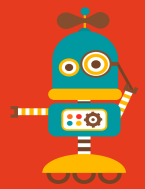
htl3 rennweg

Bitoperationen

DI Reinhold Buchinger

Höhere Abteilung für Mechatronik
Höhere Abteilung für Informationstechnologie
Fachschule für Informationstechnik



Lizenz/Credits

» Creative Commons-Lizenz CC BY-NC-SA 4.0 AT.

Ziele

- Einsatzgebiet von Bitoperationen verstehen
- Bitoperationen verstehen und in Java anwenden können

Inhalt

- Anwendungsgebiet
- Darstellung ganzer Zahlen in Java
- Bitoperationen

Was sind Bitoperationen

» Anstatt auf der Ebene von primitiven Datentypen oder Objekten zu arbeiten, arbeiten wir mit Bits.

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

Einsatzgebiet

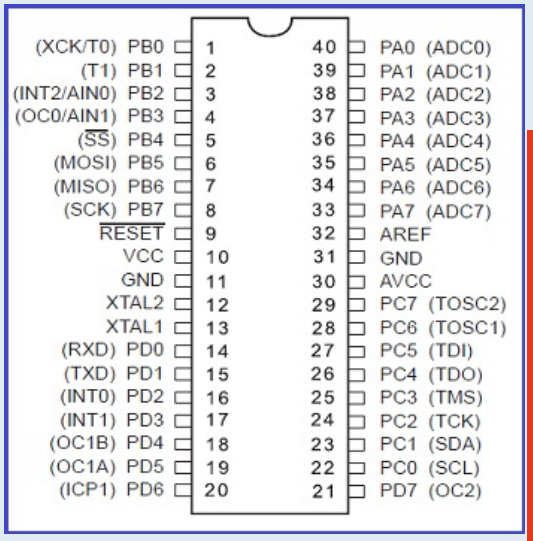
» In der hardwarenahen Programmierung wird oft mit einzelnen Bits gearbeitet.

» Aktivieren/Deaktivieren einer einzigen Datenleitung -> 1 Bit

» Fasst mehrere solcher Leitungen (Ports) in einem Datenwort zusammen

» "Flags" (einzelne Bits) in einer Variable

zu speichern anstatt einer großen Anzahl an Boolean kann manchmal sinnvoll sein.



Darstellung ganzer Zahlen

int und long in Java

» Ein **int** hat immer **32 bits** in Java.

0000 0000 0000 0000 0000 0000 0000 0000

» Ein **long** hat immer **64 bits** in Java.

0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000

Zweier Komplement – positive Zahlen

- Um negative und positve Zahlen einfach speichern zu können, werden die Zahlen als Zweierkomplement kodiert.
- Das Bit an der höchsten Stelle (ganz links) ist dabei 1 bei negativen, 0 bei positiven Zahlen.

0101 1110 0111 0000 0111 0000 0100 0110

Stelle mit der höchsten Wertigkeit

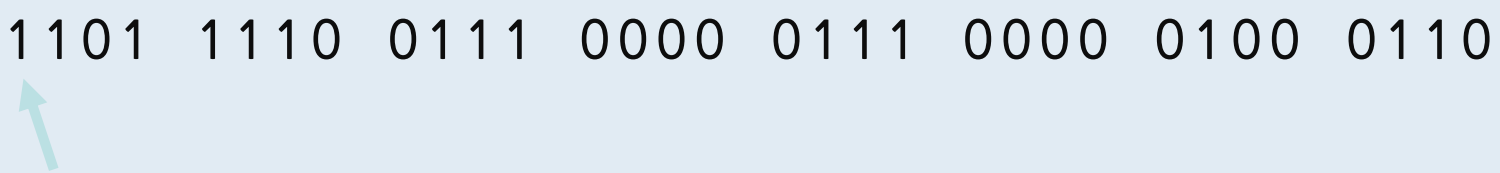
- Bei n Stellen ist die größte positive darstellbare Zahl daher

$$2^{n-1} - 1$$

  - n-1 Stellen für die Darstellung der Zahl, 1 Stelle für das Vorzeichen
  - -1 weil wir 0 auch darstellen müssen

Zweier Komplement – negative Zahlen

» Negative Zahlen haben an der höchsten Stelle eine 1.



» Bei n Stellen ist die kleinste (negative) darstellbare Zahl daher

$$-2^{n-1}$$

Zweier Komplement

» Um von einer positiven Zahl die Darstellung der negativen Zahl zu erhalten werden...
  » Sämtliche binären Stellen negiert
  » Zu dem Ergebnis der Wert *1* addiert

0000 0000 0000 0000 0000 0000 0000 0100	4 als Binärzahl
1111 1111 1111 1111 1111 1111 1111 1011	Intervieren
1111 1111 1111 1111 1111 1111 1111 1100	Eins addieren

Von Binär zu Hexadezimal

1111	0000	1100	1001	1111	1111	1111	0110
F	0	C	9	F	F	F	6

0xF0C9FFF6

Binäroperationen

& Bitweises Und

» Bitweises und: **&** → Beide Bits müssen 1 sein, damit das Ergebnis 1 ist.

```
   1100  0100
&  0010  0101
=  0000  0100
```

| Bitweises Oder

» Bitweises oder: | → Mindestens ein Bit muss 1 sein.

```
   1100  0100
|  0010  0101
=  1110  0101
```

^ Bitweises Exklusiv oder

» Bitweises exklusiv oder (XOR): ^ → Genau ein Bit darf 1 sein

```
   1100  0100
^  0010  0101
=  1110  0001
```

~ Bitweises Invertieren

» Bitweises invertieren: ~ → Jedes Bit wird umgedreht

```
~   1100 0100
=   0011 1011
```

Binäroperationen
Shift Operationen

<< nach links schieben

» nach Links schieben: << → Schiebt die Bits um beliebig viele Stellen nach links (von rechts kommen 0 nach)

```
    1100 0100 << 2
=   0001 0000
```

>> nach rechts schieben (Arithmetischer Rechtsshift)

- nach rechts schieben: >> → Schiebt die Bits um beliebig viele Stellen nach rechts.
- Dabei wird links immer das Vorzeichen (= das "linkste" Bit) hereingeschoben.

```
    1100 0100 >> 2
=   1111 0001

    0100 0100 >> 2
=   0001 0001
```

>>> nach rechts schieben (logischer Rechtsshift)

htl3 rennweg
IT & MECHATRONIK

» nach rechts schieben: >>> → Schiebt die Bits um beliebig viele Stellen nach rechts.
» Dabei wird links immer eine **Null** hereingeschoben.

```
    1100 0100 >>> 2
=   0011 0001

    0100 0100 >>> 2
=   0001 0001
```