Wieso benutzen wir auf dem Computer das Dualsystem?

Fakt 1: Wir möchten einer Maschine das Rechnen mit Zahlen beibringen

Was steht uns zur Verfügung?

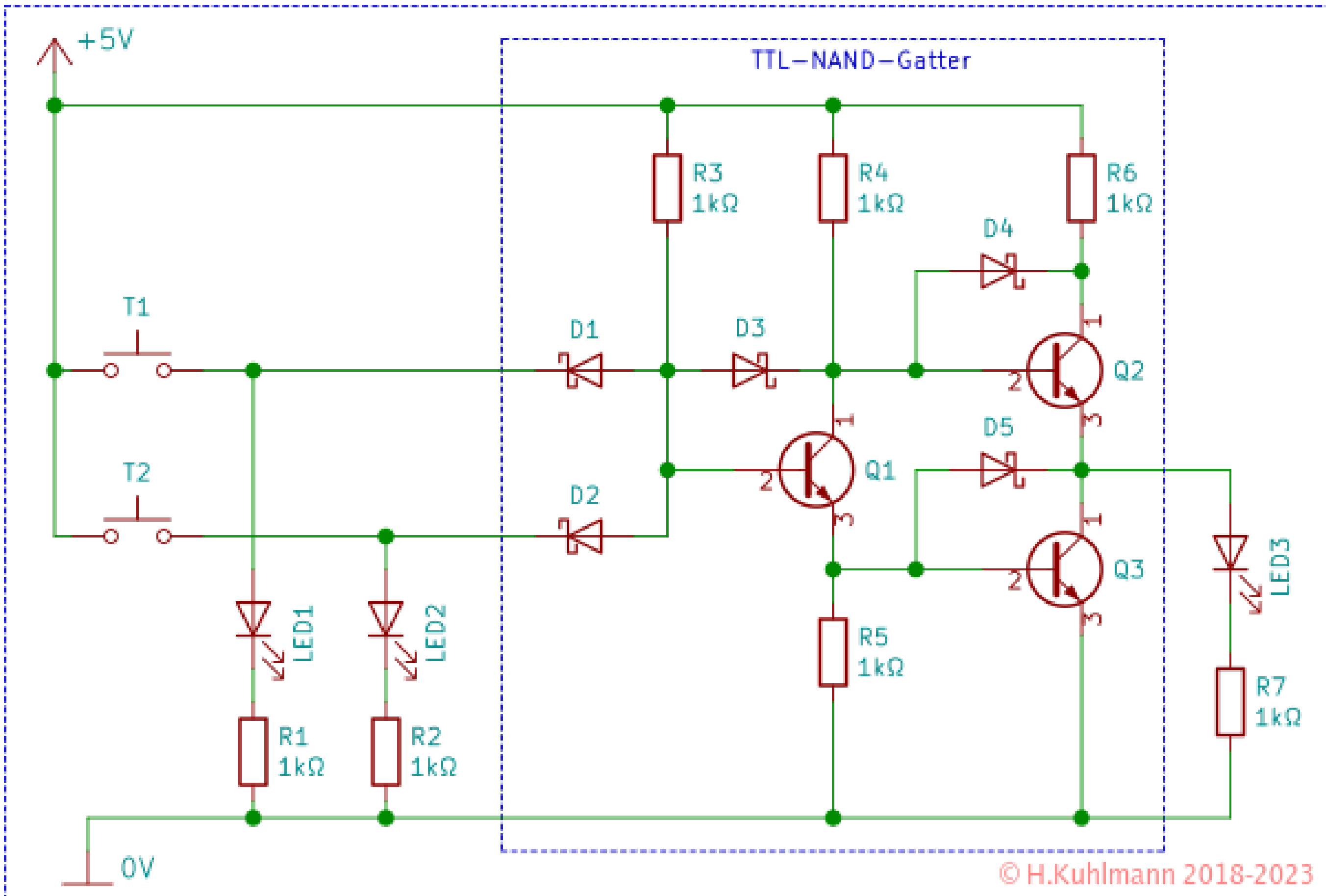
- Unsere Computer heute sind elektrotechnische Maschinen (Stand der Technik)
  - Aus Gründen der **Robustheit** und **Geschwindigkeit** nutzen wir **Digitaltechnik**
    - Entscheidend ist, **ob Strom fließt**, **nicht wie viel Strom** fließt
    - Folge: **nur zwei Werte** unterscheidbar: **an/aus**
    - Material: Kupferleitungen, + andere Metalle,
      Widerstände, Kondensatoren, Dioden, Transistoren,...
- **In Zukunft vielleicht:**
  - Optische Verfahren, Licht/Quanten, Kristalle, …
    - Eventuell mehr als zwei Werte schnell und robust nutzbar?
  - Biochemische (organische?) Verfahren mit schnellen Schaltzeiten?
- Vorerst jedoch **Digitaltechnik auf Basis elektrotechnischer Geräte**

Elektrotechnische Grundlage

Bauelemente: Leitungen, Widerstände, Kondensatoren, Dioden, Transistoren, …

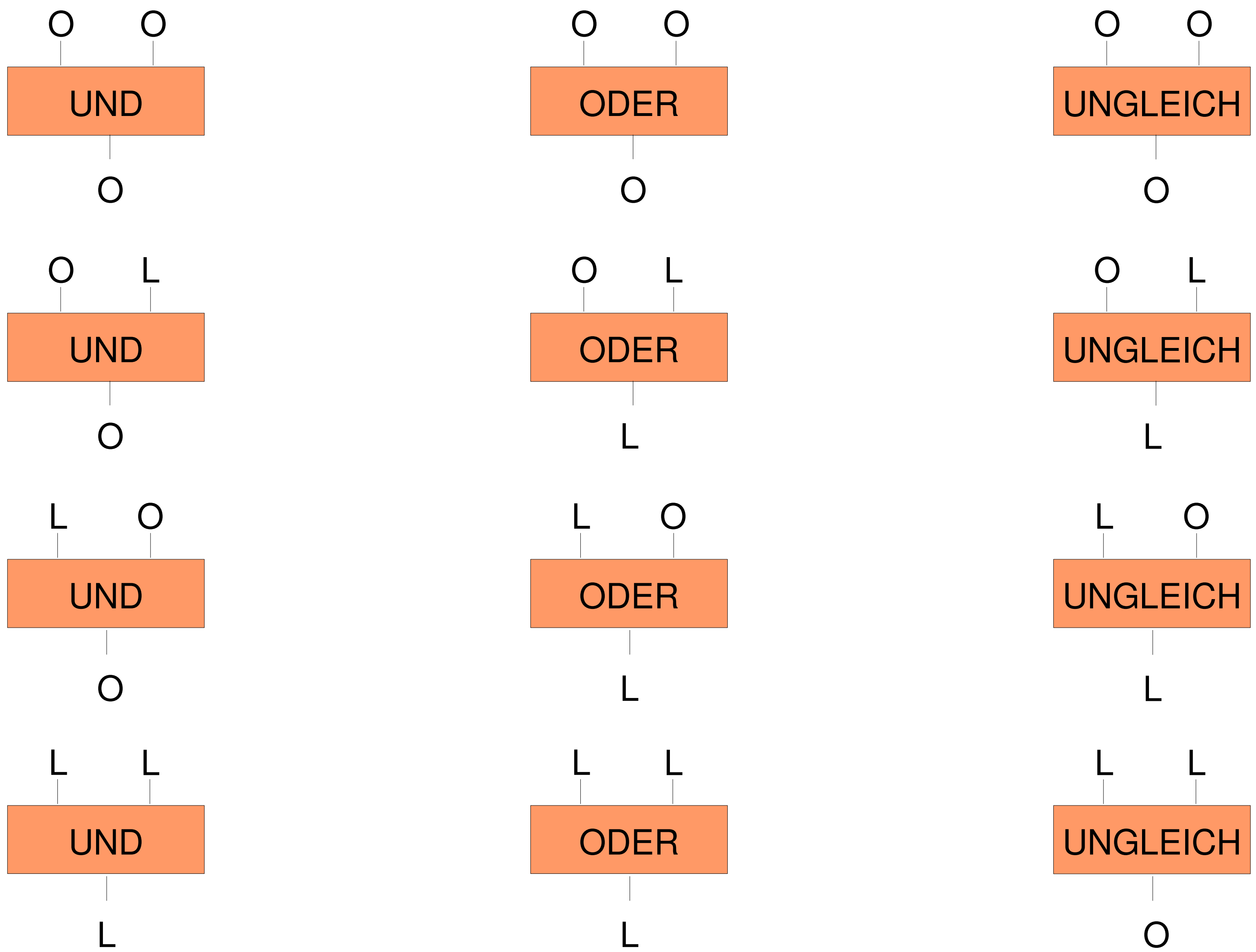
Beispiel: elektrotechnische Realisierung eines NAND-Gatters (nicht UND)

Merke: alle booleschen Funktionen (UND, ODER, XOR, NOT,…) können unter alleiniger Verwendung von einem oder mehreren NAND-Gattern realisiert werden!



Quelle: https://praktische-elektronik.dr-k.de/Praktikum/Analog/DiodenTransistoren/Le-Gatter-mit-Transistoren-und-Dioden.html

Logische Schaltglieder : UND, ODER, UNGLEICH (XOR)



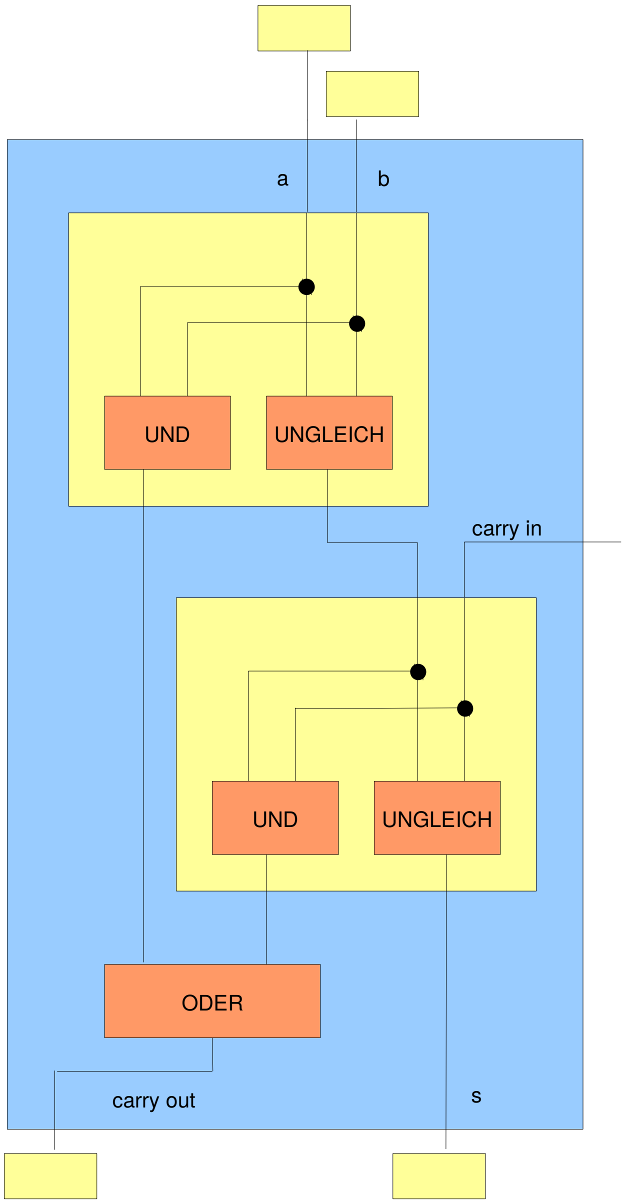
Welche Überträge können bei Addition zweier Zahlen im 10er System entstehen?

```
Addition im 10er-System
mit Ziffern 0,1,2,3,4,5,6,7,8,9

Beispiel 1:    4
               3
 Carry Out         Carry In (Übertrags Eingang)
        0 <--  0   <-- 0
           + -----
               7


Beispiel 2:    6
               3
 Carry Out         Carry In
        1 <--  1   <-- 1
           + -----
               0

Beispiel 3:    9
               9
 Carry Out         Carry In
        1 <--  1   <-- 1
           + -----
               9
```

Welche Überträge können bei Addition zweier Zahlen im 2er System entstehen?

```
Addition im 2er-System
mit Ziffern O für 0 und L für 1

Beispiel 1:    O
               L
 Carry Out         Carry In (Übertrags Eingang)
        O <--  O   <-- O
           + -----
               L


Beispiel 2:    O
               L
 Carry Out         Carry In
        L <--  L   <-- L
           + -----
               O

Beispiel 3:    L
               L
 Carry Out         Carry In
        L <--  L   <-- L
           + -----
               L
```

Realisierung eines 1Bit-Volladdierers durch logische Gatter UND,ODER,UNGLEICH

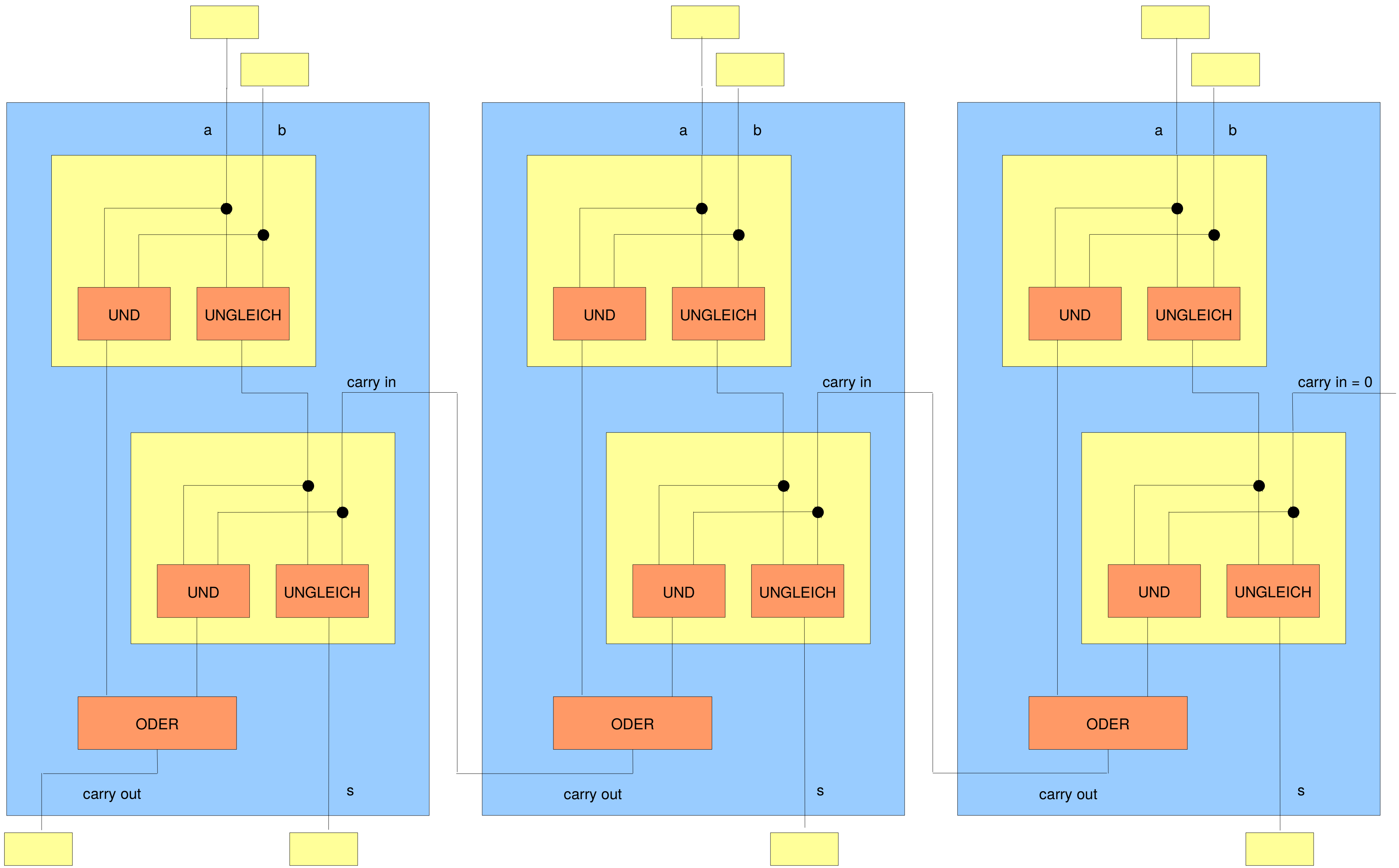



```
Addition im Dualsystem
mit Ziffern O für 0 und L für 1

Beispiel 1:    O
               L
 Carry Out           Carry In (Übertrags Eingang)
        O <--  O     <-- O
           + -----
               L


Beispiel 2:    O
               L
 Carry Out           Carry In
        L <--  L     <-- L
           + -----
               O

Beispiel 3:    L
               L
 Carry Out           Carry In
        L <--  L     <-- L
           + -----
               L
```

Wir sehen: Rechnen mit Übertrag im Dualsystem = Kombination logischer Gatter

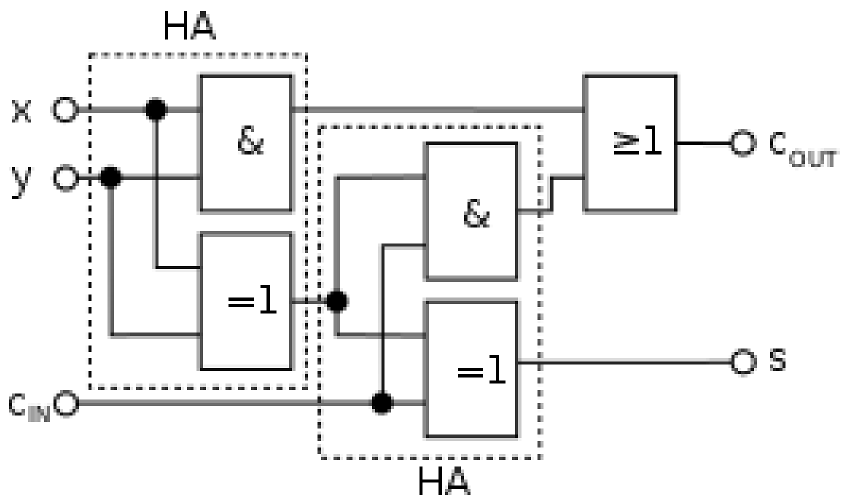
3-Bit-Addierer mit Übertragswelle (Carry Ripple)

(vergleiche: https://de.wikipedia.org/wiki/Volladdierer)



Elektronische Schaltung für Volladdierer

Beispiel: Volladdierer aufgebaut aus zwei Halbaddierern



Legende (IEC 60617-12):

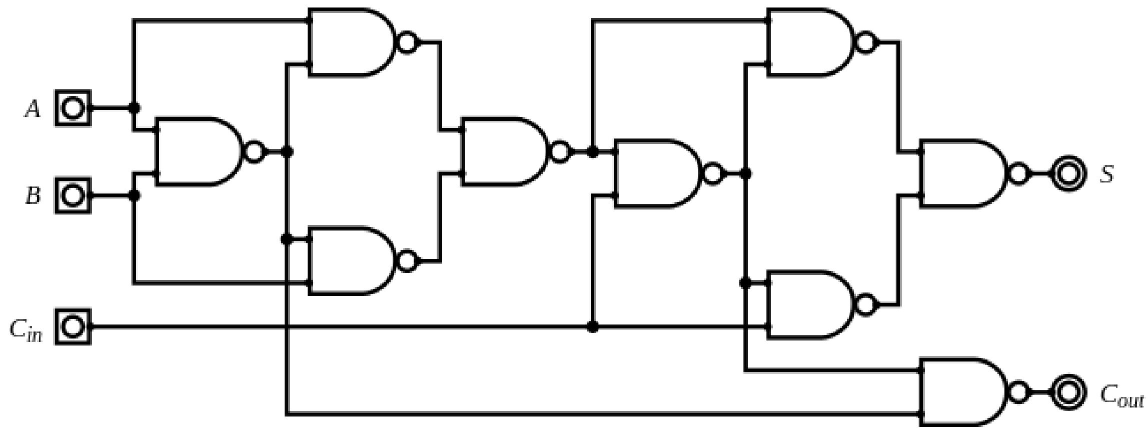
&: UND (AND)

≧1: ODER (OR)

=1: UNGLEICH (XOR)

Beispiel: Volladdierer aufgebaut aus NAND-Gattern (NAND-Symbol nach US ANSI 91-1984)

Merke: alle booleschen Funktionen (AND, OR, XOR, NOT,…) können unter alleiniger Verwendung von einem oder mehreren NAND-Gattern realisiert werden!

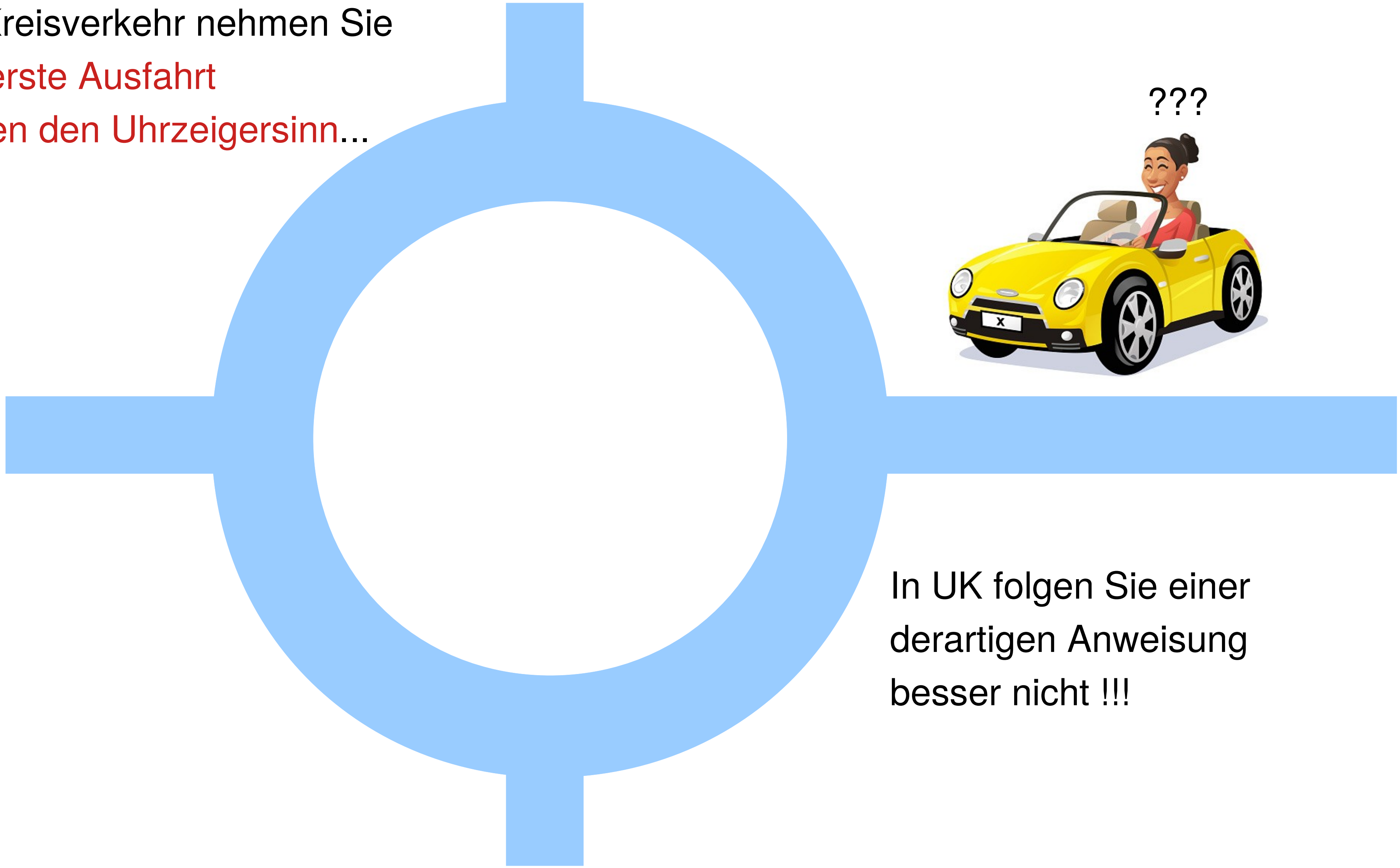


Und was ist der Vorteil der Verwendung von NAND-Gattern?

Quelle: https://de.wikipedia.org/wiki/Volladdierer

Navigation durch Kreisverkehr in UK

Im Kreisverkehr nehmen Sie
die erste Ausfahrt
gegen den Uhrzeigersinn...



In UK folgen Sie einer
derartigen Anweisung
besser nicht !!!

Quelle für Auto: https://pixabay.com/de/illustrations/wagen-cartoon-auto-4065110/

Drive smart!

Im Kreisverkehr nehmen Sie die erste Ausfahrt gegen den Uhrzeigersinn...

We are smart!

erste Ausfahrt gegen den Uhrzeigersinn

entspricht

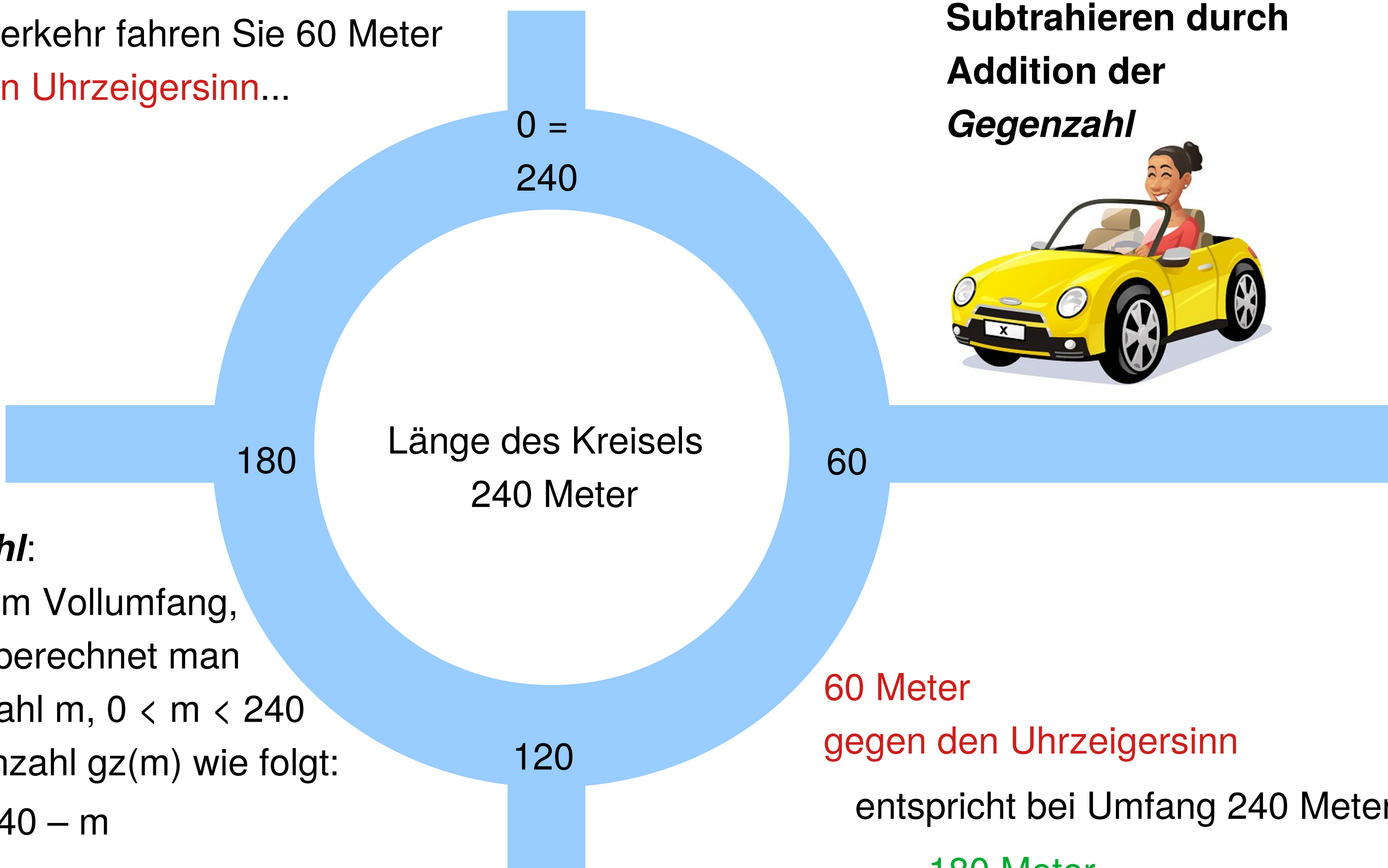
dritte Ausfahrt Im Uhrzeigersinn

Quelle für Auto: https://pixabay.com/de/illustrations/wagen-cartoon-auto-4065110/

Und wie kann ein Addierwerk subtrahieren?

Im Kreisverkehr fahren Sie 60 Meter
gegen den Uhrzeigersinn...

Subtrahieren durch Addition der *Gegenzahl*



Gegenzahl:
Relativ zum Vollumfang, hier 240, berechnet man für eine Zahl m, $0 < m < 240$ die Gegenzahl gz(m) wie folgt:

$gz(m) = 240 - m$

$gz(60) = 240 - 60 = 180$

60 Meter
gegen den Uhrzeigersinn

entspricht bei Umfang 240 Meter

180 Meter
im Uhrzeigersinn

Quelle für Auto: https://pixabay.com/de/illustrations/wagen-cartoon-auto-4065110/

Grundsätzliche Entscheidung: Verwendung endlicher Zahlbereiche

* Aus den vorherigen Beispielen erahnen wir, dass man die Subtraktion (nach links auf dem Zahlenkreis) durch Addition (nach rechts auf dem Zahlenkreis) irgendwie simulieren kann

* Allerdings hat dies einen gewissen **Preis**!
  Wir müssen uns auf einen endlichen Zahlenbereich (Intervall) beschränken

  Im Beispiel mit dem Kreisel: Zahlen zwischen 0 und 239

* In der Mathematik bildet man die vorhin angedeuteten Zahlenkreise durch die Bildung von Restklassen bezügliche eines Moduls n (Grenzzahl).
  Im Beispiel des Kreisels war der Modul die Zahl 240

* Die in beiden Richtungen (positiv und negativ) unendlich ausgedehnte Menge der ganzen Zahlen wird durch **Teilen mit Rest** in sogenannte **Restklassen** abgebildet

  Man teilt ganze Zahlen **m** ganzzahlig durch den Modul **n** und betrachtet nur den verbleibenden Rest **r**.
  Alle Zahlen m, die beim Teilen durch n den gleichen Rest r lassen, werden in den gleichen Topf geworfen, die sogenannte Restklasse [r]

  Durch dieses Verfahren bilden sich für einen positiven Modul n genau n verschiedene Restklassen: [0], [1], [2], …, [n-1]

* Zusammen mit den Rechenoperationen + und * entsteht die algebraische Struktur eines **Rings mit 1**, der **Restklassenring modulo n**

Repräsentanten im Restklassenring modulo n mit natürlicher Zahl n

Im **Restklassenring modulo n**, wobei n eine natürliche Zahl ist,

* gibt es n **Äquivalenzklassen**, die sogenannten **Restklassen modulo n**

* die **Standard-Repräsentanten** sind **{0,1,2, …, n-1}**

* **Beispiel n = 16**: Restklassenring modulo 16

  Die Standard-Repräsentanten im Restklassenring modulo 16 sind:

  {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15}

  Die 16 Restklassen bezeichnet man mit [r], wobei r einer der Standard-Repräsentanten ist, mit 0 <= r < 16

  [0], [1], …, [15]

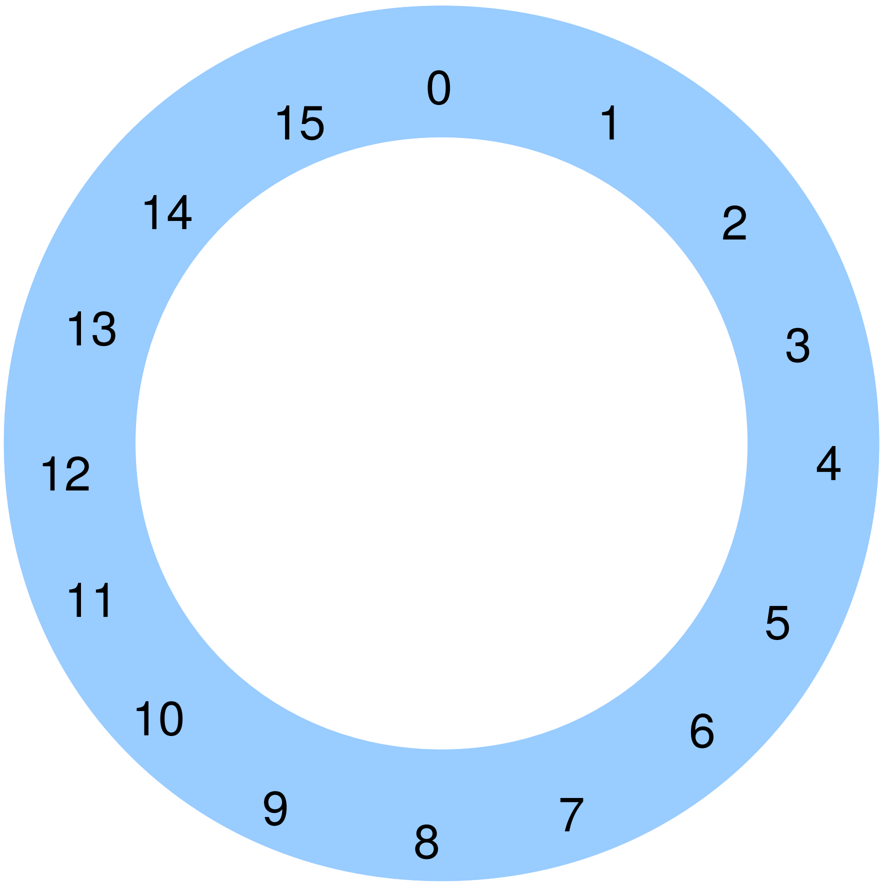
  Eine ganze Zahl m ist in der Restklasse [r] genau dann, wenn es eine ganze Zahl q gibt, so dass gilt:

  m = q * 16 + r   wobei 0 <= r < 16

  Also [3] = {… , -61, -45, -29, -13, 3, 19, 35, 51, 67, … }

  Damit gilt für beliebige m1, m2 aus [r]: (m1 - m2) mod 16 = 0

Repräsentanten im Restklassenring modulo 16




```
[ 0] = {... , -48,-32,-16, 0,16,32,48, ...}
[ 1] = {... , -47,-31,-15, 1,17,33,49, ...}
[ 2] = {... , -46,-30,-14, 2,18,34,50, ...}
[ 3] = {... , -45,-29,-13, 3,19,35,51, ...}
[ 4] = {... , -44,-28,-12, 4,20,36,52, ...}
[ 5] = {... , -43,-27,-11, 5,21,37,53, ...}
[ 6] = {... , -42,-26,-10, 6,22,38,54, ...}
[ 7] = {... , -41,-25, -9, 7,23,39,55, ...}
[ 8] = {... , -40,-24, -8, 8,24,40,56, ...}
[ 9] = {... , -39,-23, -7, 9,25,41,57, ...}
[10] = {... , -38,-22, -6,10,26,42,58, ...}
[11] = {... , -37,-21, -5,11,27,43,59, ...}
[12] = {... , -36,-20, -4,12,28,44,60, ...}
[13] = {... , -35,-19, -3,13,29,45,61, ...}
[14] = {... , -34,-18, -2,14,30,46,62, ...}
[15] = {... , -33,-17, -1,15,31,47,63, ...}
```

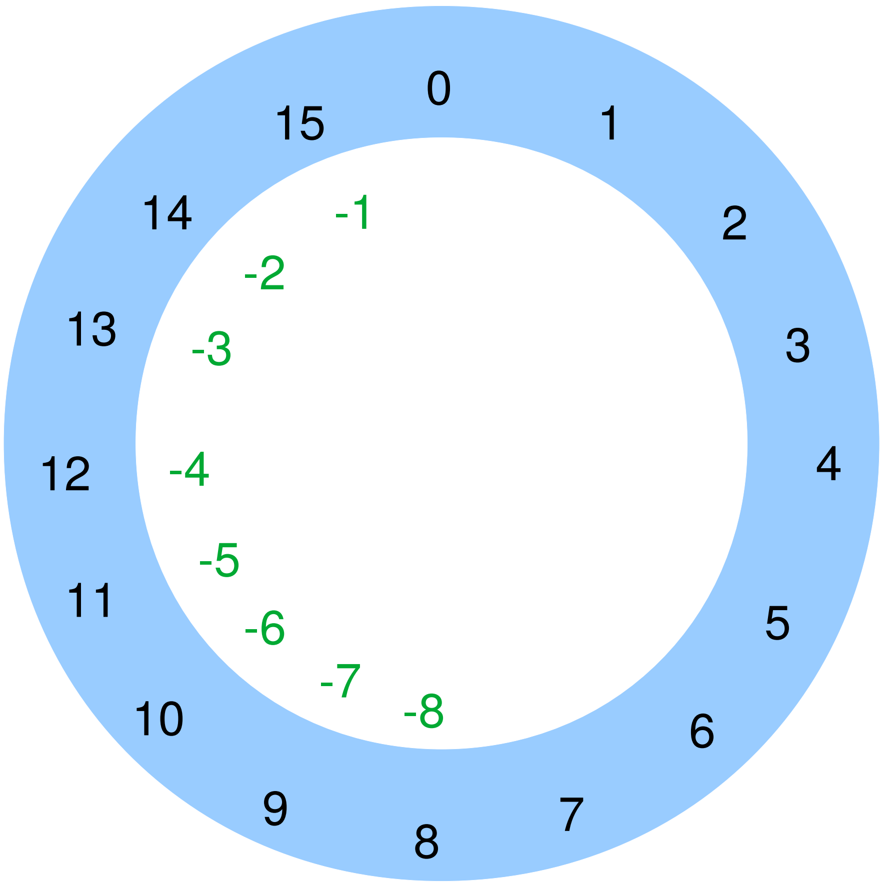
Für den Modul 16 gilt:

Eine ganze Zahl m ist in der Restklasse [r] genau dann, wenn es eine ganze Zahl q gibt, so dass gilt:

```
m = q * 16 + r     wobei 0 <= r < 16
```

Beispiel: -22 ist in Klasse [10], denn -22 = (-2)*16 + 10

Wahl des darstellbaren (kodierbaren) Zahlbereichs




```
[ 0] = {... , -48,-32,-16, 0,16,32,48, ...}
[ 1] = {... , -47,-31,-15, 1,17,33,49, ...}
[ 2] = {... , -46,-30,-14, 2,18,34,50, ...}
[ 3] = {... , -45,-29,-13, 3,19,35,51, ...}
[ 4] = {... , -44,-28,-12, 4,20,36,52, ...}
[ 5] = {... , -43,-27,-11, 5,21,37,53, ...}
[ 6] = {... , -42,-26,-10, 6,22,38,54, ...}
[ 7] = {... , -41,-25, -9, 7,23,39,55, ...}
[ 8] = {... , -40,-24, -8, 8,24,40,56, ...}
[ 9] = {... , -39,-23, -7, 9,25,41,57, ...}
[10] = {... , -38,-22, -6,10,26,42,58, ...}
[11] = {... , -37,-21, -5,11,27,43,59, ...}
[12] = {... , -36,-20, -4,12,28,44,60, ...}
[13] = {... , -35,-19, -3,13,29,45,61, ...}
[14] = {... , -34,-18, -2,14,30,46,62, ...}
[15] = {... , -33,-17, -1,15,31,47,63, ...}
```

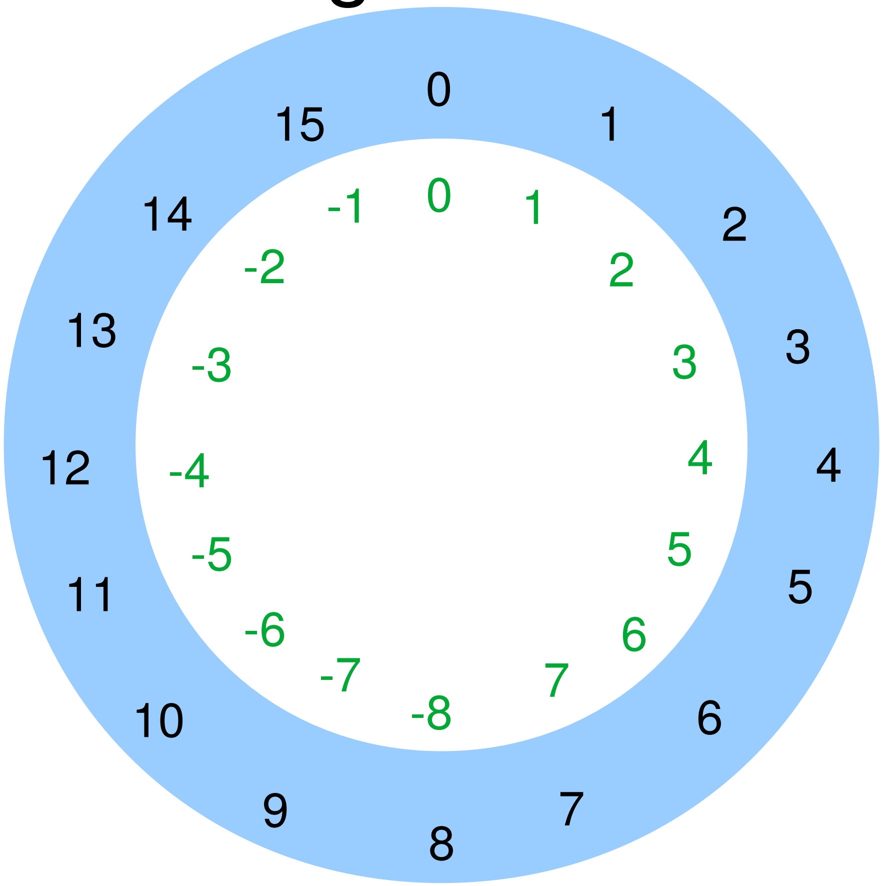
Für den Modul 16 gilt: die Standardrepräsentanten sind {0,1, … 14,15}

Um die Eindeutigkeit unserer Rechenergebnisse zu gewährleisten, müssen wir uns nun entscheiden, welchen Zahlbereich wird mit diesen Repräsentanten darstellen möchten.

Für Addition und Subtraktion sollen Argumente und Resultat darstellbar sein. Aufgrund praktischer Erwägungen entscheiden wir uns daher für den Zahlbereich

[-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]

Kodierung des Zahlbereichs im Dualsystem (4 Bit)



Binär	Repr.	Int-Zahl
OOOO	0	0
OOOL	1	1
OOLO	2	2
OOLL	3	3
OLOO	4	4
OLOL	5	5
OLLO	6	6
OLLL	7	7
LOOO	8	-8
LOOL	9	-7
LOLO	10	-6
LOLL	11	-5
LLOO	12	-4
LLOL	13	-3
LLLO	14	-2
LLLL	15	-1

Für den Modul 16 gilt: die Standardrepräsentanten sind [0,1, … 14,15]

Gewählter Zahlbereich: [-8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7]

Kodierung der Standardrepräsentanten durch Binärzahlen O,L mit 4 Stellen

Beobachtung: die Repräsentanten 8 .. 15 der negativen Integer-Zahlen -8 .. -1 haben alle ein L als höchstwertiges Bit (MSB). Damit kann das MSB als Vorzeichen interpretiert werden, was es aber orginär nicht ist.

Bezug zur C-Programmierung

* In der Programmiersprache C gibt es unterschiedliche Zahlbereiche für
  ganze Zahlen (Integer) mit Vorzeichen (sign)

Beispiel: Intel 80686 unter Linux (VM 32bit):

* Typ **char** mit **8 Bit**: [-128, … , 0, … 127]   (siehe Bemerkung ganz unten)

* Typ **short int** mit **16 Bit**: [-32768, … , 0, … 32767]

* Typ **int** mit **32 Bit**: [-2147483648, … , 0, … 2147483647]

* Typ **long long int** mit **64 Bit**: [-9223372036854775808, … , 0, … 9223372036854775807]

* **allgemein**:
  bei Kodierung mit **n Bit**: **[-2^(n-1), … , 0, … , 2^(n-1)-1]**

Bemerkung: Der C-Standard legt nicht genau fest, ob der Typ **char**
ein **signed char** oder ein **unsigned char** ist

Intel/AMD: signed char
Raspberry PI (arm 7): unsigned char

Die anderen Typen **short int**, **int**, **long int**, **long long int**
sind implizit immer **signed**

Gegenzahl im Dualsystem: Zweierkomplement

Im **Restklassenring modulo n** berechnet man die **Gegenzahl g(m)** der Zahl m wie folgt:

```
gz(0) = 0        für m = 0
gz(m) = n - m    für 0 < m < n
```

Bei Codierung im Dualsystem kann man die Berechnung von **g(m)** auf das Inverse **inv(m)** der Zahl m zurückführen, welches durch einfaches Invertieren der Bits entsteht

Invertieren der Bits **inv(m)** nennt man **Einerkomplement** (one's complement)
Die **Gegenzahl gz(m)** bilden nennt man **Zweierkomplement** (two's complement)

Beispiel bei Codierung mit 4 Bits: n = 2^4 = 16 entspricht L0000

Codes: [0000, 000L, … , LLL0, LLLL] für die Zahlen [0,1, …, 14,15]

```
 code(m):  0L0L
  inv(m):  L0L0
         + ----
   Summe:  LLLL
```

Beobachtungen:

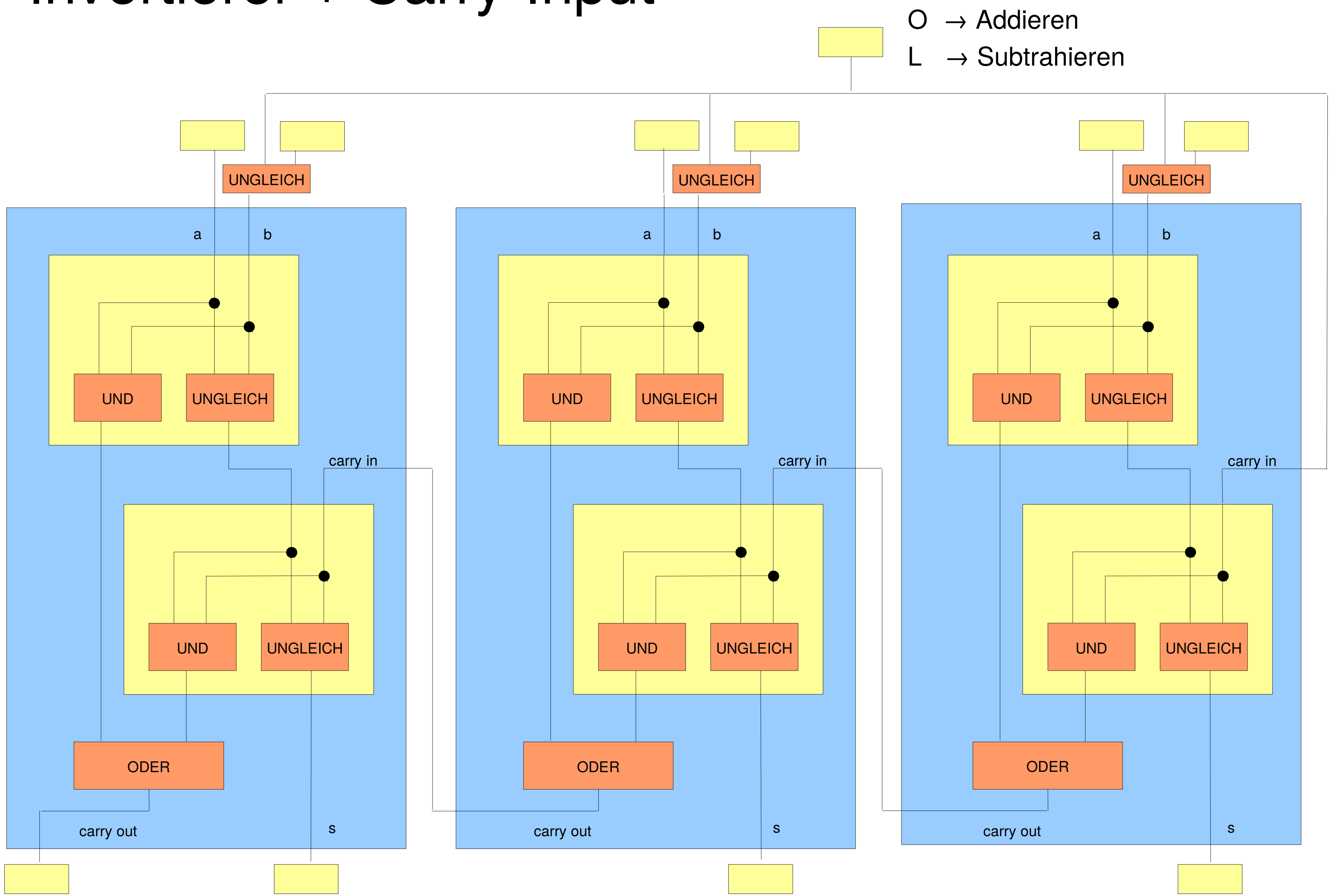
1) code(m) und inv(m) ergänzen sich zu LLLL für alle 0 < m < 16

   Damit: inv(m) = LLLL - code(m)

2) L0000 = LLLL + L

```
Wegen 1) inv(m) = LLLL - code(m)
Und   2)  L0000 = LLLL + L

gilt:

gz(m) = L0000 - code(m)
      = (LLLL + L) - code(m)
      = (LLLL - code(m)) + L
      = inv(m) + L
```

Subtraktion mit implizitem Zweierkomplement durch Invertierer + Carry-Input



Spielwiese: CarryRippleDemo.hs

- Zunächst einmalig die Entwicklungsumgebung für Haskell installieren

```
sudo apt-get update
sudo apt-get install ghc ghc-doc cabal-install
sudo apt-get install ghc-prof llvm-13 gmp-doc libgmp10-doc libmpfr-dev
```

- Die neueste Version der Code-Beispiele via git abholen

```
cd ~/git_public_GdP1/CodeExamples

git pull
```

- Dann: Aufruf der Simulationsumgebung für den Carry-Ripple Addierer

```
cd ~/git_public_GdP1/CodeExamples/Haskell/TwosComplement
ghci CarryRippleDemo.hs

*CarryRippleDemo>
```

- Der Simulator unterstützt Binärzahlen mit einer beliebigen Anzahl von Bits

```
a = mkFromInteger 5000 ( 3)
b = mkFromInteger 5000 (-2)
a `add` b == mkFromInteger 5000 1

Ausgabe: True
```

CarryRippleDemo: Erzeugung von Binärzahlen

In der laufenden Session der Carry-Ripple Demo:

*CarryRippleDemo>

Wir können beliebig lange Binärwörter einer bestimmten Länge erzeugen.

Variante 1: aus einer Zeichenkette per Factory-Methode **fromBitList**

Beispiel: Erzeugung einer 4-Bit Binärkodierung aus Zeichenkette "OOOL"

*CarryRippleDemo> **fromBitList 4 "OOOL"**

Ausgabe: fromBitList 4 "OOOL"

Die interne Kodierung wird absichtlich nicht angezeigt!

Die Ausgabe bedeutet:
Es wurde diejenige Binärkodierung erzeugt, die auch per Factory-Methode **fromBitList** mittels Aufruf **fromBitList 4 "OOOL"** erzeugt wird

Beispiel: Erzeugung einer 32-Bit Binärkodierung.
Die Punkte werden ignoriert und dienen der besseren Lesbarkeit.
Bei der Ausgabe wird alle 8 Bit ein Punkt eingestreut.

*CarryRippleDemo> **fromBitList 32 "LLLL.LLLL.LLLL.LLLL.LLLL.LLLL.LLLL.LLOL"**

Ausgabe: fromBitList 32 "LLLLLLLL.LLLLLLLL.LLLLLLLL.LLLLLLOL"

CarryRippleDemo: Erzeugung von Binärzahlen

In der laufenden Session der Carry-Ripple Demo:

```
*CarryRippleDemo>

Variante 2: aus einer Zahl per Factory-Methode mkFromInteger

Beispiel: Erzeugung einer 4-Bit Binärkodierung aus der Zahl 1

*CarryRippleDemo> mkFromInteger 4 (1)

Ausgabe:  fromBitList 4 "OOOL"

Die Ausgabe bedeutet:
Es wurde diejenige Binärkodierung erzeugt, die auch per Factory-Methode
fromBitList mittels Aufruf fromBitList 4 "OOOL" erzeugt wird

Beispiel: Erzeugung einer 32-Bit Binärkodierung aus der Zahl (-3)
          Bei der Ausgabe wird alle 8 Bit ein Punkt eingestreut.

*CarryRippleDemo> mkFromInteger 32 (-3)

Ausgabe:  fromBitList 32 "LLLLLLLL.LLLLLLLL.LLLLLLLL.LLLLLLOL"


Hinweis: negative Zahlen müssen zur Eingabe in ( ) eingeschlossen werden
```

CarryRippleDemo: Einerkomplement

In der laufenden Session der Carry-Ripple Demo:

```
*CarryRippleDemo>
```

Einerkomplement (one‘s complement): Funktion **one's**

```
Beispiel: Erzeugung einer 32-Bit Binärkodierung aus der Zahl (-310010253)
          und dann Erzeugung des Einerkomplements (Invertierung)
```

```
*CarryRippleDemo> mkFromInteger 32 (-310010253)
```

```
Ausgabe: fromBitList 32 "LLLOLLOL.LOOOOLOL.LOOLLLLO.OLLLOOLL"
```

```
*CarryRippleDemo> one's (mkFromInteger 32 (-310010253))
```

```
Ausgabe: fromBitList 32 "OOOLOOLO.OLLLLOLO.OLLOOOOL.LOOOLLOO"
```

Man kann Terme auch in Variablen speichern

```
*CarryRippleDemo> a = mkFromInteger 32 (-310010253)

*CarryRippleDemo> one's a

Ausgabe: fromBitList 32 "OOOLOOLO.OLLLLOLO.OLLOOOOL.LOOOLLOO"
```

Funktion one's ist idempotent: `one's (one's a) == a`

CarryRippleDemo: Zweierkomplement

In der laufenden Session der Carry-Ripple Demo:

`*CarryRippleDemo>`

Zweierkomplement (two‘s complement): Funktion **two's**

Beispiel: Erzeugung einer 32-Bit Binärkodierung aus der Zahl (-310010253)
und dann Erzeugung des Zweierkomplements (Gegenzahl)

`*CarryRippleDemo>` **`mkFromInteger 32 (-310010253)`**

Ausgabe: `fromBitList 32 "LLLOLLOL.LOOOOLOL.LOOLLLLO.OLLLOOLL"`

`*CarryRippleDemo>` **`two's (mkFromInteger 32 (-310010253))`**

Ausgabe: `fromBitList 32 "OOOLOOLO.OLLLLOLO.OLLOOOOL.LOOOLLOL"`

Man kann Terme auch in Variablen speichern

`*CarryRippleDemo> a = mkFromInteger 32 (-310010253)`

`*CarryRippleDemo> two's a`

Ausgabe: `fromBitList 32 "OOOLOOLO.OLLLLOLO.OLLOOOOL.LOOOLLOL"`

Funktion two's ist idempotent: `two's (two's a) == a`

CarryRippleDemo: Einer- und Zweierkomplement

In der laufenden Session der Carry-Ripple Demo:

```
*CarryRippleDemo>
```

Berechne das Zweierkomplement direkt

```
*CarryRippleDemo> a = mkFromInteger 32 (-310010253)

*CarryRippleDemo> two's a
```

Ausgabe: `fromBitList 32 "OOOLOOLO.OLLLLOLO.OLLOOOOL.LOOOLLOL"`

Berechne das Zweierkomplement über das Einerkomplement

```
*CarryRippleDemo> (one's a) `add` (mkFromInteger 32 1)
```

Ausgabe: `fromBitList 32 "OOOLOOLO.OLLLLOLO.OLLOOOOL.LOOOLLOL"`

Wir fragen das System: `two's a == (one's a) `add` (mkFromInteger 32 1)`

Ausgabe: `True`

CarryRippleDemo: Addieren

In der laufenden Session der Carry-Ripple Demo:

*CarryRippleDemo>

Addieren:

*CarryRippleDemo> mkFromInteger 8 (-23) `add` mkFromInteger 8 (26)

Ausgabe: fromBitList 8 "OOOOOOLL"

*CarryRippleDemo> mkFromInteger 8 (-23) `add` mkFromInteger 8 (-26)

Ausgabe: fromBitList 8 "LLOOLLLL"

Binärkodierung zurück in Integer verwandeln: Funktion **asInteger**

*CarryRippleDemo> asInteger (fromBitList 8 "LLOOLLLL")

Ausgabe: -49

Präfix-Schreibweise und Infix-Schreibweise für Funktionsnamen

Präfix-Schreibweise: add (mkFromInteger 8 (-23)) (mkFromInteger 8 (-26))

Infix-Schreibweise: mkFromInteger 8 (-23) `add` mkFromInteger 8 (-26)

CarryRippleDemo: Subtrahieren

In der laufenden Session der Carry-Ripple Demo:

```
*CarryRippleDemo>
```

Subtrahieren:

```
*CarryRippleDemo> mkFromInteger 8 (-23) `sub` mkFromInteger 8 (26)
```

```
Ausgabe: fromBitList 8 "LLOOLLLL"
```

```
*CarryRippleDemo> mkFromInteger 8 (-23) `sub` mkFromInteger 8 (-26)
```

```
Ausgabe: fromBitList 8 "OOOOOOLL"
```

Binärkodierung zurück in Integer verwandeln: Funktion **asInteger**

```
*CarryRippleDemo> asInteger (fromBitList 8 "LLOOLLLL")
```

```
Ausgabe: -49
```

Präfix-Schreibweise und Infix-Schreibweise für Funktionsnamen

```
Präfix-Schreibweise: sub (mkFromInteger 8 (-23)) (mkFromInteger 8 (-26))
 Infix-Schreibweise: mkFromInteger 8 (-23) `sub` mkFromInteger 8 (-26)
```

CarryRippleDemo: Addieren mit Demo-Ausgabe

In der laufenden Session der Carry-Ripple Demo:

```
*CarryRippleDemo>

Addieren mit Demo-Ausgabe:

*CarryRippleDemo> mkFromInteger 8 (-23) `add_demo` mkFromInteger 8 (26)

Ausgabe:

Addition explained:
  Integer numbers without parenthesis show the signed integer encoding.
  Integer numbers within  parenthesis show the original integer value.

    LLLOLOOL   233  (-23)
    OOOLLOLO    26  ( 26)
(+) --------
    OOOOOOLL     3  (  3)

   Flags:
     Carry out (ignore)
```

CarryRippleDemo: Subtrahieren mit Demo-Ausgabe

In der laufenden Session der Carry-Ripple Demo:

```
*CarryRippleDemo>
```

Subtrahieren mit Demo-Ausgabe:

```
*CarryRippleDemo> mkFromInteger 8 (-23) `sub_demo` mkFromInteger 8 (26)
```

Ausgabe:

```
Subtraction explained:
  Integer numbers without parenthesis show the signed integer encoding.
  Integer numbers within  parenthesis show the original integer value.

    LLLOLOOL   the first  operand  : 233  (-23)
    OOOLLOLO   the second operand  :  26  ( 26)
    LLLOOLLO   its two's complement: 230  (-26)

    LLLOLOOL   233  (-23)
    LLLOOLLO   230  (-26)
(+) --------
    LLOOLLLL   207  (-49)

   Flags:
     Carry out (ignore)
```

CarryRippleDemo: Arithmetischer Überlauf

In der laufenden Session der Carry-Ripple Demo:

```
*CarryRippleDemo>
```

Addieren mit Demo-Ausgabe: Beispiel mit Überlauf

```
*CarryRippleDemo> mkFromInteger 8 (23) `add_demo` mkFromInteger 8 (126)

Ausgabe:

Addition explained:
  Integer numbers without parenthesis show the signed integer encoding.
  Integer numbers within  parenthesis show the original integer value.

    OOOLOLLL    23  (  23)
    OLLLLLLO   126  ( 126)
(+) --------
    LOOLOLOL   149  (-107)

   Flags:

     Arithmetic overflow!
```

Addieren mit Demo-Ausgabe: ein weiteres Beispiel mit Überlauf

```
*CarryRippleDemo> mkFromInteger 8 (-23) `add_demo` mkFromInteger 8 (-126)
```

CarryRippleDemo: Arithmetischer Überlauf

```
*CarryRippleDemo>
```

Subtrahieren mit Demo-Ausgabe: Beispiel mit Überlauf

```
*CarryRippleDemo> mkFromInteger 8 (-23) `sub_demo` mkFromInteger 8 (126)

Ausgabe:

Subtraction explained:
  Integer numbers without parenthesis show the signed integer encoding.
  Integer numbers within  parenthesis show the original integer value.

    LLLOLOOL   the first  operand  : 233  ( -23)
    OLLLLLLO   the second operand  : 126  ( 126)
    LOOOOOLO   its two's complement: 130  (-126)

    LLLOLOOL   233  ( -23)
    LOOOOOLO   130  (-126)
(+) --------
    OLLOLOLL   107  ( 107)

   Flags:
     Carry out (ignore)
     Arithmetic overflow!
```

Testen Sie ebenfalls: `mkFromInteger 8 (23) `sub_demo` mkFromInteger 8 (-126)`