Chair of Embedded Systems
Department of Computer Science
University of Augsburg, Germany

Martin Frieb

Universität Augsburg University

Wintersemester 2020/2021

Informatik 1

Kapitel 4 – Codierung ganzer Zahlen

Contents

Version 20201124-1627

4.1 Weitere mathematische Grundlagen

Kartesisches Produkt

Definition: 4.1 Kartesisches Produkt

Sei A eine Menge, so bezeichnet die Menge

$$A^2 := A \times A := \{(a_1, a_2) \mid a_1, a_2 \in A\}$$

das **kartesische Produkt von A mit sich selbst**.

- Ein Element $(a_1, a_2) \in A \times A$ heißt 2-**Tupel** oder **geordnetes Paar**
- Es besteht aus den **Komponenten** a_1 und a_2 (a_i ist die i-**te Komponente**)
- $A \times A$ liest man A *kreuz* A
- Für A endlich gilt: $|A \times A| = |A|^2$

Beachte: Bei den Tupeln ist die Reihenfolge wichtig!

Beispiel:

- $\{0,1\}^2 = \{(0,0),(0,1),(1,0),(1,1)\}$
- $\mathbb{N}^2 = \{(0,0),(0,1),(1,0),(0,2),(1,1),(2,0),\ldots\}$
- $\mathbb{R}^2 =$ zweidimensionale reelle Ebene

Binäre Relation

Definition: 4.3 Binäre Relation

Sei A eine Menge. Eine **binäre Relation auf** A ist eine Teilmenge $R \subseteq A^2$.
R heißt

- **irreflexiv** $:\Leftrightarrow (a,a) \notin R$ für alle $a \in A$
- **transitiv** $:\Leftrightarrow (a,b),(b,c) \in R \implies (a,c) \in R$ für alle $a,b,c \in A$

Statt $(a,b) \in R$ schreibt man auch aRb (**Infixnotation**).

Die Notation $:\Leftrightarrow$ steht für *genau dann wenn*, d.h. hier wird der linke Begriff definiert durch das was rechts steht.

Beispiel:

Als Beispiel betrachten wir eine simple binäre Relation R, also eine Menge mit Paaren aus zwei Elementen, welche die von uns gewünschten Bedingungen erfüllen. Sei $A = \{a \mid a \in \mathbb{N}\}, a \leqslant 10$ und wir wollen die Relation R aufstellen, welche aussagt, dass Elemente echt kleiner sind als andere.
Es soll also gelten, dass $a < b$ für alle $a, b \in A$ gilt.
Er sieht dann wie folgt aus: $R = \{(1,2),(1,3),(1,4),$ ..., $(1,10),(2,3),(2,4),$

$(2,5)$, ..., $(2,10)$, $(3,4)$, $(3,5)$, ..., $(3,10)$, ..., $(9,10)\}$
Für diese Relation gelten nun sowohl Irreflexivität und Transitivität.

- So ist beispielsweise $(1,1) \notin R$ oder auch $(10,10) \notin R$. D.h. es gilt 1 ist nicht kleiner als 1 und auch 10 ist nicht kleiner als 10.
- Ebenso kann man an den Paaren $(1,3)$ und $(3,4)$ Transitivität erkennen, denn $(1,4) \in R$.

Definition: 4.4 (Irreflexive) Ordnungsrelation

Sei A eine Menge. Eine irreflexive, transitive Relation R auf A heißt **(irreflexive) Ordnungsrelation**.

- Ordnungsrelationen haben keine Zyklen, d.h. es gibt kein $n \in \mathbb{N}$ und Elemente $a_1, \ldots, a_n \in A$ mit $(a_1, a_2), \ldots, (a_{n-1}, a_n), (a_n, a_1) \in R$ (ohne Beweis).
- Für Ordnungsrelationen benutzt man in der Regel nicht das Symbol R, sondern eines der Symbole $<$, $\prec$ oder $\ll$.
- Eine Ordnungsrelation legt eine Reihenfolge zwischen *kleineren* und *größeren* Elementen in A fest.

Beispiel:

Die *ist kleiner als*-Relation $<$ ist eine (irreflexive) Ordnungsrelation auf $\mathbb{R}$.

Definition: 4.5 Geordnete Menge

Eine **geordnete Menge** besteht aus einer Menge A und einer Ordnungsrelation $<$ auf A.

Ist $<$ **total**, d.h. gilt $a < b$ oder $b < a$ für alle Elemente $a, b \in A$, so ist die Menge **total geordnet**.

Beispiel:

- $\mathbb{R}$ und $<$ bilden eine total geordnete Menge: für alle Elemente $a, b \in \mathbb{R}$ ist ein Vergleich via $<$ möglich.
- Die *ist echte Teilmenge von*-Relation $\subset$ ist eine (irreflexive) Ordnungsrelation auf $P(A)$ (für eine beliebige Menge A). $P(A)$ und $\subset$ bilden eine (nicht total)[a] geordnete Menge.

Erinnerung: $P(A)$ *ist die Potenzmenge von* A*, also die Menge aller Teilmengen von* A.

[a] Betrachte das Beispiel der Menge $M = \{a, b\}$ mit $P(M) = \{\emptyset, \{a\}, \{b\}, \{a, b\}\}$. Zwar gilt $\subset$ für einige Elemente der Potenzmenge wie z.B. $\emptyset \subset \{a\}$, aber nicht für alle. Beispielsweise gilt weder $\{b\} \subset \{a\}$ noch $\{a\} \subset \{b\}$, somit ist $\subset$ **nicht** total.

Alphabet

Definition: 4.7 Alphabet, Buchstabe

Ein **Alphabet** ist eine endliche, total geordnete Menge A.
Deren Elemente nennen wir **Buchstaben**. Die Ordnung bezeichnen wir mit $<$.

In der Literatur werden Buchstaben auch häufig als Symbole bezeichnet.

Beispiel:

- Menge der Binärziffern $\mathbb{B} = \{0, 1\}$
  mit der Ordnung $0 < 1$
- Menge der Dezimalziffern $\mathbb{D} = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
  mit der Ordnung $0 < 1 < 2 < 3 < 4 < 5 < 6 < 7 < 8 < 9$
- Menge der lateinischen Großbuchstaben
  mit der Ordnung $A < B < \ldots < Y < Z$
- Menge der ASCII-Zeichen $\mathbb{A}$
  mit der Ordnung bzgl. der ASCII-Codes

Wörter (über einem Alphabet)

Definition: Wort, formale Sprache

- Ein **Wort** (über einem Alphabet A) ist eine endliche Folge $w = a_1 a_2 \ldots a_k$ von Buchstaben aus A
- Die **Länge** $|w|$ eines Wortes w ist die Anzahl der Buchstaben von w (gleiche Buchstaben werden mehrfach gezählt)
- Das **leere Wort** ε hat die Länge $|\varepsilon| = 0$
  Achtung: ε ist kein Buchstabe!
  Anmerkung: In der Literatur wird auch das Symbol λ *verwendet*
- Wir bezeichnen mit A^* die **Menge aller Wörter (über** A**)**.
- Wir bezeichnen mit $A^+ := A^* \setminus \{\varepsilon\}$ die Menge aller **nichtleeren** Wörter über A.
- Wir bezeichnen mit $A^k := \{w \mid w \in A^*, |w| = k\}$ die Menge aller Wörter **der Länge** k.
- Eine **formale Sprache (über** A**)** ist eine Teilmenge $L \subseteq A^*$.

Beispiel:

Sei $A = \{a, b, c\}$, dann ist

- $w = abc$ ein Wort über dem Alphabet A.
- A^0 die Menge der Wörter der Länge 0: $A^0 = \{\varepsilon\}$.
- A^1 die Menge der Wörter der Länge 1: $A^1 = \{a, b, c\}$.
- A^2 die Menge der Wörter der Länge 2:
  $A^2 = \{aa, ab, ac, ba, bb, bc, ca, cb, cc\}$.
- A^3 die Menge der Wörter der Länge 3:
  $A^3 = \{aaa, aab, aac, aba, abb, abc, aca, acb, acc, baa, bab, bac, bba, bbb, bbc, bca, bcb, bcc, caa, cab, cac, cba, cbb, cbc, cca, ccb, ccc\}$.
- usw.
- A^+ umfasst $A^1 \cup A^2 \cup A^3 \cup ...$
- A^* umfasst $A^0 \cup A^1 \cup A^2 \cup ...$
- $L = \{a, ab, abc, bc, c\}$ eine formale Sprache über dem Alphabet A. Ebenso ist $S = \{abababa, bcca, abba\}$ eine formale Sprache über dem Alphabet A. Eine formale Sprache muss **nicht** alle Wörter enthalten!

(Formale) Sprachen

Definition: 4.10 Wörter konkatenieren, Teilwörter

- Zwei Wörter u und v kann man **hintereinanderschreiben** (**konkatenieren**). Man erhält so das Wort uv.
- Gibt es zu einem Wort x Wörter u, v mit $x = uv$, so heißt u **Anfangsstück** (**Präfix**) von x.
- Gibt es zu einem Wort x Wörter u, v mit $x = uv$, so heißt v **Endstück** (**Suffix**) von x.
- Gibt es zu einem Wort x Wörter u, v, w mit $x = uvw$, so heißt v **Teilwort** von x.

Ein Präfix/Suffix ist also alles, was man als Anfang/Ende eines Wortes verstehen kann, ein Teilwort alles, was man als Teil eines Wortes verstehen kann. Diese Begriffe sind so weit auszulegen, dass auch das gesamte Wort ein Anfang/Ende/Teilwort sein kann. Außerdem ist das leere Wort ε Präfix/Suffix/Teilwort von jedem Wort.

Beispiel:

Sei $\mathbb{B}$ wie oben definiert die Menge der Binärziffern und sei $w = 011$ ein Wort über $\mathbb{B}^*$.

- Dann sind $\mathbb{B}^0 = \{\varepsilon\}$, $\mathbb{B}^1 = \{0, 1\}$ $\mathbb{B}^2 = \{00, 01, 10, 11\}$, $\mathbb{B}^3 = ...$
- $\mathbb{B}^* = \mathbb{B}^0 \cup \mathbb{B}^1 \cup \mathbb{B}^2 \cup ... = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, ...\}$
- Das Wort w hat die Präfixe $\varepsilon, 0, 01, 011$

- Das Wort w hat die Suffixe $\varepsilon, 1, 11, 011$
- Das Wort w hat die Teilwörter $\varepsilon, 1, 0, 01, 11$ und 011.

Lexikographische Ordnung
Die Ordnung der Buchstaben lässt sich unter der Verwendung der Definition 4.10 für Präfixe zur **lexikographischen Ordnung zwischen Wörtern** fortsetzen.

Definition: 4.12 Lexikografische Ordnung

Die **lexikografische Ordnung** $u < v$ **zwischen Wörtern** $u, v \in A^*$(**gebildet aus einem Alphabet A**) ist wie folgt definiert mit vollständiger Induktion nach **der Länge** $n := |u|$:

1. Induktionsanfang $n = |u| = 0$ (also $u = \epsilon$):
   Es gilt $\epsilon < v$ für alle Wörter $v \neq \epsilon$
2. Induktionsschritt $n \to n + 1$:
   Sei $|u| = n + 1$, $u = ax$ und $v = by$ für $a, b \in A$ und $x, y \in A^*$:
   (a) Ist $a < b$, so gilt $u < v$
   (b) Ist $a = b$, so gilt $u < v :\Leftrightarrow x < y$ (Rückführung auf kürzeres Wort x mit Länge n)
   (c) Ist $b < a$, so gilt $v < u$

Wenn zwei Wörter unterschiedlicher Länge gleich beginnen, steht also das kürzere vor dem längeren Wort. Ab dem Punkt wo sich die beiden Wörter unterscheiden, entscheiden die Suffixe, welches Wort gemäß lexikographischer Ordnung vor dem anderen steht. Diese Ordnung wird auch in Wörterbüchern, Lexika oder dem Duden so eingehalten (z.B. steht dort *Aal* vor *Aalen*, da *Aal* kürzer ist; ebenso *Tal* vor *Talent*, da *Tal* kürzer ist).

Beachte: Die lexikographische Ordnung vergleicht nur Zeichenfolgen in Wörtern und interpretiert diese nicht. Wenn man also das Alphabet der Binärziffern $\mathbb{B}$ zu Grunde legt und z.B. die Wörter 100, 11 und 1001 ordnen möchte, so werden diese **nicht** als Zahlen interpretiert, sondern einzig anhand der Zeichenfolgen sortiert: 100 < 1001 < 11.

Beispiel:

- $10 < 11 \overset{(2b)}{\Leftrightarrow} 0 < 1$, rechte Aussage ist wahr wegen (2a)
- $10 < 100 \overset{(2b)}{\Leftrightarrow} 0 < 00 \overset{(2b)}{\Leftrightarrow} \epsilon < 0$, rechte Aussage ist wahr wegen (1)

Abbildungen / Funktionen

Definition: 4.13 Abbildung

Seien A, B Mengen. Eine **Abbildung** oder **Funktion**

$$f : A \to B$$

ordnet jedem Element $a \in A$ ein eindeutiges Element $f(a) \in B$ zu.

- A ist der **Definitionsbereich** von f.
- B ist der **Wertebereich** von f.
- a nennt man **Operand** oder **Argument**.

Eine Abbildung/Funktion ordnet also jedem Element einer Menge ein Element einer (anderen) Menge zu. Dabei müssen die beiden Mengen nicht übereinstimmen.

Beispiel:

- Aus der Schule bekannte mathematische Funktionen:
  $\log(x)$, $\exp(x)$, 2^x, $\sin(x)$, $\cos(x)$, $\sqrt[n]{x}$ $(n \geqslant 2)$, x^n $(n \geqslant 2)$
- Aufrundungsfunktion `ceil(x)` nimmt beispielsweise reelle Zahlen und bildet sie auf ganze Zahlen ab. Dabei ist der Definitionsbereich $\mathbb{R}$, der Wertebereich $\mathbb{N}_0$ und der Operand x.

Codierung

Mithilfe der Abbildungen können wir uns nun der Codierung widmen. Die Codierung ist ein Prinzip, welches jedem Zeichen eines Alphabets ein Wort eines zweiten Alphabets zuordnet.

Definition: 4.15 Codierung

Seien A, B Alphabete. Eine **Codierung** ist eine Abbildung $c : A \to B^*$ mit $c(a_1) \neq c(a_2)$ für $a_1 \neq a_2$. Der Bildbereich $c(A) := \{c(a) \mid a \in A\}$ heißt **Code**. Jedes Element $b \in c(A)$ heißt **Codewort**.

Die Definition legt also fest, dass durch eine Abbildung $A \to B^*$ aus einem Zeichen eine Zuordnung zu mehreren Zeichen (Wort) gemacht werden kann. Außerdem wird durch $c(a1) \neq c(a2)$ sichergestellt, dass keine zwei Zeichen ein und demselben Wort zugeordnet werden. Der in der Definition beschriebene Bildbereich der Codierung $c(A)$ entspricht dem Wertebereich der Abbildung $A \to B^*$.

Beispiel:

Ein einfaches Beispiel für eine Codierung ist es, den Buchstaben des lateinischen Alphabets ihre entsprechende Position als Zahl zuzuordnen:
A→1, B→2, C→3, ..., Z→26.
Dabei ist A im Sinne der Definition das Alphabet der lateinischen Buchstaben

und B das Alphabet der Dezimalziffern $\mathbb{D}$. Die Codierung $\{A, ..., Z\} \to \mathbb{D}^*$ bildet jedes Element aus der Menge $\{A, ..., Z\}$ auf eine eindeutige Zahl aus $\mathbb{D}^*$ ab. Dabei ist der Bild-/Wertebereich $\{1, 2, ..., 26\}$ und 1, 2, ..., 26 sind die Codewörter.

Codierung von Wörtern: Man codiert ein Wort u, indem man die Codewörter der einzelnen Buchstaben hintereinander schreibt (induktive Definition nach der Länge des Wortes $n := |u|$):

1. Induktionsanfang $n = 0$: $c(\epsilon) := \epsilon$
2. Induktionsschritt $n \to n + 1$: Sei $|u| = n + 1$, $u = ax$ mit $a \in A$ und $x \in A^*$:
   $c(u) := c(a)c(x)$ (Rückführung auf Wort x mit Länge n)

Beispiel:

- Mit der im letzten Beispiel beschriebenen Codierung von lateinischen Buchstaben zu ihrer jeweiligen Position würde die Zeichenfolge INFO wie folgt codiert werden:
  $c(\text{INFO}) = c(\text{I})c(\text{N})c(\text{F})c(\text{O}) = 9\ 14\ 6\ 15$
  Im Rechner gibt es allerdings keine Abstände, somit würde die Codierung das Ergebnis als 914615 ablegen, was sich nicht mehr umkehren lässt.
- Beispiel für $c(A) := 1000001$ und $c(B) := 1000010$
  $c(AB) = c(A)c(B) = 10000011000010$

Decodierung

Die Decodierung kehrt die Codierung wieder um, so dass man wieder die ursprüngliche Eingabe erhält.

Definition: 4.16 Decodierung

Für ein Codewort $b \in B^*$ heißt $a \in A$ **Decodierung von** b, falls $c(a) = b$. Wir schreiben in diesem Fall $c^{-1}(b) := a$.

Wichtige Codierungsklassen

- Allgemeine Codierungsklassen: Seien A, B Alphabete. Eine Codierung
  - $c : A \to B$ heißt **Chiffrierung**
  - $c : A \to B^n$ (mit genau n Zeichen) heißt **Blockcodierung**
- Binär-Codierung: Sei A ein Alphabet. Eine Codierung
  - $c : A \to \mathbb{B}$ heißt **Binärcodierung (von A)** (hier wird nur ein Bit verwendet)
  - $c : A \to \mathbb{B}^n$ (n fest) heißt **n-Bit-Codierung (von A)** (hier werden n Bits verwendet)

Beispiel:

- Hätte man im obigen Beispiel des lateinischen Alphabets und der Position der Buchstaben eine führende Null für Werte kleiner 9 verwendet, hätte man eine Blockcodierung und könnte problemlos eine Decodierung durchführen:
  A→01, B→02, C→03, ..., Z→26.
  Somit $c(\mathrm{INFO}) = c(\mathrm{I})c(\mathrm{N})c(\mathrm{F})c(\mathrm{O}) = 09140615$ und
  $c^{-1}(09140615) = c^{-1}(09)c^{-1}(14)c^{-1}(06)c^{-1}(15) = \mathrm{INFO}$.
- Eine weitverbreite Chiffrierung ist die Caesarcodierung, bei der man Buchstaben codiert, indem man statt des ursprünglichen Buchstabens einen benachbarten Buchstaben mit festem Abstand verwendet. Zum Beispiel wählt man den Abstand 2, dann wird A → C, B → D, C → E, ..., Y → A, Z → B. Aus INFO wird somit KPHQ.
- Zur Speicherung von Zahlen und Zeichen im Arbeitsspeicher verwendet man n-Bit-Codierungen. Für jeden Datentyp ist ein festes $\mathbb{B}^n$ festgelegt, so dass die Codierung und Decodierung eindeutig ist. Details dazu folgen im Kapitel 4.6.

Zahlencodierungen

Jedes Wort in $\mathbb{B}^n$ bezeichnen wir als **Bitmuster (der Länge n)**. Als solche Bitmuster sind Zahlen und Zeichen im Rechner abgespeichert. Im Rest dieses Kapitels und im nächsten Kapitel werden wir verschiedene Codierungen kennenlernen. Um diese klar unterscheiden zu können, notieren wir diese als $c_{N,n}$: N gibt dabei an, auf welche Codierung wir uns beziehen, einen Überblick gibt Tabelle 1. n ist die Länge des Bitmusters, es handelt sich also um eine n-Bit-Codierung.

Table 1: Überblick über Codierungen in der *Informatik 1*.

$c_{N,n}$	**Beschreibung**	**Kap.**
$c_{2,n}$	Binärcodierung, entspricht Binärdarstellung einer Zahl, ggf. aufgefüllt mit führenden Nullen auf n Bits.	4.2
$c_{1K,n}$	1-Komplementcodierung, ähnlich zu $c_{2,n}$, aber auch für negative Zahlen geeignet	4.3
$c_{2K,n}$	2-Komplementcodierung, Verbesserung von $c_{1K,n}$	4.4
$c_{EX-q,n}$	Exzeß-q-Codierung, für ganze Zahlen, nötig für standardisierte Gleitkomma-Codierung	5.1
$c_{FK,k,n}$	Festkomma-Codierung mit k Nachkommastellen	5.2
$c_{GK,k,n}$	standardisierte Gleitkomma-Codierung mit $k-1$ Bit für Mantisse und $n-k$ Bit für Exponenten (und 1 Bit für Vorzeichen)	5.3

Eine ausführliche Übersicht zu den Codierungen wird als Merkblatt bereitgestellt. Üblicherweise interessiert uns das Bitmuster einer Zahl x, die in einer Codierung $c_{N,n}$ im Rechner abgespeichert ist. Diese fragen wir als $c_{N,n}(x)$ ab. Als Gegenstück dazu notieren wir zu einem Bitmuster b

folgendes, wenn wir es decodieren wollen: $(b)_{N,n}$. Wir wenden dann die Decodierung von $c_{N,n}$ an, also $c_{N,n}^{-1}(b) = (b)_{N,n}$.

Bei den Zahlencodierungen, die wir im Folgenden betrachten, stehen folgende Fragen im Mittelpunkt:

- Welche Zahlen sollen codiert werden?
- Wie werden negative Zahlen codiert?
- Wie kann man direkt mit den Codewörtern / Bitmustern rechnen?

4.2 Binärcodierung ganzer Zahlen in n Bit

Die Binärcodierung entspricht der Binärdarstellung einer Zahl, wie wir sie im Kapitel 3 kennengelernt haben. Der einzige Unterschied ist, dass die Zahlen immer genau n Stellen haben (so viele Stellen wie Bits verwendet werden) und daher evtl. mit Nullen aufgefüllt werden müssen.

Codierung

Definition: 4.18 Binärcodierung

Die **Binärcodierung** $c_{2,n} : \{0, ..., 2^n - 1\} \to \mathbb{B}^n$ ist definiert durch

$$c_{2,n}(x) := \underbrace{0 \quad \dots \quad 0}_{n-k-1\text{Bits}} b_k \dots b_0$$

mit $(b_k \dots b_0)_2 = x$.

Sollte eine Zahl nicht alle verfügbaren Stellen ausnutzen, so sehen wir, dass die genutzten Stellen 2er-Potenzen von 2^0 bis 2^k verwenden und somit $k + 1$ Stellen belegt werden. Dann gibt es $n - k - 1$ führende Nullen. Negative Zahlen bleiben unberücksichtigt und lassen sich nicht darstellen.

Beispiel:

- $c_{2,8}(7) = 00000111$
- $(00000111)_{2,8} = 7$

Decodierung

Es gilt

$$(b_{n-1}...b_1b_0)_{2,n} = \sum_{i=0}^{n-1} b_i \cdot 2^i$$

Beispiel:

Berechnungsschema:

2^{n-1}	128	64	32	16	8	4	2	1	2er-Potenzen
b_{n-1}	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	Ziffern

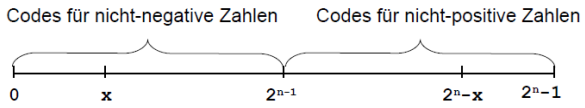
Codierung der Zahl $(00010110)_{2,8}$:

128	64	32	16	8	4	2	1	2er-Potenzen
0	0	0	1	0	1	1	0	Ziffern

$1 \cdot 16 + 1 \cdot 4 + 1 \cdot 2 = 16 + 4 + 2 = 22$

4.3 1-Komplementcodierung ganzer Zahlen (in n Bit)

Nachdem die Binärcodierung keine negativen Zahlen darstellen kann, hat man in den frühen Großrechnern (z.B. PDP-1, 1959 oder UNIVAC 1100 Serie, 1960er) die sog. 1-Komplementcodierung verwendet. Die verfügbaren Codes werden hierbei in zwei gleich große Teile geteilt, so dass die eine Hälfte für positive und die andere Hälfte für negative Zahlen verwendet werden kann:



Die positiven Zahlen werden genau wie in der Binärcodierung gespeichert, allerdings steht das höchste Bit nicht zur Verfügung. Nur wenn dieses 0 ist, ist die Zahl positiv. Negative Zahlen werden also daran erkannt, dass das höchste Bit 1 ist.

Gleichzeitig sind negative Zahlen **nicht** einfach wie in der Binärcodierung und mit führender 1 gespeichert, sondern invertiert, d.h. alle Bits werden gekippt:

- Eine 0 wird zu einer 1
- Eine 1 wird zu einer 0

Beispiel:

Kippt man die Binärzahl
100110011
wird daraus:
011001100

Da sich Bits in Hardware sehr leicht kippen lassen, vereinfacht dieses Vorgehen die Hardware gegenüber einer Lösung, wo die Zahl und ihr Vorzeichen getrennt betrachtet werden müssten. Auch beim Rechnen zeigen sich Vorteile, wie wir in Kapitel 4.5 sehen werden.

Wenn wir über 1-Komplement-Codierung sprechen, notieren wir diese als $c_{1K,n}$. Formal definieren wir sie wie folgt:

Definition: 1-Komplement-Codierung

Die 1-**Komplement-Codierung** (1K-**Codierung**)
$c_{1K,n} : \{-(2^{n-1}-1), ..., 0, 1, ..., 2^{n-1}-1\} \to \mathbb{B}^n$ ist definiert durch

- $c_{1K,n}(x) := c_{2,n}(x)$, falls $0 \leqslant x < 2^{n-1}$
  (Codewörter zu nicht-negativen Zahlen beginnen mit 0)
- $c_{1K,n}(x) := c_{2,n}((2^n-1)+x)$ falls $-2^{n-1} \leqslant x \leqslant 0$
  (Codewörter zu nicht-positiven Zahlen beginnen mit 1)

Für $0 \leqslant x < 2^{n-1}$ heißt $\overline{x} := (2^n-1) - x$ das 1-**Komplement von** x
Es gilt für $0 \leqslant x < 2^{n-1}$: $c_{1K,n}(-x) = c_{2,n}(\overline{x})$

Schnelle Komplement-Bildung
Berechne das Codewort von $-x$ aus dem Codewort von x.
Sei $x \geqslant 0$ mit $c_{1K,n}(x) = b_{n-1}\dots b_0$:

$$\begin{aligned} c_{1K,n}(-x) &= c_{2,n}(2^n-1) - c_{2,n}(x) \\ &= \underbrace{1 \quad \dots \quad 1}_{n\text{Bits}} - b_{n-1}\dots b_0 \\ &= (1-b_{n-1})\dots(1-b_0) \end{aligned}$$

Das Codewort von $-x$ ensteht aus dem Codewort von x durch **Kippen aller Bits**

Beispiel:

- $c_{1K,4}(7) = c_{2,4}(7) = 0111$
- $(0101)_{1K,4} = (0101)_{2,4} = 4+1 = 5$
- $c_{1K,4}(-7) = 1000$
  (Kippen aller Bits von 0111)
- $(1101)_{1K,4} = -(0010)_{2,4} = -2$
  (Kippen aller Bits von 1101)

Beispiel: 1K-Darstellungen für $n = 4$

$c_{1K,4}(0) = 0000$	$c_{1K,4}(-0) = 1111$
$c_{1K,4}(1) = 0001$	$c_{1K,4}(-1) = 1110$
...	$c_{1K,4}(-2) = 1101$
$c_{1K,4}(6) = 0110$	...
$c_{1K,4}(7) = 0111$	$c_{1K,4}(-7) = 1000$

Decodierung vom 1-Komplement
Es gilt:

$$(b_{n-1}...b_1b_0)_{1K,n} = -b_{n-1} \cdot (2^{n-1} - 1) + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

Beispiel:

Codierung der Zahl $(11001101)_{1K,8}$:

-127	64	32	16	8	4	2	1	2er-Potenzen
1	1	0	0	1	1	0	1	Ziffern

$-127 + 64 + 8 + 4 + 1 = -50$

Alternativ:

Kpl.	64	32	16	8	4	2	1	2er-Potenzen
0	0	1	1	0	0	1	0	Ziffern

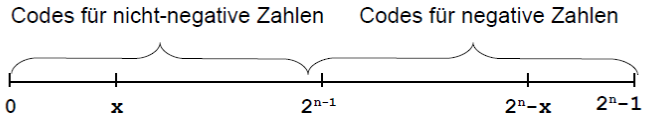
$(11001101)_{1K,8} = -(00110010)_{2,8} = -(32 + 16 + 2) = -50$

4.4 2-Komplementcodierung ganzer Zahlen (in n Bit)

Die 1-Komplementcodierung hat einen entscheidenden Nachteil: Die beiden Bitmuster $(0000)_{1K,4}$ und $(1111)_{1K,4}$ stehen beide für die Zahl Null. Daher muss beispielsweise für den Vergleich, ob ein Ergebnis 0 ist, zusätzlicher Aufwand betrieben werden – dieser verzögert die Berechnungen oder erfordert zusätzliche Hardware. Um dem entgegen zu wirken, hat sich in den Rechnern seit den 1980ern bis heute die 2-Komplementcodierung durchgesetzt. Diese unterscheidet sich von der 1-Komplementcodierung insbesondere durch:

- Die Zahl 0 besitzt nur noch eine Codierung: 00...00
- Binärzahlen, die mit einer 1 beginnen, sind alle ausnahmslos negative Zahlen

Die Codes sind in der 2-Komplementcodierung wie folgt verteilt:



Positive Zahlen entsprechen weiterhin ihrer Binärcodierung. Negative Zahlen sind nun um 1 in Richtung der 0 verschoben, so dass etwa $(1111)_{2K,4}$ der Zahl -1 entspricht.

Wenn wir über 2-Komplement-Codierung sprechen, notieren wir diese als $c_{2K,n}$. Formal definieren wir sie wie folgt:

Definition: 2-Komplement-Codierung

Die 2-**Komplement-Codierung** 2K-**Codierung**)
$c_{2K,n} : \{-2^{n-1}, ..., 0, 1, ..., 2^{n-1} - 1\} \to \mathbb{B}^n$ ist definiert durch

- $c_{2K,n}(x) := c_{2,n}(x)$, falls $0 \leqslant x < 2^{n-1}$
  (Codewörter zu nicht-negativen Zahlen beginnen mit 0)
- $c_{2K,n}(x) := c_{2,n}(2^n + x)$ falls $-2^{n-1} \leqslant x < 0$
  (Codewörter zu negativen Zahlen beginnen mit 1)

Für $0 \leqslant x < 2^{n-1}$ heißt $\overline{x} := 2^n - x$ das 2-**Komplement von** x
Es gilt für $0 \leqslant x < 2^{n-1}$: $c_{2K,n}(-x) = c_{2,n}(\overline{x})$

Schnelle Komplement-Bildung
Um „schnell“ die negative Zahl $-x$ in die 2K-Darstellung zu codieren, kann man auch den Trick des Bitflips nutzen:
Das Codewort von $-x$ entsteht aus dem Codewort von x durch

1. Kippen aller Bits
   - 0 wird zu 1
   - 1 wird zu 0
2. Addition von 1

Eine formalere bzw. mathematischere Schreibweise dafür lautet wie folgt:
Sei $x \geqslant 0$ mit $c_{2K,n}(x) = b_{n-1} \dots b_0$:

$$\begin{aligned} c_{2K,n}(-x) &= c_{2,n}(2^n) - c_{2,n}(x) \\ &= c_{2,n}(2^n - 1) - c_{2,n}(x) + c_{2,n}(1) \\ &= \underbrace{1 \quad \dots \quad 1}_{n\text{Bits}} - b_{n-1} \dots b_0 + \underbrace{0 \quad \dots \quad 0}_{n-1\text{Bits}} 1 \\ &= (1 - b_{n-1}) \dots (1 - b_0) + \underbrace{0 \quad \dots \quad 0}_{n-1\text{Bits}} 1 \end{aligned}$$

Beispiel:

- $c_{2K,4}(7) = c_{2,4}(7) = 0111$
- $(0101)_{2K,4} = (0101)_{2,4} = 1 + 4 = 5$
- $c_{2K,4}(-7) = 1000 + 0001 = 1001$
  (Kippen aller Bits von 0111 und Addition von 1)
- $(1101)_{2K,4} = -(0011)_{2,4} = -3$
  (Subtrahiere 1 von 1101 und kippe dann alle Bits)

Beispiel: 2K-Darstellungen für $n = 4$

$$\begin{aligned} c_{2K,4}(0) &= 0000 & c_{2K,4}(-1) &= 1111 \\ c_{2K,4}(1) &= 0001 & c_{2K,4}(-2) &= 1110 \\ &\dots & &\dots \\ c_{2K,4}(6) &= 0110 & c_{2K,4}(-7) &= 1001 \\ c_{2K,4}(7) &= 0111 & c_{2K,4}(-8) &= 1000 \end{aligned}$$

Decodierung vom 2-Komplement

Will man nun eine Zahl vom 2-Komplement in das Dezimalsystem umwandeln, kann man entweder folgenden Satz anwenden:

Satz 1. *Es gilt*

$$(b_{n-1}...b_1b_0)_{2K,n} = -b_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} b_i \cdot 2^i$$

Oder man greift wieder auf das Bitflippen zurück: Liegt eine positive Zahl vor (d.h. die Binärzahl fängt mit einer 0 an), rechne sie ganz normal vom Binär- ins Dezimalsystem. Liegt eine negative Zahl vor (d.h. die Binärzahl fängt mit einer 1 an), dann ziehe davon 1 ab und flippe alle Bits (oder umgekehrt: erst Bits flippen und dann 1 addieren). Berechne nun diese Zahl im Dezimalsystem, womit man eine positive Zahl bekommt. Da es aber ursprünglich eine negative Zahl war, schreibt man abschließend ein Minuszeichen davor.

Beispiel:

Berechnung von $(11001101)_{2K,8}$:

−128	64	32	16	8	4	2	1	2er-Potenzen
1	1	0	0	1	1	0	1	Ziffern

$-128 + 64 + 8 + 4 + 1 = -51$

oder mithilfe des Bitflips (die +1 nicht vergessen):

Kpl.	64	32	16	8	4	2	1	2er-Potenzen
0	0	1	1	0	0	1	1	Ziffern

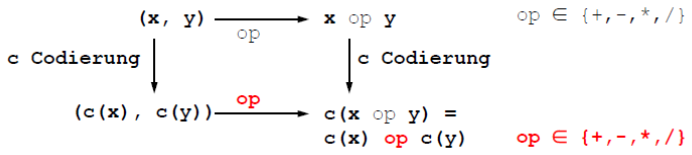
$(11001101)_{2K,8} = -(00110011)_{2,8} = -(32 + 16 + 2 + 1) = -51$

4.5 Rechnen in Komplement-Darstellungen

Rechnen in Komplement-Darstellungen
Natürlich will man auch im 1- und 2-Komplement arithmetische Operationen wie Addition, Subtraktion, etc. ausführen.
Hierfür muss aber gelten, dass die Rechnung kommutativ ist. Das bedeutet, es macht keinen Unterschied, ob ich zuerst die Addition im Dezimalsystem ausführe und die Zahl danach umwandle, oder zuerst zwei Zahlen umwandle und erst danach addiere.

> Beispiel:
>
> Addition der Zahlen 3 und 4 in 2K-Codierung.
> Dann muss folgendes gelten:
>
> $$c_{2K,4}(3) + c_{2K,4}(4) \Leftrightarrow c_{2K,4}(3+4)$$

Diese Fragestellung lässt sich auch mithilfe einer kleinen Grafik veranschaulichen. Hierbei ist op eine beliebige Operation (also Addition, Subtraktion, etc.).



Wie muss nun op aussehen, damit das Diagramm **kommutativ** ist?
Das Diagramm **kommutiert** (ist **kommutativ**), falls beide Wege von der linken oberen zur rechten unteren Ecke zum gleichen Ergebnis führen (egal ob zuerst c und dann op angewendet wird, oder umgekehrt):

$$c(x \,\mathrm{op}\, y) = c(x) \,\mathrm{op}\, c(y)$$

4.5.1 Bereichsüberlauf

Bevor wir uns der Addition / Subtraktion widmen können, muss noch die Frage geklärt werden: Was ist ein Bereichsüberlauf?

Ein Bereichsüberlauf ist ein Phänomen, dass bei Operationen auftreten kann. Dieser Bereichsüberlauf führt nämlich zu verfälschten Ergebnissen.
Der Grund dafür ist, dass man nicht mehr genug Bits besitzt um die benötigte Zahl zu codieren. Ein Beispiel, das man zur Übung nachrechnen kann: Versuche die Zahl 20 mit nur 4 Bits zu codieren. Man merkt schnell, dass das nicht möglich ist.
Addiert man nun bspw. zwei Zahlen, wie $(1110)_{2,4} + (0110)_{2,4}$, kommt es zum Bereichsüberlauf. Das Ergebnis wäre eigentlich $(10100)_{2,4}$. Da die Binärzahl allerdings nur 4 Bits besitzen kann, muss überlegt werden, wie mit der führenden 1 umgegangen wird.

4.5.2 Addition / Subtraktion von 1K-Darstellungen

Definition: 4.31 Addition

$c_{1K,n}(x) \oplus_{1K,n} c_{1K,n}(y) := (c_{1K,n}(x) + c_{1K,n}(y)) \bmod (2^n - 1)$

- $\oplus_{1K,n}$: Addition auf 1K-Darstellungen in n Bit
- $+$: Addition auf Binärzahlen
- mod $(2^n - 1)$: **Addition des Überlaufs** (führende 1 ignorieren und an letzter Stelle addieren, falls Ergebnis $n + 1$ Stellen hat)

Wie oben schon beschrieben, muss auf den Bereichsüberlauf geachtet werden. Um das Problem von „zu vielen Bits“zu umgehen, wird im Falle des Bereichsüberlaufes die führende 1 entfernt und zur Binärzahl eine 1 hinzu addiert. Der Grund dafür liegt an den zwei Arten, die Zahl 0 darzustellen. Würde man beispielsweise nur die führende 1 entfernen, kommt es zu einem verfälschten Ergebnis.

Beispiel:

Beispiel bei der Addition mit Bereichsüberlauf:

$$(1100)_{1K,4} + (1100)_{1K,4} = (11000)_{1K,4}$$

Da die Zahl aus 5 Bits besteht, allerdings nur 4 besitzen darf, hat man hier das Problem des Bereichsüberlaufes. Dazu schneidet man die führende 1 ab

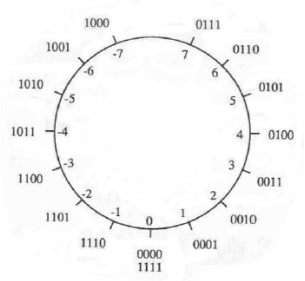
$$(11000)_{1K,4} \Rightarrow (1000)_{1K,4}$$

und addiert zur Zahl die 1

$$(1000)_{1K,4} + (0001)_{1K,4} = (1001)_{1K,4}$$

Veranschaulichung auf dem Zahlenring:

- Addition einer positiven Zahl y:
  y Schritte gegen Uhrzeigersinn
- Addition einer negativen Zahl y:
  y Schritte im Uhrzeigersinn
- Bei Überlauf muss man an der 0 vorbei, d.h. an 0000 und 1111. Dies ist nicht ein Schritt (wie z.B. von 3 zu 4), sondern 2 Schritte. Daher muss bei einem Überlauf zusätzlich 1 addiert werden.



Addition

$c_{1K,n}(x) \oplus_{1K,n} c_{1K,n}(y) := (c_{1K,n}(x) + c_{1K,n}(y)) \bmod (2^n - 1)$

- $\oplus_{1K,n}$: Addition auf 1K-Darstellungen in n Bit
- $+$: Addition auf Binärzahlen
- $\mod(2^n - 1)$: **Addition des Überlaufs** (führende 1 ignorieren und an letzter Stelle addieren, falls Ergebnis $n + 1$ Stellen hat)

> **Beispiel: Addition**
>
> - $c_{1K,4}(4) \oplus_{1K,4} c_{1K,4}(3) = (0100 + 0011) \mod (2^4 - 1) = 0111 \mod (2^4 - 1) = 0111$
> - $c_{1K,4}(4) \oplus_{1K,4} c_{1K,4}(-3) = (0100 + 1100) \mod (2^4 - 1) = 10000 \mod (2^4 - 1) = 0000 + 0001 = 0001$
> - $c_{1K,4}(-4) \oplus_{1K,4} c_{1K,4}(3) = (1011 + 0011) \mod (2^4 - 1) = 1110 \mod (2^4 - 1) = 1110$
> - $c_{1K,4}(-4) \oplus_{1K,4} c_{1K,4}(-3) = (1011 + 1100) \mod (2^4 - 1) = 10111 \mod (2^4 - 1) = 0111 + 0001 = 1000$

Subtraktion
Subtraktion $\ominus_{1K,n}$ wird realisiert, indem das Komplement (die invertierte Zahl) addiert wird. Dies führt zum richtigen Ergebnis, da die invertierte Bitfolge dem negativen Wert der ursprünglichen Bitfolge entspricht. Da sich Hardware zum Bits flippen sehr einfach umsetzen lässt, ist dies ein sehr eleganter Weg und erspart zusätzliche Hardware für Subtraktionsberechnungen.

> **Beispiel: Subtraktion**
>
> Um zwei 1K-Codierte Zahlen zu subtrahieren, schreibt man es in eine Addition um. Hier gilt zu beachten:
> $3 - 4 = 3 + (-4)$
> Das bedeutet: Man addiert zur 3 die -4
>
> $$\begin{aligned} & c_{1K,4}(3) \ominus_{1K,4} c_{1K,4}(4) \\ = \; & c_{1K,4}(3) \oplus_{1K,4} c_{1K,4}(-4) \\ = \; & (0011 + 1011) \mod (2^4 - 1) = 1110 \mod (2^4 - 1) = 1110 \end{aligned}$$

Bereichsüberlauf bei 1K-Addition
Überlauf im positiven Bereich:
Ein **Bereichsüberlauf im positiven Bereich** liegt vor falls $0 \leqslant x, y$ und $2^{n-1} \leqslant x + y$.
In einfachen Worten bedeutet das: Ein Bereichsüberlauf kann passieren, falls zwei positive Zahlen addiert werden und der binäre Zahlenwert dieser Addition größer wird als die größte darstellbare Zahl.
In diesem Fall hat die Addition das folgende Ergebnis:

$$c_{1K,n}(x) \oplus_{1K,n} c_{1K,n}(y) = c_{1K,n}(x + y - (2^n - 1))$$

Überlauf im negativen Bereich:
Ein **Bereichsüberlauf im negativen Bereich** liegt vor falls $x, y \leqslant 0$ und $x + y \leqslant -2^{n-1}$.

In diesem Fall hat die Addition das folgende Ergebnis:

$$c_{1K,n}(x) \oplus_{1K,n} c_{1K,n}(y) = c_{1K,n}(x + y + (2^n - 1))$$

Beispiel:

- $c_{1K,4}(4) \oplus_{1K,4} c_{1K,4}(5) = (0100 + 0101) \bmod (2^4 - 1) = 1001 = c_{1K,4}(-6)$
- $c_{1K,4}(-4) \oplus_{1K,4} c_{1K,4}(-5) = (1011 + 1010) \bmod (2^4 - 1) = 0110 = c_{1K,4}(6)$

4.5.3 Addition / Subtraktion von 2K-Darstellungen

Definition: 4.28 Addition

$c_{2K,n}(x) \oplus_{2K,n} c_{2K,n}(y) := (c_{2K,n}(x) + c_{2K,n}(y)) \bmod 2^n$

- $\oplus_{2K,n}$: Addition auf 2K-Darstellungen in n Bit
- $+$: Addition auf Binärzahlen
- $\bmod 2^n$: **Ignorieren des Überlaufs** (führende 1 ignorieren, falls Ergebnis $n + 1$ Stellen hat)

Ähnlich zur 1K muss auch bei der 2K-Arithmetik ein möglicher Bereichsüberlauf beachtet werden. Hierbei verwirft man einfach die führende 1, die zuviel ist.

Beispiel:

Beispiel bei der Addition mit Bereichsüberlauf:

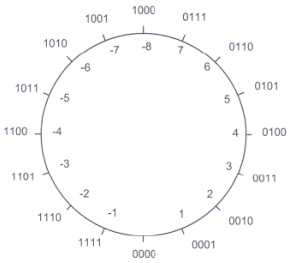
$$(1100)_{2K,4} + (1100)_{2K,4} = (11000)_{2K,4}$$

Da die Zahl aus 5 Bits besteht, allerdings nur 4 besitzen darf, hat man hier das Problem des Bereichsüberlaufes. Dazu verwirft man die führende 1.

$$(11000)_{2K,4} \Rightarrow (1000)_{2K,4}$$

Veranschaulichung auf dem Zahlenring:

- Addition einer positiven Zahl y:
  y Schritte gegen Uhrzeigersinn
- Addition einer negativen Zahl y:
  y Schritte im Uhrzeigersinn
- **Subtraktion** $\ominus_{2K,n}$:
  Addition des Komplements
  (vgl. 1K-Codierung)
- Keine Doppelbelegungen
  $\Rightarrow$ keine +1 bei Überlauf nötig



Addition:
$c_{2K,n}(x) \oplus_{2K,n} c_{2K,n}(y) := (c_{2K,n}(x) + c_{2K,n}(y)) \bmod 2^n$

- $\oplus_{2K,n}$: Addition auf 2K-Darstellungen in n Bit
- $+$: Addition auf Binärzahlen
- $\bmod 2^n$: **Ignorieren des Überlaufs** (führende 1 ignorieren, falls Ergebnis $n+1$ Stellen hat)

Beispiel: Addition

- $c_{2K,4}(4) \oplus_{2K,4} c_{2K,4}(3) = (0100+0011) \bmod 2^4 = 0111 \bmod 2^4 = 0111$
- $c_{2K,4}(4) \oplus_{2K,4} c_{2K,4}(-3) = (0100 + 1101) \bmod 2^4 = 10001 \bmod 2^4 = 0001$
- $c_{2K,4}(-4) \oplus_{2K,4} c_{2K,4}(3) = (1100 + 0011) \bmod 2^4 = 1111 \bmod 2^4 = 1111$
- $c_{2K,4}(-4) \oplus_{2K,4} c_{2K,4}(-3) = (1100+1101) \bmod 2^4 = 11001 \bmod 2^4 = 1001$
- Bereichsüberlauf: $c_{2K,4}(4) \oplus_{2K,4} c_{2K,4}(5) = 1001 = c_{2K,4}(-7)$
- Bereichsüberlauf: $c_{2K,4}(-4) \oplus_{2K,4} c_{2K,4}(-5) = 0111 = c_{2K,4}(7)$

Beispiel: Subtraktion

Die Subtraktion von 2K-Codierten Zahlen verläuft analog wie in der 1K-Darstellung beschrieben wurde. Um zwei Zahlen zu subtrahieren, schreibt man das in eine Addition um.

$$c_{2K,4}(4) \ominus_{2K,4} c_{2K,4}(3) = c_{2K,4}(4) \oplus_{2K,4} c_{2K,4}(-3)$$

Bereichsüberlauf bei 2K-Addition
Überlauf im positiven Bereich:
Ein **Bereichsüberlauf im positiven Bereich** liegt vor, falls $0 \leqslant x, y$ und $2^{n-1} \leqslant x+y$. In diesem Fall hat die Addition das folgende Ergebnis:

$$c_{2K,n}(x) \oplus_{2K,n} c_{2K,n}(y) = c_{2K,n}(x + y - 2^n)$$

Überlauf im negativen Bereich
Ein **Bereichsüberlauf im negativen Bereich** liegt vor, falls $x, y \leqslant 0$ und $x + y < -2^{n-1}$. In diesem Fall hat die Addition das folgende Ergebnis:

$$c_{2K,n}(x) \oplus_{2K,n} c_{2K,n}(y) = c_{2K,n}(x + y + 2^n)$$

4.6 Ganzzahlige Datentypen in C

Die bis hierhin beschriebenen Codierungen werden in der Programmiersprache C für die ganzzahligen Datentypen verwendet. Schauen wir uns diese also genauer an.

Datentyp `int`
Speicherbedarf und Codierung des Datentyps `int` sind systemabhängig.
Datentyp `int` auf alten Systemen:

- Speicherbedarf: 2 Byte = 16 Bit (Mindestwert nach C89)
- Codierung: $c_{1K,16}$ oder $c_{2K,16}$
- Wertebereich: Es sind 2^{16} Zahlen darstellbar
- Grenzen des Wertebereichs abhängig von der Codierung

Datentyp `int` auf aktuellen Systemen:

- Speicherbedarf: 4 Byte = 32 Bit
- Codierung: $c_{2K,32}$
- Wertebereich: von -2^{31} bis $2^{31} - 1$ (jeweils einschließlich)

Datentyp `char`

- Speicherbedarf: 1 Byte = 8 Bit
- Wertebereich compilerabhängig:
  von 0 bis 255 (vorzeichenlos, Codierung $c_{2,8}$)
  oder von -128 bis 127 (vorzeichenbehaftet, Codierung $c_{2K,8}$)
- Nur die Werte $0 - 127$ sind für ASCII-Zeichen verwendbar!

Weitere Ganzzahltypen
Datentyp `short`:

- Speicherbedarf: 2 Byte = 16 Bit
- Codierung: $c_{2K,16}$
- Wertebereich: von -2^{15} bis $2^{15} - 1$ (jeweils einschließlich)
- Umwandlungsangabe für `printf`: `%hi`

Datentyp `long`:

- Speicherbedarf: mindestens 4 Byte = 32 Bit
- Codierung: $c_{2K,32}$
- Wertebereich: mindestens von -2^{31} bis $2^{31} - 1$ (jeweils einschließlich)
- Schreibweise Konstanten: mit `L` beenden (Beispiel: `1L`)
- Umwandlungsangabe für `printf`: `%li`

Table 2: Übersicht über bitweise Operationen.

Bedeutung	Operationszeichen	C-Operator	Definition
Nicht	$\neg$	`~`	$\neg 0 := 1$ $\neg 1 := 0$
Und	$\wedge$	`&`	$1 \wedge 1 := 1$ $1 \wedge 0 := 0$ $0 \wedge 1 := 0$ $0 \wedge 0 := 0$
Oder	$\vee$	`|`	$1 \vee 1 := 1$ $1 \vee 0 := 1$ $0 \vee 1 := 1$ $0 \vee 0 := 0$
Exklusives Oder	`xor`	`xor`	1 xor 1 := 0 0 xor 1 := 1 1 xor 0 := 1 0 xor 0 := 0

Allgemeines zu Speicherbedarf und Wertebereich

- Der Speicherbedarf und die Grenzen des Wertebereichs eines Ganzzahl-Datentyps sind Systemabhängig und können über Bibliotheks-Konstanten in `limits.h` abgefragt werden.
- Wenn in einer Berechnung oder durch eine Benutzereingabe der Wertebereich verlassen wird, kommt es zu Programm- oder Rechenfehlern.
- Der Speicherbedarf einer Variable `x` kann mit `sizeof(x)` abgefragt werden.
- Der Speicherbedarf eines Datentyps `T` kann mit `sizeof(T)` abgefragt werden.
- Der Compiler ordnet jeder Variablen in einem Programm einen festen Speicherbereich zu, und zwar durch Festlegung der **Adresse der ersten Speicherzelle** dieses Bereichs.

4.7 Bitweise Operatoren in C

Bitweise Operatoren werden insbesondere in der hardwarenahen Programmierung eingesetzt. Auf Mikrocontrollern kann man oft Komponenten ein-/ausschalten, indem man das jeweilige Bit ändert (z.B. eine LED ein-/auschalten). Hier ist es wichtig, das passende Bit zu erwischen und nicht die Bits daneben mitzunehmen. Auch auf performanten Plattformen lassen sich gelegentlich teure und aufwendige Operationen durch bitweise Operationen ersetzen.

Tabelle 2 gibt einen Überblick über bitweise Operationen, welche bereits aus anderen Fächern bekannt sein sollten. In C werden diese Operationen auf Bitmustern angewandt, d.h. auf den Zahlen 0 und 1. Dabei interpretieren wir 0 und 1 als Wahrheitswerte, d.h.

- 0 steht für **falsch**
- 1 steht für **wahr**

Die in der Tabelle gezeigten C-Operatoren beschreiben wir im Folgenden näher.

4.7.1 C-Operatoren auf Bitmustern

Seien x, y ganzzahlige Variablen mit den Codierungen $x = (x_{n-1} \dots x_0)_{2K,n}$ und $y = (y_{n-1} \dots y_0)_{2K,n}$, d.h. x und y sind im 2-Komplement codierte Zahlen. Es lassen sich nun verschiedene logische Operatoren auf diese Binärzahlen anwenden.

Bitweise Negation
Die Bitweise Negation entspricht dem Bitflip. Formal notiert:

$$\begin{aligned} \sim x &:= ((\neg x_{n-1}) \dots (\neg x_0))_{2K,n} \\ &= ((1 - x_{n-1}) \dots (1 - x_0))_{2K,n} \end{aligned}$$

Beispiel:

$\sim (0011)_{2K,4}$ wird zu $(1100)_{2K,4}$

Beispiel:

Als C-Code:

```
int main(void)
{
    /* Gibt den Wert -5 aus. */
    printf("%i", ~4);

    return 0;
}
```

Linksshift
Beim Linksshift (<< als C-Operator) wird ganz rechts in der Binärzahl eine 0 eingefügt, das Bit ganz links wird weggeschnitten. Das bedeutet auch, dass mit einem Linksshift die Zahl verdoppelt wird (Beweis wird dem Leser bzw. der Leserin überlassen ;))

Beispiel:

- Linksshift um 1:

$$(0011)_{2K,4} << 1 \Rightarrow (0110)_{2K,4}$$

- Linksshift um 2 fügt zwei 0en rechts ein:

$$(0011)_{2K,4} << 2 \Rightarrow (1100)_{2K,4}$$

Beispiel:

Als C-Code:

```c
int main(void)
{
    /* Gibt den Wert 8 aus. */
    printf("%i", 4 << 1);

    /* Gibt den Wert 16 aus. */
    printf("%i", 4 << 2);

    return 0;
}
```

Formal bedeutet das nun:

$$x << k := x \cdot 2^k$$
$$= (x_{n-1-k} \dots x_0 0 \dots 0)_{2K,n}$$

für $0 \leqslant x < 2^{n-k}$ und $k \in \mathbb{N}$

Beispiel:

Weitere Beispiele:
$3 << 2 = 12$ (Zahl k-mal verdoppeln)
$1 << k = 2^k$ (Codierung dieser Zahl hat genau an $k+1$-letzter Stelle eine 1
$\sim (1 << k)$ (Codierung dieser Zahl hat genau an $k+1$-letzter Stelle eine 0

Rechtsshift

Der Rechtsshift (>> als C-Operator) verhält sich ähnlich zum Linksshift. Der Unterschied ist jedoch, dass der Binärwert an ganz rechter Stelle weggeschnitten wird, und eine 0 ganz links eingefügt wird.

Beispiel:

- Rechtsshift um 1:
  $(0011)_{2K,4} >> 1 \Rightarrow (0001)_{2K,4}$
- Rechtsshift um 2:
  $(1001)_{2K,4} >> 2 \Rightarrow (0010)_{2K,4}$

Beispiel:

Als C-Code:

```c
int main(void)
{
    /* Gibt den Wert 4 aus. */
    printf("%i", 8 >> 1);
```

```
    /* Gibt den Wert 2 aus. */
    printf("%i", 5 >> 1);

    return 0;
}
```

Formal lautet der Rechtsshift nun:

$$x >> k := x \div 2^k$$
$$= (0 \dots 0 x_{n-1} \dots x_k)_{2K,n}$$

für $x \geqslant 0$ und $k \in \mathbb{N}$
(Rechtsshift Verhalten für $x < 0$ compiler-abhängig)

Beispiel:

weitere Beispiele:
$20 >> 2 = 5$
$7 >> 1 = 3$ (Zahl k-mal ganzzahlig halbieren)

Bitweises Oder
Das bitweise Oder (| als C-Operator) verknüpft elementweise die Bitwerte: Jedes Bit der ersten Eingabe wird mit dem Bit an der selben Position der zweiten Eingabe ODER-verknüpft.

Formal lässt sich ODER folgendermaßen beschreiben:
$x \mid y := ((x_{n-1} \vee y_{n-1}) \dots (x_0 \vee y_0))_{2K,n}$

Beispiel:

$(0011)_{2K,4} \mid (1101)_{2K,4} = (1111)_{2K,4}$

Beispiel:

Als C-Code:

```
int main(void)
{
    /* Gibt den Wert 13 aus. */
    printf("%i", 4 | 9);

    return 0;
}
```

Bitweises Und

Das bitweise Und (`&` als C-Operator) verknüpft elementweise die Bitwerte: Jedes Bit der ersten Eingabe wird mit dem Bit an der selben Position der zweiten Eingabe UND-verknüpft.

Formal lässt sich das Und folgendermaßen beschreiben:
$x \,\&\, y := ((x_{n-1} \wedge y_{n-1}) \dots (x_0 \wedge y_0))_{2K,n}$

> Beispiel:
>
> $(0011)_{2K,4} \,\&\, (1101)_{2K,4} = (0001)_{2K,4}$

> Beispiel:
>
> Als C-Code:

```c
int main(void)
{
    /* Gibt den Wert 0 aus. */
    printf("%i", 4 & 9);

    /* Gibt den Wert 4 aus. */
    printf("%i", 4 & 7);

    return 0;
}
```

Bitweises XOR

Zu guter Letzt gibt es noch das XOR (`^` als C-Operator). Auch dieser Operator verknüpft die Bitwerte elementweise: Jedes Bit der ersten Eingabe wird mit dem Bit an der selben Position der zweiten Eingabe XOR-verknüpft.

Formal lässt sich das XOR folgendermaßen beschreiben:
$x \,\hat{}\, y := (x \mid y) \,\&\, (\sim (x \,\&\, y))$

> Beispiel:
>
> $(0011)_{2K,4} \,\hat{}\, (1101)_{2K,4} = (1110)_{2K,4}$

> Beispiel:
>
> Als C-Code:

```c
int main(void)
{
    /* Gibt den Wert 13 aus. */
    printf("%i", 4 ^ 9);
```

```
    /* Gibt den Wert 3 aus. */
    printf("%i", 4 ^ 7);

    return 0;
}
```

4.8 Literaturverzeichnis

siehe Foliensatz.