Data rep - conversion between different sizes (type casting in C/C++) 1/3

- Grundregel 1: Beim **Verkleinern** von Ganzzahltypen wird das Bitmuster im übernommenen Teil beibehalten

- Beispiel:

```
int x = 53191; // 00000000 00000000 11001111 11000111
short sx = x;  //                   11001111 11000111
               // → -12345
```

Data rep - conversion between different sizes (type casting in C/C++) 2/3

- Grundregel 2: Beim **Vergrößern** von Ganzzahltypen wird das Vorzeichenbit nach links erweitert (=sign extension; wenn 1-> Erweiterung mit 1; wenn 0 -> Erweiterung mit 0)

- Beispiel (sign extension):

```
short sx = -12345;  //                    11001111 11000111
int x = sx;         //11111111 11111111 11001111 11000111
                    //→ -12345
```

Data rep - conversion between different sizes (type casting in C/C++) 3/3

- Grundregel 3: Beim Wandeln signed <-> unsigned bleibt das Bitmuster erhalten

- Beispiel:

```
int s = -1;          // 11111111 11111111 11111111 11111111
unsigned s2u=s;      // 11111111 11111111 11111111 11111111
                     //  → 4294967295
unsigned u = 2147483648;
                     // 10000000 00000000 00000000 00000000
int u2s=u;           //  → - 2147483648
```

Rückblick 1/2

- Warum ist nun

```
int x = 200 * 300 * 400 * 500; //-> x = -884 901 888
```

  - Rein mathematisch: x = 12‘000‘000‘000
  - Binärdarstellung:

x = 10‘11001011‘01000001‘01111000‘00000000

x = $\cancel{10‘}\underbrace{11001011‘01000001‘01111000‘00000000}$

$$T2U(x) = -2^{31} + 2^{30} + 2^{27} + 2^{25} + 2^{24} + 2^{22} + 2^{16} + 2^{14} + 2^{13} + 2^{12} + 2^{11}$$
$$= -884\,901\,888$$

Rückblick 2/2

- Lösung/ Vermeidung:
  - Verwendung ausreichend großer Variablen UND AUCH richtige Suffix Angabe der Literale
  - `unsigned long x = 200ul * 300ul * 400ul * 500ul; //~~64bit~~`
  - `unsigned long long x = 200ull * 300ull *  400ull * 500ull; //~~32bit~~`

Data rep - Bit manipulations - not, and, or, xor 1/2

- Da die binär Werte 0 und 1 die Kernwerte sind, wie Computer Daten kodieren, speichern und manipulieren hat die Boolsche Algebra eine gewisse Bedeutung
- Die Boolsche Algebra definiert Operationen, die mit Werten von 0 und 1 arbeiten, z.B.

	NOT	AND	OR	XOR (excl. or)
Funktions gleichung	$y = \overline{x1}$	$y = x1 \wedge x2$	$y = x1 \vee x2$	$y = x1 \oplus x2$
C bit-level	y= ~x1;	y= x1 & x2;	y= x1 | x2;	y = x1 ^ x2;
Wahrheitstabelle	see below	see below	see below	see below

NOT:

x_1	y
0	1
1	0

AND:

x_2	x_1	y
0	0	0
0	1	0
1	0	0
1	1	1

OR:

x_2	x_1	y
0	0	0
0	1	1
1	0	1
1	1	1

XOR (excl. or):

x_2	x_1	y
0	0	0
0	1	1
1	0	1
1	1	0

Data rep - Bit manipulations - not, and, or, xor 2/2

- Diese Operationen lassen sich auf Bit-Vektoren erweitern, bei denen auf jedem einzelnen Bit dann diese Operation ausgeführt wird.
- Beispiele:
  - $$\begin{array}{rr} & 0110 \\ \& & 1100 \\ \hline & 0100 \end{array} \qquad \begin{array}{rr} & 0110 \\ | & 1100 \\ \hline & 1110 \end{array} \qquad \begin{array}{rr} & 0110 \\ \wedge & 1100 \\ \hline & 1010 \end{array} \qquad \begin{array}{rr} \sim & 1100 \\ \hline & 0011 \end{array}$$

Data rep - Bit manipulations - logical operations 1/2

- Da die binär Werte 0 und 1 die Kernwerte sind, wie Computer Daten kodieren, speichern und manipulieren hat die Boolsche Algebra eine gewisse Bedeutung
- Die Boolsche Algebra definiert Operationen, die mit Werten von 0 und 1 arbeiten, z.B.

<table>
<tr><th></th><th>NOT</th><th>AND</th><th>OR</th></tr>
<tr><td>Funktions gleichung</td><td>$\mathbf{y} = \overline{\boldsymbol{x}\mathbf{1}}$</td><td>$\mathbf{y} = \boldsymbol{x}\mathbf{1} \wedge \boldsymbol{x}\mathbf{2}$</td><td>$\mathbf{y} = \boldsymbol{x}\mathbf{1} \vee \boldsymbol{x}\mathbf{2}$</td></tr>
<tr><td>C logical
(0=false; 1=true)</td><td>y= !x1;</td><td>y= x1 && x2;</td><td>y= x1 || x2;</td></tr>
<tr><td>Wahrheitstabelle</td>
<td><table><tr><th>x_1</th><th>y</th></tr><tr><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td></tr></table></td>
<td><table><tr><th>x_2</th><th>x_1</th><th>y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>0</td></tr><tr><td>1</td><td>0</td><td>0</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td>
<td><table><tr><th>x_2</th><th>x_1</th><th>y</th></tr><tr><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>1</td><td>1</td></tr><tr><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td></tr></table></td></tr>
</table>

Data rep - Bit manipulations - logical operations 2/2

- Beispiel:

Expression	Result
`!0x41`	`0x00`
`!0x00`	`0x01`
`!!0x41`	`0x01`
`0x69 && 0x55`	`0x01`
`0x69 | | 0x55`	`0x01`

Data rep - Bit manipulations - shift operations 1/2

- Left shift: x << k; //x um k stellen nach links schieben (und mit Nullen nachfüllen)
- Right shift: x >> k; //x um k stellen nach rechts schieben
  - 2 Varianten: logical shift und arithmetical shift
    - Logical shift: x um k stellen nach rechts schieben (und mit Nullen nachfüllen); wird für unsigned Datentypen verwendet
    - Arithmetical shift: x um k stellen nach rechts schieben (~~und mit 1-sen nachfüllen~~); wird für signed Datentypen verwendet (Vorzeichen muss erhalten bleiben!)

Data rep - Bit manipulations - shift operations 2/2

- Beispiel:

Operation	Value 1	Value 2
Argument `x`	[01100011]	[10010101]
`x << 4`	[0011*0000*]	[0101*0000*]
`x >> 4 (logical)`	[*0000*0110]	[*0000*1001]
`x >> 4` (arithmetic)	[*0000*0110]	[*1111*1001]

Int arithmetic - bin - unsigned - addition

- Die Binäre Addition kann ähnlich wie die klassische dezimale Addition per Hand durchgeführt werden

- Beispiel:

```
    0011 1010        (  58)
  + 0001 1011        (  27)
     111  1
  -----------
  = 0101 0101        (  85)
```

Int arithmetic - bin - unsigned - addition - Übung

- Binäre Addition 114dec + 53dec:

```
  0111 0010        (114)
+ 0011 0101        ( 53)
___________
=            ?     (   )
```

Int arithmetic - bin - unsigned - multiplication

- Multiplikation = $w - 1$ Additionen des verschobenen Multiplikators
- **Achtung!** Zwei w breite Zahlen können ein $2\,w$ breites Produkt ergeben
- Beispiel (3*11=33): //3= Multiplikant, 11= Multiplikator

```
0011  *  1011
    0011
     0000
      0011
       0011
     1111
= 00100001  //bin2dec(00100001)=32+1=33
```

Int arithmetic - bin - unsigned - multiplication - Übung

- 5dec * 26dec in Binärsystem berechnen:

*

=

Int arithmetic - bin - unsigned - subtraction - pos. result

- Die Binäre Subtraktion kann ähnlich wie die klassische dezimale Subtraktion per Hand durchgeführt werden

- Beispiel:

```
    0110 0101        (101) Minuend
  - 0011 0100        ( 52) Subtrahend
     11
  -----------
  = 0011 0001        ( 49)
```

Int arithmetic - bin - unsigned - substraction - Übung

- Binäre Subtraktion 114dec - 53dec:

```
  0111 0010          (114)
- 0011 0101          ( 53)
_____________
=                    (   )
```

Int arithmetic - bin - unsigned - subtraction - neg. result

- Die Binäre Subtraction kann ähnlich wie die klassische dezimale Subtraction per Hand durchgeführt werden

- Beispiel:

```
    0011 0100      (  52) Minuend
  - 0110 0101      ( 101) Subtrahend
   11   1 111
  -----------
  = 1100 1111      ( 207) !=(- 49)
```

Das klassische schriftliche Subtrahieren funktioniert nicht für ein negatives Ergebnis!

!!! Nicht richtig!!!
So bitte nicht rechnen!!!

-> Wie dann rechnen wenn Minuend < Subtrahend ?
-> -(subtrahend - minuend)
-> 52-101=-(101-52)!!!