkommazahl übrig. Wir rechnen die Bitkette in eine Binärzahl um und subtrahieren 15, den sogenannten **Bias**, um den Zahlenbereich -14 bis 15 darstellen zu können. Für die Mantisse denken wir uns eine $0{,}1$ und speichern dann die folgenden Nachkommastellen mit den verbleibenden 10 Bits.

Berechnen wir als Beispiel zuerst die zwei größten Zahlen die wir in der 16 Bits IEEE 754 normalisiert darstellen können.

$$[0\ 11110\ 11\ 1111\ 1111]_{754} = (0{,}11111111111)_{bin} \cdot 2^{15} = (111111111110000)_{bin} = 32752$$

$$[0\ 11110\ 11\ 1111\ 1110]_{754} = (0{,}11111111110)_{bin} \cdot 2^{15} = (111111111100000)_{bin} = 32736$$

Der Abstand zwischen diesen beiden Zahlen sieht mit 16 zunächst recht groß aus. Der relative Abstand ist jedoch klein.

$$\frac{32752-32736}{32752} = \frac{16}{32752} \approx 0{,}0005$$

Die zwei kleinsten positiven Zahlen, die wir normalisiert darstellen können, bekommen wir mit dem Exponent -14.

$$[0\ 00001\ 00\ 0000\ 0000]_{754} = (0{,}1)_{bin} \cdot 2^{-14} = (2^{-15})_{bin} \approx 0{,}00003051$$

$$[0\ 00001\ 00\ 0000\ 0001]_{754} = (0{,}10000000001)_{bin} \cdot 2^{-14} \approx 0{,}00003054$$

Der relative Abstand ist damit ähnlich zum relativen Abstand der größten zwei Zahlen.

$$\frac{0{,}00003054-0{,}00003051}{0{,}00003054} \approx 0{,}001$$

Insgesamt ist es beachtlich wie viel wir mit 16 Bit erreichen können. Dadurch das auch der zur Verfügung stehende Speicher unserer Computer stark gewachsen ist, rechnen die meisten Computer heute standartmäßig in der 32 oder 64 Bit Variante der IEEE 754. Mit den 64 Bit decken wir dann einen Zahlenbereich von ungefähr -10^{308} bis 10^{308} ab.

1.3.2 Zeichen

Damit haben wir einen guten Überblick über die zentralen Ideen der Speicherung von Zahlen in unseren Computern. Doch sind das nicht die einzigen Daten, die wir in einem Computer speichern wollen. Diesen Lehrtext habe ich mit dem Computer geschrieben, er besteht aus einer Folge von Zeichen, nicht aus Zahlen.

Wir speichern Zeichen in unserem Computer, indem wir sie **codieren**. Codieren bedeutet dabei einfach, dass wir jedem Zeichen eine Bitkette zuordnen. Zum Beispiel könnten wir jeden Buchstaben durch seine Position im Alphabet darstellen. A wäre 1, B wäre 2, C wäre 3, und so weiter. Damit wäre die Zeichenfolge $H\ A\ L\ L\ O\ W\ E\ L\ T$ die Zahlenfolge 8 1 12 12 15 23 5 12 20. Dann speichern wir jede Zahlenfolge als Folge von Binärzahlen.

Natürlich kommen wir mit den Zeichen A bis Z nicht weit. Wir müssen auch Kleinbuchstaben, Sonderzeichen, Satzzeichen, Akzente, Zeichen verschiedener Sprachen und mittlerweile auch Emojis speichern. Wir machen das, indem wir die Codierung auf zusätzliche Zeichen erweitern. Umso mehr Zeichen wir abdecken wollen, umso mehr verschiedene Bitketten benötigen wir.

Bei der Codierung von Zahlen konnten wir den Code immer durch eine formale Abbildung angeben. Zum Beispiel, das Codewort einer natürlichen Zahl bekommen wir, wenn wir die Zahl als Binärzahl darstellen. Die Abbildung zwischen jeder Zahl und Ihrem Codewort ist damit eindeutig und leicht definiert. Dabei ist so eine Abbildung günstig, wenn sie möglichst viele Rechnungen respektiert. Wollen wir das Ergebnis einer Addition zweier Zahlen speichern, können wir entweder das Ergebnis der Rechnung codieren, oder die zwei zugehörigen Codewörter addieren. Das Ergebnis beider Methoden ist gleich.

Bei der Codierung von Zeichen ist das nicht notwendig, da wir mit Zeichen nicht rechnen wollen. Aus diesem Grund stellen wir uns jede Zeichencodierung einfach als Tabelle vor. Auf der linken Seite stehen alle Zeichen, auf der rechten Seite stehen die zugehörigen Codewörter. Mit dem Beispiel des Alphabetes der Großbuchstaben sieht eine **Codetabelle** folgendermaßen aus. Hier brauchen wir 5 Bits, um die 26 Buchstaben zu codieren.

A	00001
B	00010
C	00011
…	…
Z	11010

Wenn wir Texte in Computern speichern, wollen wir diese meist nicht nur für uns behalten. Wir wollen Textdateien austauschen, Emails verschicken oder Webseiten veröffentlichen. Deshalb hat es Sinn, dass nicht jedes System seine eigene Codetabelle zum Speichern von Zeichen verwendet.

Der **ASCII-Code** (American Standard Code for Information Interchange) ist eine Zeichenkodierung, die ursprünglich in den 1960er Jahren entwickelt wurde, um eine standardisierte Methode zur Darstellung von Textzeichen zu schaffen. ASCII verwendet 7 Bit zur Kodierung von 128 verschiedener Buchstaben, Ziffern, Interpunktionszeichen und Steuerzeichen.

Im ASCII-Code sind die ASCII-Werte 0 bis 31 und der Wert 127 **Steuerzeichen**. Die ASCII-Werte 32 bis 126 sind sogenannte **druckbare Zeichen**. Unter den druckbaren Zeichen sind die Werte $65 - 90$ Großbuchstaben, $97 - 122$ Kleinbuchstaben, $48 - 57$ Ziffern und der Rest der Werte Sonderzeichen und Satzzeichen. Wie genau die Werte auf die Zeichen zugeordnet sind, wurde einfach durch die Veröffentlichung der ASCII-Codetabelle definiert. Diese können sie sich ganz einfach im Internet ansehen.

Der Buchstabe A hat zum Beispiel den ASCII-Wert 65. Damit ist das A die 1000001 in jeder mit ASCII codierten Textdatei. Der Buchstabe a hat den ASCII-Wert 97, wird also durch 1100001 codiert. Bei den Steuerzeichen kommen zum Beispiel die ASCII-Werte 10 für „Zeilenumbruch“, 13 für „Wagenrücklauf“ und 127 für ein „Löschzeichen“ öfters vor.

Wir können den ASCII-Code in *Microsoft Word* ausprobieren. Wir schauen uns die ASCII-Tabelle an und suchen uns das $*$ Zeichen mit dem ASCII-Code 42 aus. Jetzt aktivieren wir den

Nummernblock, halten die ALT-Taste gedrückt, geben 42 über den Nummernblock ein und lassen dann die ALT-Taste los.

Der ASCII-Code bildet bis heute die Basis für viele weitere Zeichencodierungen. Obwohl er heute teilweise durch umfangreichere Codierungen ersetzt wurde, bleibt er aufgrund seiner Einfachheit, weitreichenden Unterstützung und Effizienz ein grundlegender Bestandteil vieler Systeme und Anwendungen.

Der ASCII-Code deckt 128 Zeichen ab. Andere Standards wie ISO 8859-1 erweitern das auf 256 Zeichen, aber auch dies ist für viele Schriftsysteme und Sprachen unzureichend. Unicode wurde in den späten 1980er Jahren vom Unicode-Konsortium entwickelt, um diese Probleme zu lösen. Die erste Version, Unicode 1.0, wurde 1991 veröffentlicht. Seitdem wird der Standard kontinuierlich erweitert und aktualisiert, um neue Zeichen und Schriftsysteme zu unterstützen. Unicode ermöglicht es, nahezu alle Schriftzeichen und Sonderzeichen, die in der Informatik und Datenverarbeitung verwendet werden, einheitlich darzustellen.

Die Grundlage des **Unicode** ist die **Unicode-Tabelle**. Die Unicode-Tabelle beginnt mit der ASCII-Code Tabelle, danach folgen aber mehr als 1 Millionen weitere Zeilen. Von diesen Zeilen sind bis heute „nur" ungefähr 150.000 Zeilen mit Zeichen belegt. Jede belegte Zeile der Unicode-Tabelle nennen wir einen **Codepunkt** und die Zahl der definierten Codepunkte steigt kontinuierlich, da regelmäßig neue Versionen des Unicode-Standards veröffentlicht werden, um zusätzliche Zeichen und Symbole zu integrieren.

In der Unicode-Tabelle werden alle möglichen Codepunkte, also alle Zeilen der Tabelle einfach durchnummeriert. Um in Darstellungen der Tabelle Platz zu sparen, nummeriert man diese Tabelle durch eine vierstellige Hexadezimalzahl. In Hexadezimaler Schreibweise gibt es nicht nur die Ziffern 0 bis 9, sondern die Ziffern 0 bis F. Die Ziffern 0 bis 9 sind einfach die Zahlen 0 bis 9. Dann zählen wir weiter. A ist die 10, B ist die 11, … F ist die 15.

Sie finden die Unicode-Tabelle im Internet. Wenn sie sich diese zum Beispiel hier: https://symbl.cc/de/unicode-table/ einmal anschauen, können sie sehen, dass nach Codepunkt 0009, der Codepunkt $000A$ folgt, denn A ist 10 in hexadezimaler Schreibweise.

Suchen Sie sich in der Unicode-Tabelle ein lustiges Zeichen aus, zum Beispiel das Ÿ. Dieses Zeichen hat den Codepunkt $03AB$. Rechnen wir diese Position in Dezimalschreibweise um, erhalten wir die folgende Zahl.

$$\begin{aligned}(03\mathrm{AB})_{16} &= 0 \cdot 16^3 + 3 \cdot 16^2 + (\mathrm{A})_{16} \cdot 16^1 + (\mathrm{B})_{16} \cdot 16^0 \\ &= 3 \cdot 16^2 + 10 \cdot 16 + 11 \cdot 1 \\ &= 939\end{aligned}$$

Wieder können wir in Word die ALT-Taste gedrückt halten, 939 über den Nummernblock eingeben und erhalten Ÿ.

Das Emoji ❤️ finden wir am Codepunkt 2764. Die hexadezimale Zahl 2764 ist die dezimale Zahl 10084. Das Zeichen für die Zahl π hat den Unicodepunkt $03C0$, eine ägyptische Hieroglyphe finden wir am Codepunkt $1302C$. Die Hieroglyphe bekommen wir also mit ALT+77868.

Insgesamt hat also jedes Zeichen einen eindeutigen Codepunkt, eine Hausnummer, in der Unicode-Tabelle. Jetzt müssen wir nur noch den Codepunkt speichern. Hier gibt es jetzt drei verschiedene Varianten der Unicode-Codierung. Die Codewörter der Codepunkte können wir in UTF-8, UTF-16 oder in UTF-32 darstellen. Dabei steht UTF immer für **Unicode Transformation Format**.

UTF-8 ist die am häufigsten verwendete Unicode-Codierung, da sie effizient Speicherplatz nutzt und direkt mit ASCII kompatibel ist. UTF-8 speichert Unicode-Zeichen in einer variablen Anzahl von 8, 16, 24 oder 32 Bit.

Die ersten Codepunkte werden mit 8-Bits gespeichert. Sind diese 8-Bits aufgebraucht, werden die folgenden Codepunkte mit 16-Bits gespeichert. Und so weiter. Dadurch nehmen die vorderen Zeichen des Unicodes weniger Platz ein als die hinteren Zeichen. Wenn wir davon ausgehen, dass die Zeichen des Unicodes ungefähr nach Häufigkeit geordnet sind, führt das dazu, dass wir mit dieser Technik viel Platz bei der Speicherung von Text sparen.

Um ein UTF-8 Codewort zu **decodieren**, betrachten wir die ersten Bits der Bitkette. Ist das führende Bit eine 0, dann speichern die nächsten 7-Bits einen Codepunkt. In der Unicode-Tabelle sind das genau die ASCII-Zeichen. Beginnt die Bittkette mit einer 1, zählen wir die 1er bis zur ersten 0. Bekommen wir eine 10, befinden wir uns innerhalb eines Codeworts. Bekommen wir eine 110, ist der Codepunkt Teil der nächsten 13 Bits. Bekommen wir eine 1110, ist der Codepunkt Teil der nächsten 20-Bits. Bekommen wir eine 11110 ist der Codepunkt Teil der nächsten 27-Bits. Da die 10 wir jeden 8ter Block an Bits eine Sonderrolle einnimmt, bekommen wir für UTF-8 die folgenden vier Klassen von Möglichkeiten einen Codepunkt zu speichern. Die 0er und 1er ergeben dabei eine sogenannte **Bitmaske**. Die $x$er können je nach Codepunkt die Werte 0 oder 1 annehmen.

$$0xxxxxxx$$

$$110xxxxx\ 10xxxxxx$$

$$1110xxxx\ 10xxxxxx\ 10xxxxxx$$

$$11110xxx\ 10xxxxxx\ 10xxxxxx\ 10xxxxxx$$

Die erste Klasse von UTF-8 Zeichen hat also Platz für die ersten 128 Codepunkte der Unicode-Tabelle. Das sind genau unsere ASCII-Zeichen. Texte die nur aus diesen Zeichen bestehen, können damit genau so effizient wie mit ASCII-Code gespeichert werden. Die zweite Klasse an UTF-8 Zeichen hat die Länge 16, dabei sind jetzt aber 5 Bits für die Bitmaske reserviert. Damit können wir mit dieser Klasse 2048 verschiedene Codepunkte darstellen. Die dritte Klasse der UTF-8 deckt 65536 weitere Codepunkte ab. Die vierte Klasse der UTF-8 schafft theoretisch noch mal weitere $2\,097\,152$ Codepunkte. Wie bereits erwähnt, zurzeit sind ca. 150.000 der Punkte wirklich belegt.

#	Codepunkt	Zeichen	UTF-8-Codierung
42	$002A$	*	00101010
960	$03C0$	π	11001111 10000000
10084	2764	❤	11100010 10011101 10100100
77869	$1302D$	𓃭	11110000 10010011 10000000 10101101

Diese Tabelle zeigt vier Zeilen der Unicode-Tabelle. In der ersten Spalte steht die Nummer der Zeile als Dezimalzahl. In der zweiten Spalte steht dieselbe Zahl als Hexadezimalzahl. In der letzten Spalte steht diese Zahl noch einmal als Binärzahl, wenn wir die Bitmaske entfernen. Wir sehen noch mal deutlich, dass die zentrale Idee hinter der ganzen Unicode-Codierung einfach ist, jedem Zeichen eine „Nummer" zuzuordnen.

Insgesamt ist damit die UTF-8 eine äußerst flexible und effiziente Methode zur Codierung von Text. Natürlich nur, wenn wir davon ausgehen, dass die Unicode-Tabelle mit Ihrer Reihenfolge ungefähr die Häufigkeit der verwendeten Zeichen respektiert.

„Es ist strengstens verboten, in den Brunnen zu klettern" schreiben wir in Chinesisch einfach als:

嚴禁爬進井裡

Dabei hat zum Beispiel das Zeichen 嚴 den UTF-8 Code 11100101 10011010 10110100. Alle sechs Zeichen werden in UTF-8 mit 24 Bits codiert. Aus diesem Grund ist die UTF-8 im asiatischen Raum nicht weit verbreitet.

Die **UTF-16** Codierung probiert das etwas auszugleichen. In der UTF-16 Codierung der Unicode-Tabelle wird jeder Codepunkt direkt durch 16 oder durch 32 Bit dargestellt. Das bedeutet, dass jetzt zwar jedes ASCII-Zeichen mehr Platz einnimmt, dafür werden aber auch mehr Zeichen mit 16 statt mit 24 Bit codiert. Das Zeichen 嚴 ist in der UTF-16 Codierung das Codewort 01010110 10110100. Wir sparen für dieses Zeichen also 8 Bit Platz.

Da wir in der UTF-16 nur zwei verschiedene Längen von Codewörtern haben, ist auch die Bitmaske der UTF-16 einfacher, als die Bitmaske der UTF-8. Beginnt das Codewort mit 110110 ist es ein 32 Bit Zeichen. Beginnt es anders, stellt es einfach den Codepunkt als Binärzahl dar. Damit bekommen wir für unser Zeichen:

$$嚴 = (56B4)_{16} = (101\ 0110\ 1011\ 0100)_2 = [0101\ 0110\ 1011\ 0100]_{UTF-16}$$

Die UTF-16 stellt die sogenannte **Basic Multilingual Plane**, grob gesagt die Codepunkte 0000 bis $FFFF$, durch Codewörter der Länge 16 Bits dar. Bei der Basic Multilingual Plane sind lediglich die Codepunkte $D800$ bis $DFFF$ ausgenommen, um das Präfix 110110 für die mit 32 Bit codierten Zeichen zu reservieren.

Die **UTF-32** stellt einfach alle Codepunkte mit 32 Bits dar. Damit ist die UTF-32, genau wie der ASCII-Code, eine Codierung mit konstanter Länge für alle Codewörter. Das spart keinen Platz, ist aber etwas leichter zu decodieren. Es kommt also immer auf die Anwendung an, für welche Codierung man sich entscheidet.

Wollen wir Text verschicken oder langfristig sichern, hat es Sinn diesen nicht nur zu speichern, sondern dabei so weit wie möglich zu **komprimieren**. Im Grunde kennen wir diese Idee bereits von der UTF-8 Codierung. Häufige Zeichen wollen wir mit möglichst kurzen Bitketten codieren. Dafür spendieren wir für seltene Zeichen mehr Platz.

In der Informatik können wir uns diese Frage jetzt allgemein Stellen. Können wir für einen festen Text, eine möglichst platzsparende Codierung angeben?

Die **Huffman-Codierung** ist ein verlustfreies **Datenkompressionsverfahren**, das von David A. Huffman entwickelt wurde. Es basiert auf der Häufigkeit von Zeichen in einem Text. Diese Methode ist besonders effektiv für Texte mit einer nicht-uniformen Verteilung von Zeichen. Die Huffman-Codierung verwendet ein binäres Baumdiagramm, um eindeutige Präfixcodes für jedes Zeichen zu erzeugen. Präfixcodes sind so konzipiert, dass kein Code das Präfix eines anderen Codes ist, was eine eindeutige Dekodierung ermöglicht.

Betrachten wir als Beispiel den folgenden Satz:

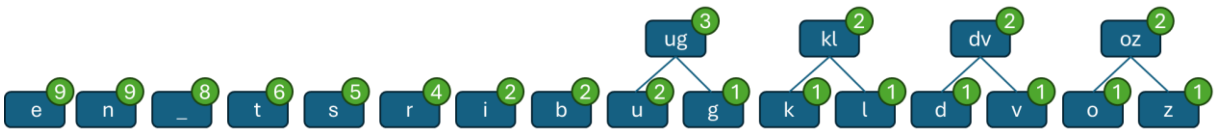
> es ist strengstens verboten in den brunnen zu klettern

Der erste Schritt der Huffman-Codierung besteht darin, die Häufigkeit jedes Zeichens zu zählen.

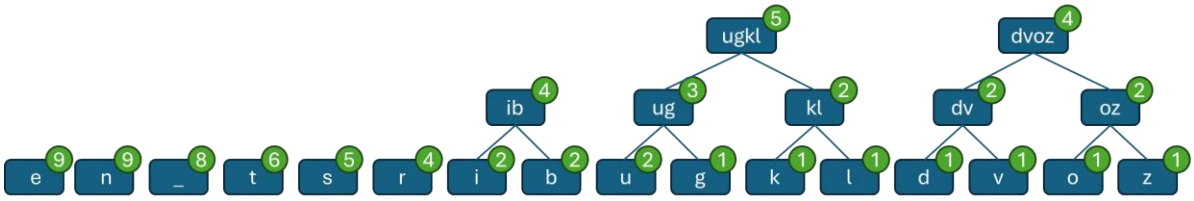
e	n	_	t	s	r	i	b	u	g	k	l	d	v	o	z
9	9	8	6	5	4	2	2	2	1	1	1	1	1	1	1

Der zweite Schritt der Huffman-Codierung besteht darin, eine Baumstruktur zu konstruieren. Dafür beginnen wir mit der Menge der Zeichen und ihren Häufigkeiten. Jedes Zeichen ergibt einen Knoten. Jetzt fassen wir immer zwei Knoten mit möglichst kleinen Häufigkeiten zu einem neuen Knoten zusammen. Die Häufigkeit des neuen Knoten ist dabei immer die Summe der beiden Kinderknoten.

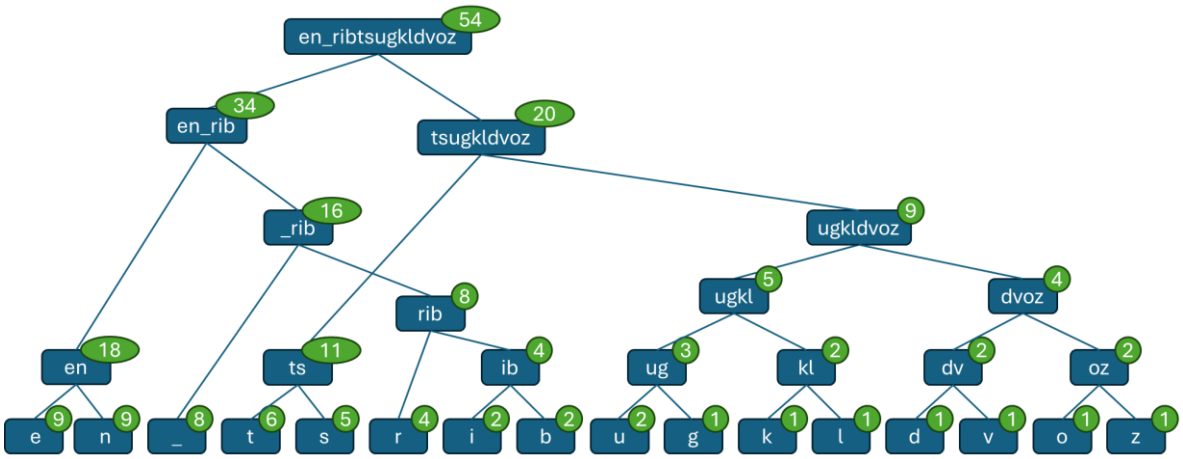
Nach vier Iterationen hat der Algorithmus alle Knoten mit Häufigkeit 1 abgearbeitet.



Nach sieben Iterationen hat der Algorithmus alle Knoten mit Häufigkeit 2 abgearbeitet.



Insgesamt wird jeder Knoten zusammengefasst, bis schließlich nur ein einziger Knoten übrigbleibt. Die so entstandene Struktur nennen wir einen **Binärbaum**. Jeder Knoten, mit Ausnahme der sogenannten Blätter, hat genau zwei Kinder. Natürlich hängt die Struktur dieses Baums davon ab in welcher Reihenfolge wir Knoten mit gleichen Anzahlen zusammenfassen. Der Huffman Code ist nicht eindeutig.



Der dritte Schritt besteht darin, jedem Zeichen ein Codewort zuzuordnen. Wir beginnen bei der Wurzel und laufen dann durch den Baum bis zu dem Blatt, das wir codieren wollen. Biegen wir auf dem Weg nach links ab, merken wir uns eine 0. Biegen wir nach rechts ab, merken wir uns eine 1. Der Weg von der Wurzel zum Blatt e ist damit 000. Der Weg von der Wurzel zum Blatt k ist damit 11010. Der Weg zu jedem Blatt ist das Codewort für das Zeichen des Blattes.

Da kein Blatt auf dem Weg zu einem anderen Blatt liegt, ist der Huffman-Code präfixfrei. Das heißt, kein Codewort ist Präfix von einem anderen Codewort. Damit lässt sich dieser Code immer eindeutig decodieren.

Mit unserem Baum bekommen wir die folgende Codetabelle. Wir sehen sofort, dass häufige Zeichen kurze Codewörter besitzen. Diese Zeichen wurden einfach weniger zusammengefasst. Trotzdem ist der Code präfixfrei. Kein anderes Codewort beginnt mit 000. Kein anderes Codewort beginnt mit 001. Und so weiter.

e	000
n	001
_	010
t	100
s	101
r	0110
i	01110
b	01111

u	11000
g	11001
k	11010
l	11011
d	11100
v	11101
o	11110
z	11111

Unser Text

> es ist strengstens verboten in den brunnen zu klettern

ergibt mit unserem Codebaum

> 000101010011101011000101011000110000001110011011000000001101010111010000
> 110011111111010000000101001110001010111000000010100111101101100000100 10
> 000010101111111000010110101101100010010000000110001

Dadurch das der Huffman-Code präfixfrei ist, können wir ihn einfach von vorne nach hinten lesen. Immer wenn wir ein Codewort aus der Tabelle erkennen, ersetzen wir die gelesene Bitkette durch das zugehörige Zeichen.

Der codierte Satz hat jetzt insgesamt eine Bitlänge von 192. Alternativ hätten wir die 16 Zeichen mit je 4 Bit kodieren können. Damit hätten wir mit unseren 54 Zeichen ein Codewort mit einer Gesamtlänge von 216 Bits. Durch die Huffman-Codierung haben wir in diesem Beispiel 24 Bits Speicherplatz gespart.

Die Huffman-Codierung ist nur dann optimal, wenn wir die Frequenzen der zu codierenden Zeichen kennen. Fehlen uns diese Informationen oder ändern sie sich während der Verarbeitung, führt die Codierung zu suboptimaler Komprimierung. Da verschiedene Anwendungen unterschiedliche Frequenzverteilungen der Zeichen erwarten, müssen wir für jede Anwendung eine separate Huffman-Codierung berechnen und speichern. Das erschwert die Vereinheitlichung und macht es schwierig, Codes zu standardisieren. In vielen Fällen ist es daher nicht sinnvoll, die Huffman-Codierung als Standardlösung anzuwenden, da der Aufwand für die Berechnung und Speicherung der entsprechenden Frequenzdaten in keinem Verhältnis zur tatsächlichen Komprimierungsleistung steht.

1.3.3 Ton und Bild

Wir wissen jetzt, wie wir Zahlen und Texte in unserem Computer speichern. Aber was ist mit Bildern, Musik und Videos? Können wir diese auch einfach binär codieren? Ton und Bilder, das ist doch etwas Analoges.

Ein **analoges** Signal oder System kann jeden beliebigen Wert innerhalb eines bestimmten Bereichs annehmen. Ein schönes Beispiel für ein analoges System ist eine Uhr. Der Sekundenzeiger einer analogen Uhr durchläuft die Minute und nimmt dabei jeden möglichen Wert an. Eine **digitale** Armbanduhr zerlegt die Minute in 60 Sekunden. Moderne Oszilloskope können elektrische Signale im Nanosekundenbereich messen, aber Zeit vergeht kontinuierlich!

Um einer physikalischen und kontinuierlichen Größe einen Wert zuordnen zu können, müssen wir diese zunächst **diskretisieren**. Auch die genaue Position unserer analogen Uhr können wir nie genau ablesen. Die diskretisierte und gemessene Größe, zum Beispiel eine Anzahl von Sekunden, können wir dann wie gewohnt **digitalisieren**, also codieren und abspeichern.

Wie genau funktioniert das jetzt mit Musik? Im Grunde können wir Musik auch analog speichern, zum Beispiel mit einer Schallplatte. Schallwellen drücken auf eine Membran und die so erzeugte Vibration wird durch einen sogenannten Lackschneider in eine Schalplatte geschnitten. Beim Abspielen der Schalplatte fährt der Tonarm eines Plattenspielers über die Platte, was die aufgenommenen Vibrationen erneut erzeugt. Die Vibrationen werden auf eine Membran übertragen und durch einen Lautsprecher verstärkt. Dadurch können wir die Musik auf der Schallplatte speichern, aber in einem Computer funktioniert das Ganze doch ein bisschen anders.

Bei der digitalen Speicherung wird analoge Musik zunächst diskretisiert und dann in digitale Daten umgewandelt. Erst dann kann sie von Computern und anderen digitalen Geräten verarbeitet werden. Wir können Musik dann speichern, streamen, bearbeiten und vieles mehr.

Um die Ideen der digitalen Speicherung von Musik zu verstehen, stellen wir uns das analoge, akustische Signal auch wieder als Schallwelle vor. Die eigentliche Diskretisierung dieser Welle geschieht dann in zwei Schritten, dem Sampling und der Quantisierung.

Beim sogenannten **Sampling** wird die analoge Schallwelle in regelmäßigen Abständen abgetastet. Die Abtastrate bestimmt, wie oft das Signal pro Sekunde abgetastet wird. Eine übliche Abtastrate für Musik ist 44,1 kHz, was bedeutet, dass 44.100 Samples pro Sekunde genommen werden. Für jedes Sample bestimmen wir dann den aktuellen Druck, die „Höhe“, die Amplitude, der Schallwelle. Dieser Schritt heißt **Quantisierung**. Je genauer wir die Höhe messen und je mehr Bits wir für die Speicherung dieses Wertes zur Verfügung stellen, je präziser wird jedes Sample erfasst. Ein üblicher Wert ist hier eine 16-Bit-Tiefe.

Wenn wir die Abtastrate kennen, können wir die Folge von Amplituden als einfache Folge von Zahlen speichern. Mit dieser Folge können wir dann die Schallwelle jederzeit rekonstruieren. Dabei können wir das aufgenommene Signal umso besser rekonstruieren, je höher unsere Abtastrate ist und je genauer wir quantisiert haben. Dabei führt eine höhere Qualität aber auch direkt zu einer größeren Datenmenge.

Genau nach diesen Prinzipien speichern wir Musik auf einer CD und als Datei im WAV- oder AIFF-Format. CDs haben eine sehr hohe Klangqualität, da die Schallwelle fast unkomprimiert, bei hoher Abtastrate und guter Quantisierung, relativ verlustfrei digitalisiert wird. Dazu ist dieses Format auch sehr einfach zu verarbeiten. Die erzeugten Dateien sind im Vergleich zu komprimierten Formaten jedoch sehr groß.

Betrachten wir das folgende Beispiel: https://www.youtube.com/watch?v=dQw4w9WgXcQ

Das Lied hat eine Länge von $3{:}32$ Minuten. Für jede Sekunde brauchen wir 44100 Samples zu je 16 Bits. Zusätzlich speichern wir jetzt zwei Audiokanäle, um das Lied in Stereo zu speichern und können den Platzbedarf wie folgt abschätzen.

$$44100 \cdot 16\,Bits \cdot 2 \cdot 212 = 299.174.400\,Bits = 37.396.800 Byte \approx 35\,MB$$

Auf einer üblichen Festplatte ist eine solche Dateigröße völlig in Ordnung. Auch eine Audio-CD hat ungefähr $700\,MB$ Speicher. Trotzdem ist ein solches unkomprimiertes Format nicht für alle Anwendungsbereiche effizient.

Da wir es mittlerweile gewohnt sind unsere Musik direkt aus dem Internet zu streamen, zum Beispiel mit Apple Musik, YouTube oder klassisch mit Spotify, gibt es weitere clevere Formate, mit denen wir Musik komprimiert abspeichern können.

Der **Advanced Audio Codec** (AAC) ist ein verlustbehaftetes Audioformat, das als Nachfolger des MP3-Formats entwickelt wurde. Es wurde von einem Konsortium aus Unternehmen und Organisationen wie Fraunhofer IIS, AT&T, Dolby, Sony und Nokia entwickelt und ist im Rahmen der MPEG-2- und MPEG-4-Spezifikationen standardisiert.

Die Datenkompression im AAC-Format basiert auf einer Reihe von Techniken, die darauf abzielen, die Datenmenge zu reduzieren, ohne die wahrgenommene Audioqualität zu beeinträchtigen. AAC nutzt ein psychoakustisches Modell, um zu bestimmen, welche Teile des Audiosignals vom menschlichen Ohr weniger wahrgenommen werden. Diese Teile werden dann stärker komprimiert oder sogar ganz entfernt. Das psychoakustische Modell basiert auf Erkenntnissen darüber, wie Menschen verschiedene Frequenzen hören und auf bestimmte Töne reagieren. Zusätzlich teilt AAC das Audiosignal mit einer Filterbank in verschiedene Frequenzbänder auf. So müssen wir dann nur die Frequenzkoeffizienten quantisieren und codieren. Wir können uns das im Grunde so vorstellen, dass wir die Schallwelle als gewichtete Summe von Cosinus- und Sinuskurven approximieren, anstatt viele konkrete Werte abzutasten. Die Frequenzkoeffizienten werden dann noch durch Skalierungsfaktoren normalisiert, nach dem psychoakustischen Modell gewichtet und letztendlich mit der Huffman-Codierung weiter komprimiert. Diese Methoden machen AAC zu einem der effizientesten und qualitativ hochwertigsten verlustbehafteten Audioformaten. Unser Lied, mit einer Länge von $3{:}32$, nimmt im AAC-Format bei mir auf der Festplatte ungefähr $4\,MB$ Speicherplatz ein.

Die digitale Speicherung von Musik hat die Art und Weise revolutioniert, wie wir Musik produzieren, speichern und konsumieren.

Bei der **digitalen Speicherung eines Bildes** wenden wir die gleichen Prinzipien an. Zunächst müssen wir das Bild **diskretisieren**. Genau wie wir eine Schallwelle in einem gewissen Takt abtasten, zerlegen wir ein Bild in eine Menge von Punkten. Anschließend **quantifizieren** wir das Bild, indem wir für jeden Punkt, seine Farbe als Farbwert abspeichern. Farbwerte können wir zum Beispiel als RGB-Wert, als Mischung der Farben Rot, Grün und Blau, darstellen. Typischerweise legt man dabei für jede der Farben einen Wert zwischen 0 und 255 fest. Für die Farbe Rot zum Beispiel bedeutet ein Wert von 0 das kein Rot enthalten ist. 50 bedeutet, es ist etwas Rot enthalten, 255 bedeutet so viel Rot wie möglich. Insgesamt ist der RGB-Wert $(0,0,0)$ die Farbe Schwarz, denn hier ist einfach keine Farbe enthalten. Der Wert $(255,255,255)$ ist die Farbe Weiß, der Wert $(255,200,255)$ ist eine Mischung aus Rot und Blau mit etwas Grün. Das ergibt ein schönes helles Schweinchenrosa. Mit Microsoft Paint lässt sich das alles wunderbar ausprobieren.

Genau nach diesen Prinzipien arbeitet das Bitmap Format (BMP). Dieses ist das klassische Rastergrafikformat, das hauptsächlich von Windows verwendet wird. Die Anzahl der tatsächlichen Farbpunkte (Pixel) pro Zentimeter hängt dabei von der eingestellten Auflösung des Bildes ab. Zum Beispiel enthält ein BMP-Bild mit einer Auflösung von 300 Pixel pro Zoll (PPI) etwa 118 Pixel pro Quadratzentimeter. Die Farbtiefe von BMP-Dateien kann je nach Version und Anwendung variieren. Mit den oben beschriebenen 8 Bits pro Farbe eines Farbwertes, benötigen wir 24 Bits für jeden RGB-Wert. Damit bekommen wir BMP-Bilder mit $16{,}7$ Millionen verschiedenen Farben (True Color). Dies ist die am häufigsten verwendete Farbtiefe für Fotografien und detaillierte Grafiken

Genau wie das WAV-Format für Audio-Dateien, ist das BMP-Format für Bild-Dateien nicht komprimiert. Das bedeutet, dass wir hier größere Dateien erzeugen als komprimierte Formate wie JPEG oder PNG.

Wenn wir kurz einen Blick in die Foto-Mediatheken unserer Smartphones werfen, erkennen wir sofort, dass es auch bei der Speicherung von Bildern Sinn haben kann auf den dadurch verbrauchten Speicherplatz zu achten.

Das **Joint Photographic Experts Group (JPEG) – Format** ist ein sehr weit verbreitetes Dateiformat für die digitale Speicherung von Bildern. Ähnlich wie das AAC-Format basiert auch das JPEG-Format aus einer Reihe von Schritten, die zu einer verlustbehafteten Kompression führen.

Zunächst führen wir eine Farbraumumwandlung durch. Dabei wandeln wir das Bild vom RGB-Farbraum in den YCbCr-Farbraum um. Dieser Farbraum trennt die Helligkeitsinformation, die sogenannte Y-Komponente, von den Farbinformationen, den Cb- und Cr- Komponenten. Dann wird das gesamte Bild in 8x8 Pixel große Blöcke unterteilt und jeder Block wird einzeln verarbeitet. Auf jeden Block wird eine diskrete Kosinustransformation angewendet. Diese Transformation konvertiert das Bild von der räumlichen Domäne in die Frequenzdomäne. Stellen wir uns das wieder so vor, dass die Helligkeitsverteilung auf einem Foto eine Welle bildet, denn Farben gehen fließend ineinander über. Diese Welle können wir, genau wie eine Schallwelle, approximieren. Nach der Transformation werden die dadurch berechneten Koeffizienten quantisiert. Auch hier werden weniger wichtige oder feine Details aus den Bildinformationen stark reduziert, da wir verschiedene Farben unterschiedlich gut wahrnehmen können. Die quantisierten Koeffizienten werden abschließend mit der Huffman-Codierung komprimiert und dann gespeichert.

Das JPEG-Format verwendet also die gleichen Prinzipien wie das ACC-Format. Einmal komprimieren und speichern wir die Schallwelle, einmal die Helligkeitsverteilung von Farbwerten. Beide Formate sind, durch die effiziente Komprimierung für die Speicherung und Übertragung von Dateien sehr beliebt.

Die **digitale Speicherung eines Videos** ist im Grunde die Speicherung einer Folge von Bildern und der Tonspur des Videos. Wir können also Videos speichern, in dem wir unsere Techniken für Bild und Ton miteinander kombinieren. Zusätzlich können wir auf diese Kombination weitere Techniken aufsetzen. Wir können in einer Folge von Bildern nur die Bereiche eines Bildes abspeichern, die sich zu dem vorangegangenen Bild ändern. Verfahren die diese Technik verwenden heißen **Interframe-Kompressionsverfahren**. Bei einem Interframe-Kompressionsverfahren unterscheiden wir Intra-Frames, Bilder die komplett gespeichert werden, und Predictive-Frames und Bidirectional-Frames, die nur die Unterschiede zu vorangegangenen und nachfolgenden Bildern speichern. Das hier bekannteste Format ist wohl der H.265 Video-Codec.

Das MP4-Format (MPEG-4) ist ein **Digital-Multimedia-Format**, das wir verwenden, um mehrere verschiedene Inhalte in einer Datei zu speichern. MP4 ist ein Containerformat, das verschiedene Arten von Daten, Video, Audio, Untertitel und Bilder, in einem einzigen Datei-Container speichern kann. Es trennt die unterschiedlichen Datentypen in separate Spuren (Tracks). Jede Spur enthält eine spezifische Art von Daten, wie z.B. eine Videospur, eine Audiospur oder eine Untertitelspur. Jede dieser Spuren kann verschiedene Codecs verwenden, um die Daten zu kodieren. Das MP4-Format hat sich als das am weitesten verbreitete und vielseitigste Format zur Speicherung und Wiedergabe von Multimedia-Inhalten etabliert.

1.3.4 Der ganze Rest

In den vorangegangenen Abschnitten haben wir gelernt, wie wir Zahlen, Zeichen und Medien digital in unseren Computern speichern. Aber ist das schon Alles? Sie studieren Wirtschaftsinformatik und in modernen Enterprise Ressource Planning Anwendungen, wie zum Beispiel in SAP, verwalten wir Kunden, Aufträge, Bestellungen, Rechnungen und vieles mehr.

In der Informatik nennen wir Gegenstände oder Dinge, die wir im Computer speichern wollen, **Objekte**. Um ein Objekt zu speichern, müssen wir es zunächst beschreiben. Dazu betrachten wir als erstes die Eigenschaften unseres Objekts. In der Informatik beschreiben wir Objekte, indem wir eine Menge von **Attributen** des Objektes und die entsprechenden **Werte** aufzählen.

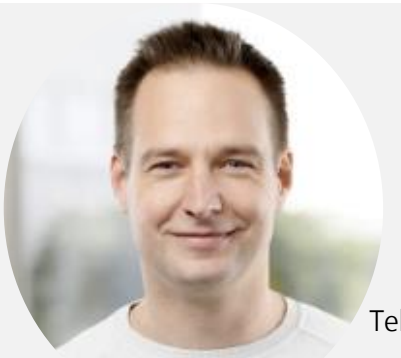
Wollen Sie mich, zum Beispiel als Kontakt in Ihrer Telefonbuchapp ablegen, schauen Sie zunächst auf die Menge der Attribute, die sie benötigen, um Kontakte in Ihrer Anwendung zu beschreiben. So eine Menge von Attributen nennen wir in der Informatik eine **Klasse**. Um Kontakte zu speichern, brauchen Sie von jedem Kontakt die zugehörige Anrede, Titel, Vorname, Nachname, Telefonnummer und eine E-Mail-Adresse. Um „mich“ zu speichern, speichern Sie für jedes dieser Attribute den für mich passenden Wert.

Wenn wir Objekte speichern, können wir die Liste der Attribute beliebig anpassen, je nach dem, was sie benötigen, um den Zweck Ihrer Anwendung zu erfüllen. Sie können meinen Geburtstag

speichern, damit dieser automatisch in Ihrem Kalender auftaucht und sie ihn bloß nicht vergessen. Die FernUni speichert zum Beispiel noch meine IBAN und meine Steuernummer. Beides benötigt sie, um mein Gehalt zu zahlen. Das Dekanat speichert die Anzahl meiner Lehrverpflichtungsstunden, um zu schauen, ob ich genügend arbeite. So oder so, die Kombination aller Werte der Attribute, dass bin dann ich, innerhalb ihrer Anwendung.

Vorname: Robin

Nachname: Bergenthum



Anrede: Herr

Titel: Dr.

Geburtsdatum: 30.10.1980

Telefonnummer: 023319871773

E-Mail-Adresse: robin.bergenthum@fernuni-hagen.de

Haben wir die Menge der Attribute einer Klasse festgelegt, können wir Objekte dieser Klasse relativ einfach im Computer als eine Art Tabelle ablegen. Die Klasse definiert die Struktur der Tabelle, denn wir haben für jedes Attribut eine Spalte. Die Tabelle wird dann mit den Objekten Zeile für Zeile befüllt.

Herr	Dr.	Robin	Bergenthum	023319871773	robin.bergenthum@fernuni-hagen.de
Herr	Dr.	Sebastian	Küpper	023319872988	sebastian.kuepper@fernuni-hagen.de
Herr		Jakub	Kovář	023319874782	jabub.kovar@fernuni-hagen.de
Herr	Prof.	Friedrich	Steimann	023319872998	steimann@fernuni-hagen.de

Jetzt reservieren wir Platz in unserem Computer und schreiben die Attributwerte aller Objekte, von oben nach unten, in unseren Speicher. Reservieren bedeutet dabei, dass wir uns merken, wo die Tabelle in unserem Speicher liegt, damit wir die entsprechenden Werte später wiederfinden können, und dass wir die Tabelle nicht mit anderen Rechnungen oder Werten überschreiben. Ist der Speicher reserviert, tragen wir Zeichenketten, Nummernfolgen, Adressen und Zahlen entsprechend ein, um die Objekte zu speichern. Beim Speichern dieser Zahlen und Nummernfolgen innerhalb der Tabelle, können wir alle Techniken anwenden, die wir bereits gelernt haben.

Im **Modul 65002 Grundlagen der Informatik II** von Sebastian Küpper werden Sie mehr zur sogenannten objektorientierten Programmierung erfahren.