

Codierung

- Alle Daten, die verarbeitet werden, müssen letztendlich als Bitfolge gespeichert werden (mehr kann der Rechner nicht!)
- Es steht nur endlich viel Speicherplatz zur Verfügung
- Wie lassen sich so unterschiedliche Arten von Daten (Ganzzahlen, Kommazahlen, Zeichen etc.) codieren?

Codierung

Codierung von Ganzzahlen

- Ganze Zahlen lassen sich direkt ins Zweiersystem umrechnen und so problemlos als Bitfolge speichern
 - Beispiel: $25 = 16 + 8 + 1 = (11001)_2$
- Problem dabei:
 - Was ist mit negativen ganzen Zahlen?

- Erster Ansatz:
 - Verwende ein zusätzliches Bit, um das Vorzeichen (Plus oder Minus) anzuzeigen, z. B. das Bit an vorderster Stelle (sogenanntes “Most Significant Bit”, MSB)
 - Beispiel (4 Bit pro Zahl):
 - $+3 = (0011)_2$
 - $-3 = (1011)_2$
- Nachteile:
 - “Direktes” Rechnen schwierig, z. B.
 $(-3) + (+1) = (1011)_2 + (0001)_2 = (1100)_2 = -4$
 - Zwei Darstellungen der Zahl 0
 $(0000)_2$ und $(1000)_2$

- In der Praxis rechnet ein Computer immer mit endlicher Stellenzahl (bspw. 32 Bit für Integer)
- Es gibt somit also nur endlich viele unterschiedliche Zahlen, die dargestellt werden können
- Wird eine Zahl zu groß, so tritt ein sogenannter **Überlauf** auf
 - Beispiel (3 Bit pro Speicherwort):
 $2 + 7 = (010)_2 + (111)_2 = (1001)_2$, im Speicher steht also nur $(001)_2 = 1$
- Der Computer rechnet also gar nicht in \mathbb{Z} , sondern in einem sogenannten **Restklassenring**

- Unter der **Restklasse** einer Zahl a modulo einer Zahl m versteht man die Menge aller Zahlen, die bei Division durch m den gleichen Rest haben wie die Division $a : m$
 - Kurzschreibweise: $[a]$
- Beispiel: Restklassen modulo 8
 - $[0] = \{\dots, -8, 0, 8, 16, 32, \dots\}$
 - $[1] = \{\dots, -7, 1, 9, 17, 33, \dots\}$
 - \dots
 - $[7] = \{\dots, -1, 7, 15, 23, 39, \dots\}$
- In diesem Restklassenring gilt also beispielsweise:
 - $3 + 7 \equiv 3 + 23 \equiv -5 + 7 \equiv 2 \pmod{8}$(Sprechweise: “kongruent modulo 8”)

- Betrachten wir Zahlen mit einer “Breite” von 3 Bit, so befinden wir uns im Restklassenring modulo 8 (mit 3 Bit lassen sich 8 verschiedene Bitfolgen bilden)
- Die Interpretation von $(000)_2$ als 0, $(001)_2$ als 1 usw. ist nur **eine** Möglichkeit
 - Denkbar wäre beispielsweise auch $(000)_2$ als 8, $(001)_2$ als 9 usw.
- Idee:
 - Negative Zahlen werden durch das kleinste positive Element aus derselben Restklasse codiert
 - Beispiel: $-3 \equiv 5 \pmod{8}$, also wird -3 codiert als $(101)_2$
 - Problem:
 - Wie codieren wir dann $+5$?

- Wie kommt man von einer negativen Zahl n (mit $|n| < 8$) zu ihrem kleinsten positiven Repräsentanten p ?
 - Es gilt $p = n + 8 = 8 - |n|$
 - Für $n = -2$ folgt z. B. $p = -2 + 8 = 8 - |-2| = 6$
- Binär:
 - Schreibe $|n|$ binär, im Beispiel also $(010)_2$
 - Invertiere alle Bits, im Beispiel also $(010)_2 \rightarrow (101)_2$
 - Dies entspricht der Rechnung $7 - |n|$
 - Addiere 1, im Beispiel also $(101)_2 + (1)_2 = (110)_2 = 6$
- Wir haben das **Zweierkomplement** einer Zahl gebildet

Noch was?

- Woher wissen wir, welche Bitfolgen für negative Zahlen stehen und welche für positive?
- Festlegung:
 - Alle Bitfolgen, die mit einer 0 beginnen, codieren positive Zahlen
 - Alle Bitfolgen, die mit einer 1 beginnen, codieren negative Zahlen

Bitfolge	Interpretation	Bitfolge	Interpretation
000	0	100	-4
001	1	101	-3
010	2	110	-2
011	3	111	-1

Aufgabe 1

- Erstelle eine Tabelle wie auf der letzten Folie, aber für Zahlen, die aus insgesamt 4 Bit bestehen
- Rechne schriftlich im Zweiersystem unter Verwendung der Codierungen aus Deiner Tabelle
 - a) $2 + 5$
 - b) $-2 + 5$
 - c) $-8 + 3$
 - d) $-8 + 7$
 - e) $5 + 7$

- Java codiert alle Ganzzahlen (**byte**, **short**, **int**, **long**) unter Verwendung des Zweierkomplements
 - Beispiel Datentyp **byte**: Breite 8 Bit, Wertbereich von -128 bis $+127$
- Bei C/C++ gibt es vorzeichenlose (**unsigned**) und vorzeichenbehaftete (**signed**) Datentypen

- Öffne das BlueJ-Projekt “Zweierkomplement”
 - Verwende die Methode `byteBinaer`, um verschiedene Zahlen in binärer Darstellung zu betrachten

Aufgabe 3

- Was gibt folgender Code aus? Teste mit BlueJ

```
for ( byte b = -128; b < 128; b++ ) {  
    System.out.println(b);  
}
```

- Beobachtung: Die Schleife läuft endlos. Warum?
 - Für den Wert 127 von **b** wird die Schleife das letzte Mal durchlaufen
 - Am Ende dieses Durchlaufs wird **b** nochmals inkrementiert
 - Binär betrachtet ändert sich **b** dadurch von **0111111** zu **10000000**. Dies ist die Zweierkomplementdarstellung von **-128**
 - Es ist ein sogenannter **Überlauf** aufgetreten

**judge: I sentence you to the
maximum punishment...**
Me. May I have one day more?
**judge: Ok, you have to serve
-32.768 years in prison.**



Abbildung 1: Integer-Überlauf in der Praxis ;-)

- Welche “Breite” (in Bit) hat der hier verwendete Datentyp?

- Selten wird auch noch die Codierung im **Einerkomplement** für Ganzzahlen verwendet
 - Die Bildung erfolgt einfach durch bitweises invertieren einer Zahl, z. B.
 $(010)_2 = 5_{10} \Rightarrow (1010)_2 = -5_{10}$
- Für 3 Bit “breite” Zahlen ergibt sich folgende Codierung

Bitfolge	Interpretation	Bitfolge	Interpretation
000	0	100	-3
001	1	101	-2
010	2	110	-1
011	3	111	-0

- Das Einerkomplement ist etwas einfacher zu bilden (nur Bits invertieren), aber:
 - Die Zahl 0 kommt doppelt vor ($+0$ und -0), dafür ist eine negative Zahl weniger darstellbar
 - Für eine Addition sind unter Umständen zwei Schritte nötig
 - Eigentliche Addition
 - Anschließende Addition des “übriggebliebenen” Übertrags
- In der Praxis wird daher meist das Zweierkomplement verwendet