

Data rep(resentation and int numeric)

Data rep(resentation) - motivation 1/2

- Warum soll man sich mit Datenrepräsentation (Darstellung von Zahlen und Zeichen) detaillierter beschäftigen?
- Um folgende (teure) Probleme zu vermeiden:



Ariane 5 (04.06.1996 37 s nach Start)
-Wert der Nutzlast 500 Mio \$
-Software teilweise von Ariane 4
übernommen
-Konvertierungsfehler 64-bit float →
16-bit int

Data rep - motivation 2/2

- Warum soll man sich mit Datenrepräsentation (Darstellung von Zahlen und Zeichen) detaillierter beschäftigen?
- Um folgende Phänomene zu verstehen:
 - `int x = 200 * 300 * 400 * 500; //-> x = -884 901 888`
 - `float y1 = (3.14 + 1e20) - 1e20; //-> y1 = 0, aber`
 - `float y2 = 3.14 + (1e20-1e20); //-> y2 = 3.14`

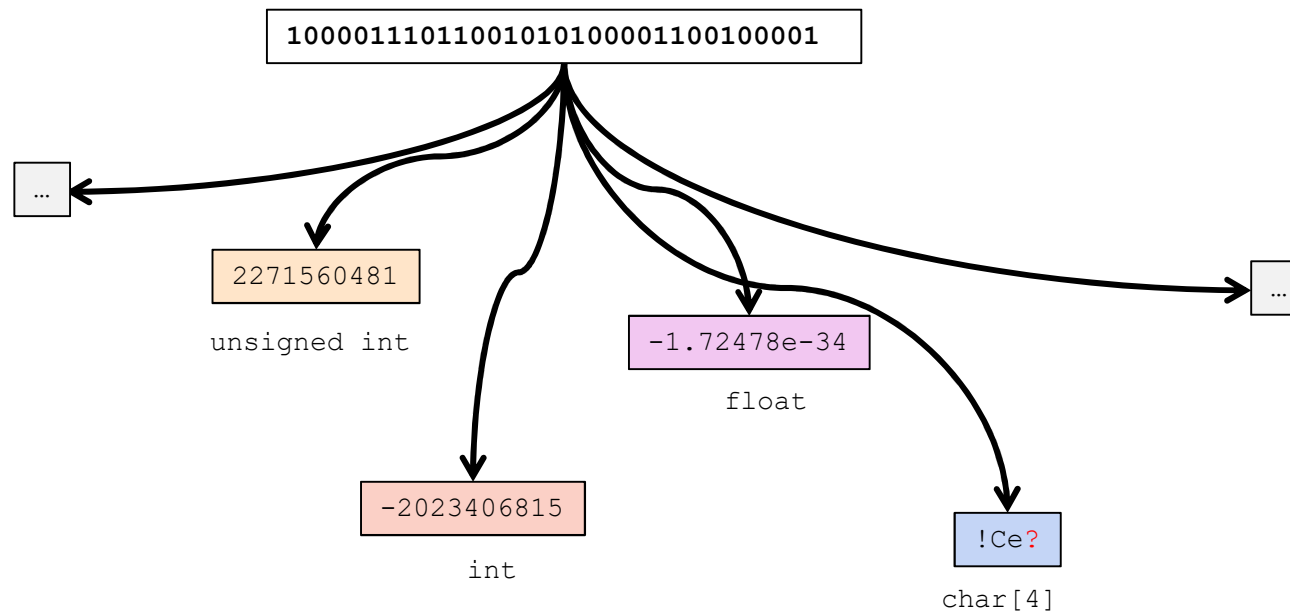
Data rep - Konventionen

- für uns soll gelten:
 - 1 KB = 1024 Byte, 1 MB = 1.048.576 Byte, 1 GB = 1.073.741.824 Byte ...
 - 1 kBit = 1000 Bit, 1 MBit/s = 1.000.000 Bit/s usw.

Data rep - Bit und Byte

- In Computersystemen werden alle Informationen z.B. ...
 - Texte (Zeichenketten) `"Text"`
 - Zahlen `42; 3,14; -123`
- ... durch Folgen/Gruppen von Bits dargestellt
 - Bit
 - = binary digit (dt. Binärziffer)
 - = zweiwertig: 0 oder 1
 - Byte= Folge/Gruppe von 8 Bits: `0b00000001`
 - Welche Bedeutung diese Bitfolge hat hängt von der interpretation ab (ob int, char, float, ...)

Data rep - Bit und Byte - Beispiele



Data rep - Binäre und Hexadezimale Darstellung

- Eine Folge von Bits kann als natürliche Zahl interpretiert werden
 - Wiederholung klassisches Dezimal-System:
 - Beispiel: $125 = 1 \cdot 10^2 + 2 \cdot 10^1 + 5 \cdot 10^0$
 - Binärsystem/Dualsystem:
 - Stellen haben hier Zweierpotenzen die von rechts nach links ansteigen
 - Beispiel: $0b101 = 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 5_{\text{dez}}$
 - Hexadezimalsystem:
 - Bei großen Zahlen ist die Binärdarstellung unhandlich, weshalb oft die Hexadezimal zur Anwendung kommt
 - Stellen haben hier 16er-potenzen die von rechts nach links ansteigen
 - Beispiel: $0x21 = 2 \cdot 16^1 + 1 \cdot 16^0 = 33_{\text{dez}}$
-

Data rep - Umwandlung bin -> dec und hex -> dec

- Umwandlung bin->dec
 - Einfach die Stellenwertigkeiten ausrechnen
 - Beispiel: $0b101 = 1*2^2 + 0*2^1 + 1*2^0 = 5_{\text{dez}}$

- Umwandlung hex->dec
 - Einfach die Stellenwertigkeiten ausrechnen
 - Beispiel: $0x21 = 2*16^1 + 1*16^0 = 33_{\text{dez}}$

Data rep - Umwandlung bin <-> hex

- Eine Umwandlung zwischen **bin** <-> **hex** erfolgt mit folgender Tabelle:

Hex digit	0	1	2	3	4	5	6	7
Decimal value	0	1	2	3	4	5	6	7
Binary value	0000	0001	0010	0011	0100	0101	0110	0111
Hex digit	8	9	A	B	C	D	E	F
Decimal value	8	9	10	11	12	13	14	15
Binary value	1000	1001	1010	1011	1100	1101	1110	1111

- Bin -> hex: 4 bin Stellen werden durch 1 hex Stelle ersetzt (immer von rechts aus (vom niederwertigsten Teil) anfangen!)
 - Beispiel: 0b11_1100_1010_1101_1011_0011 -> 0x3CADB3
- Hex -> bin: 1 hex Stelle wird durch 4 bin Stellen ersetzt
 - Beispiel: 0x173A4C -> 0b0001_0111_0011_1010_0100_1100

Data rep - Umwandlung dec -> bin

- Algorithmus dec2bin(d):
 - 1. Suche größte 2er Potenz
 $p=2^i \leq d$;
($i=\text{floor}(\log_2(d))$)
 - 2. Setze i-te Binärstelle in Ergebn. = 1
(!!!Rechteste Stelle ist 2^0 !!!)
 - 3. Bilde den Rest $d = d - 2^i$
 - 4. Wenn $d==0$ -> STOP;
ansonsten gehe wieder zu ->1
- Beispiel: dec2bin(17)
 - A1.: $p=2^i \leq 17 \rightarrow 2^4=16$
 - 2.: $\text{res}=0b1xxxx$
 - 3.: $d=17-16=1$
 - 4.: $d \neq 0$: -> weitermachen
 - B1.: $p=2^i < 1 \rightarrow 2^0=1$
 - 2.: $\text{res}=0b10001$
 - 3.: $d=1-1=0$
 - 4.: $d==0$: -> STOP

Data rep - Umwandlung dec -> bin


- Algorithmus dec2bin(d):

- Quotientenmethode

- Dividiere d und quotient
fortlaufend durch 2 und notiere
Rest bis quotient =0 (d div 2=0)
 - Rest von unten nach oben gelesen
ergibt das Ergebnis

- Beispiel: dec2bin(17)

d	d div 2	d mod2
17	8	1
8	4	0
4	2	0
2	1	0
1	0	1




- Dec2bin(17) =0b10001

Data rep - Umwandlung dec -> bin - Übung

- Algorithmus dec2bin(d):
 - Quotientenmethode
 - Dividiere d und quotient fortlaufend durch 2 und notiere Rest bis quotient =0 ($d \div 2 = 0$)
 - Rest von unten nach oben gelesen ergibt das Ergebnis

- Übung: dec2bin(78)

d	d div 2	d mod 2



Data rep - multi-byte Darstellungen

- Computersysteme verwenden typischerweise Bytes als kleinste (logisch) adressierbare Einheit
- Praktisch werden Daten in größeren Blöcken (32Bit platform typ. 4Byte / 32Bit oder 64Bit platform 8Byte / 64Bit) adressiert
- Solche multi-byte Datenblöcken werden im Speicher als zusammenhängende Bytesequenz gespeichert, die durch die kleinste Adresse adressiert werden
 - Beispiel: int ivar hat Adresse 0x100 -> int ist gespeichert in Adresse 0x100, 0x101, 0x102 und 0x103

Data rep - multi-byte Darstellungen - little /big endian 1/2

- Wie allerdings die einzelnen Bytes in den Adressen 0x100 bis 0x103 angeordnet sind, ist Architektur-/Herstellerabhängig
- Beispiel: `int ivar=0x01234567; //&ivar=0x100`

▪ Variante Big Endian



Big Endian=
höherwertigstes Byte auf
niedrigster Adresse
-> PPC (IBM), SPARC
(SUN / ORACLE)

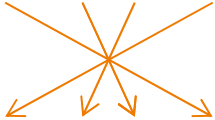
▪ Variante Little Endian



Little Endian=
niederwertiges Byte auf
niedrigster Adresse
-> x86 (Intel, AMD)

Data rep - multi-byte Darstellungen - little / big endian 2/2

- Die Endianness ist insbesondere auch bei Netzwerkübertragungen zwischen Systemen mit unterschiedlicher Endianness wichtig
- Das Berücksichtigen von little / big endian Darstellung ist insbesondere beim Disassemblieren (Befehl `objdump -d`) wichtig:

	Maschinencode	Assemblerbefehl
80483bd:	01 05 64 94 04 08	<code>add %eax, 0x8049464</code>
		
	08 04 94 64 = 8049464	

Data rep - multi-byte Darstellungen Übersicht Datentypen

▪ Übersicht Datentypen und Breite von Datentypen

C - Deklaration	32Bit	64Bit
char	1	1
short int	2	2
int	4	4
long int	4	8
long long int	8	8
char *	4	8
float	4	4
double	8	8

Data rep - char / C-string 1/2

ASCII Hex	ASCII Dec	char	ASCII Hex	ASCII Dec	char	ASCII Hex	ASCII Dec	char	ASCII Hex	ASCII Dec	char
00	0	NULL	20	32	SP	40	64	@	60	96	`
01	1	SOH	21	33	!	41	65	A	61	97	a
02	2	STX	22	34	"	42	66	B	62	98	b
03	3	ETX	23	35	#	43	67	C	63	99	c
04	4	EOT	24	36	\$	44	68	D	64	100	d
05	5	ENQ	25	37	%	45	69	E	65	101	e
06	6	ACK	26	38	&	46	70	F	66	102	f
07	7	BEL	27	39	'	47	71	G	67	103	g
08	8	BS	28	40	(48	72	H	68	104	h
09	9	TAB	29	41)	49	73	I	69	105	i
0A	10	LF	2A	42	*	4A	74	J	6A	106	j
0B	11	VT	2B	43	+	4B	75	K	6B	107	k
0C	12	FF	2C	44	,	4C	76	L	6C	108	l
0D	13	CR	2D	45	-	4D	77	M	6D	109	m
0E	14	SO	2E	46	.	4E	78	N	6E	110	n
0F	15	SI	2F	47	/	4F	79	O	6F	111	o
10	16	DLE	30	48	0	50	80	P	70	112	p
11	17	DC1	31	49	1	51	81	Q	71	113	q
12	18	DC2	32	50	2	52	82	R	72	114	r
13	19	DC3	33	51	3	53	83	S	73	115	s
14	20	DC4	34	52	4	54	84	T	74	116	t
15	21	NAK	35	53	5	55	85	U	75	117	u
16	22	SYN	36	54	6	56	86	V	76	118	v
17	23	ETB	37	55	7	57	87	W	77	119	w
18	24	CAN	38	56	8	58	88	X	78	120	x
19	25	EM	39	57	9	59	89	Y	79	121	y
1A	26	SUB	3A	58	:	5A	90	Z	7A	122	z
1B	27	Esc	3B	59	;	5B	91	[7B	123	{
1C	28	FS	3C	60	<	5C	92	\	7C	124	
1D	29	GS	3D	61	=	5D	93]	7D	125	}
1E	30	RS	3E	62	>	5E	94	^	7E	126	~
1F	31	US	3F	63	?	5F	95	_	7F	127	DEL

- char („Buchstaben Zeichen“) werden traditionell durch den ASCII Code dargestellt (7-Bit Code)
- Ein C-String ist eine folge aus einzelnen char, die durch ein NULL-byte (0x00) abgeschlossen werden
- Zahlen: 0x30-39
- Großbuchstaben: 0x41-5A
- Kleinbuchstaben: 0x61-7A

Data rep - char / C-string 2/2

- Nicht alle Buchstaben-Zeichen lassen sich damit darstellen (z.B. €, ä, ö , ...)
- -> Unicode (32bit, 4 Bytes pro Buchstabe)
 - -> jedes Zeichen benötigt 4 Byte statt 1 Byte
 - -> UTF-8
 - häufig verwendete Zeichen sind kurz (1 Byte) codiert (ASCII kompatibel!)
 - seltene Zeichen sind lang codiert (3+ Bytes)

Data rep - unsigned int

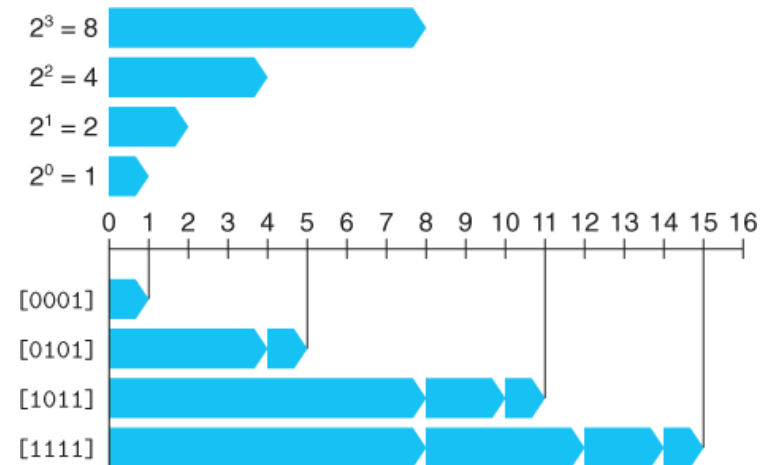
- Interpretation als natürliche Zahl mit Addition der jeweiligen Bitpositionen
 - Dezimalwert(x) =

$$B2U(x) = \sum_{i=0}^{w-1} x_i 2^i$$
 - $B2U$ = Binary - to - unsigned decimal
- Veranschaulichung als Vektor:
Gesamtlänge(=Zahlenwert)
= Addition der Teillängen

Beispiele

- Bin 0b0001, 0b0101, 0b1011, 0b1111

- Dec 1, 5, 11, 15



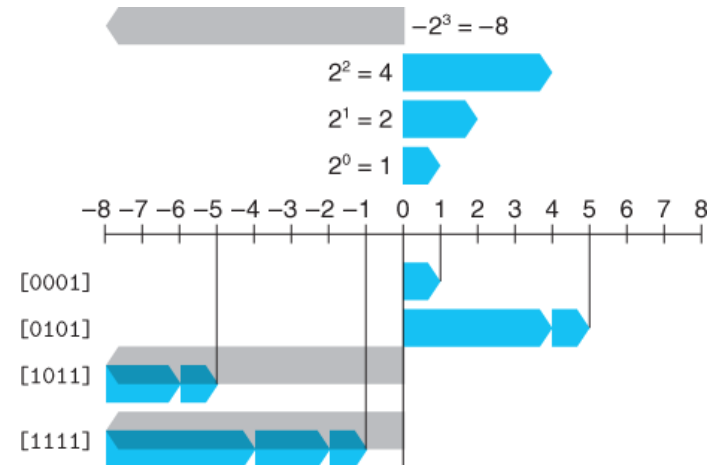
Data rep - unsigned int - Übung

- Dezimalwert(x) =
$$B2U(x) = \sum_{i=0}^{w-1} x_i 2^i$$
 - $B2U$ = Binary – to – unsigned decimal
- Übung: 8bit bin -> dec
 - 0x7A -> dec?
 - 0xFF -> dec?

Data rep - signed int 1/2

- Zweierkomplement Darstellung
 - $\text{Dezimalzahl}(x) = B2T(x) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$
 - $B2T$ = Binary - to - Two's Complement
- Veranschaulichung als Vektor:
 - Gesamtlänge(=Zahlenwert) = Addition der Teillängen
 - Höchstwertige Bitposition entspricht einem negativen Basisvektor (sign (Vorzeichen) ist enthalten!)

- Beispiele
 - Bin 0b0001, 0b0101, 0b1011, 0b1111
 - Dec 1, 5, -5, -1



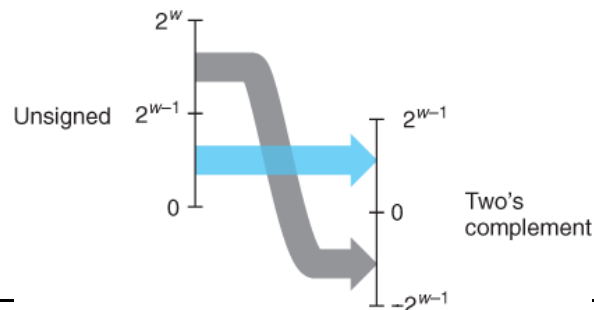
Data rep - signed int 2/2

- Zweierkomplement Darstellung
 - Dezimalzahl(x) = $B2T(x) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$
 - $B2T$ = Binary - to - Two's Complement
- Eigenschaften:
 - kleinste mit w Stellen darstellbare Zahl: -2^{w-1}
 - größte mit w Stellen darstellbare Zahl: $2^{w-1} - 1$
 - höchstwertiges Bit zeigt das Vorzeichen an ($x_{w-1} = 1 \rightarrow -$)
 - unveränderte Darstellung für nichtnegative Zahlen

Data rep – conversion unsigned -> signed ((tw)o's (c)omplement) 2/3

- Berechnung two's complement

- 0. Wenn positiv -> keine Änderung nötig -> fertig
- 1. Wenn negativ, ermittle das Bitmuster für den positiven Zahlenwert
- 2. Negiere alle Bitstellen (0 -> 1, 1 -> 0)
- 3. Addiere 1.



- Beispiel (8 bit):

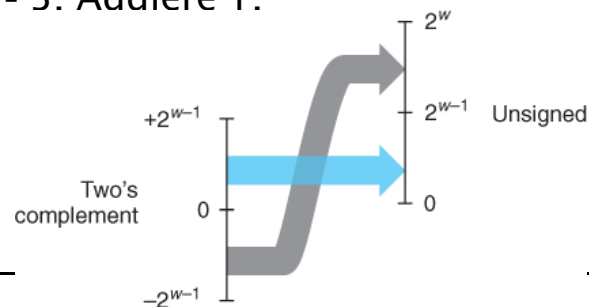
- Dec -17 -> TWC?

- 1.: 0001_0001
 - 2.: 1110_1110
 - 3.: 1110_1110
- $$\begin{array}{r}
 1 \\
 + 1 \\
 \hline
 1110_1111
 \end{array}$$

Achtung! Nur dann möglich wenn mit der eingeschränkten Darstellung des twcs möglich (betraglich nur noch $2^{(w-1)}$ statt 2^w)

Data rep - conversion signed (twc) -> unsigned 3/3

- conversion signed (twc) -> unsigned
 - 0. Wenn positiv -> keine Änderung nötig -> fertig
 - 1. Wenn negativ, ermittle das Bitmuster für den negativen Zahlenwert
 - 2. Negiere alle Bitstellen (0 -> 1, 1 -> 0)
 - 3. Addiere 1.



- Beispiel (8 bit):
 - 0xEF TWC -> Dec?
 - 1.: 1110_1111
 - 2.: 0001_0000
 - 3.: 0001_0000

+ 1

0001_0001

Data rep - conversion signed (twc) -> unsigned / dec -

Übung

- a) Zweierkomplement Darstellung
 - $\text{Dezimalzahl}(x) = B2T(x) = -x_{w-1}2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$
 - $B2T$ = Binary - to - Two's Complement
 - b) conversion signed (twc) -> unsigned
 - 0. Wenn positiv -> keine Änderung nötig -> fertig
 - 1. Wenn negativ, ermittle das Bitmuster für den negativen Zahlenwert
 - 2. Negiere alle Bitstellen (0 -> 1, 1 -> 0)
 - 3. Addiere 1.
- Übung 8bit twc -> dec
 - 0x7A -> dec?
 - 0xFF -> dec?

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 00000001 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
```

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)																																																			
Funktionsgleichung	$y = \overline{x1}$	$y = x1 \wedge x2$	$y = x1 \vee x2$	$y = x1 \oplus x2$																																																			
C bit-level	y= ~x1;	y= x1 & x2;	y= x1 x2;	y = x1 ^ x2;																																																			
Wahrheitstabelle	<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>	x_1	y	0	1	1	0	<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>	x_2	x_1	y	0	0	0	0	1	0	1	0	0	1	1	1	<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>	x_2	x_1	y	0	0	0	0	1	1	1	0	1	1	1	1	<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>0</td></tr></table>	x_2	x_1	y	0	0	0	0	1	1	1	0	1	1	1	0
x_1	y																																																						
0	1																																																						
1	0																																																						
x_2	x_1	y																																																					
0	0	0																																																					
0	1	0																																																					
1	0	0																																																					
1	1	1																																																					
x_2	x_1	y																																																					
0	0	0																																																					
0	1	1																																																					
1	0	1																																																					
1	1	1																																																					
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:

$$\begin{array}{rclcl}
 & 0110 & 0110 & 0110 & \\
 \& & 1100 & | & 1100 & \wedge & 1100 & \sim & 1100 \\
 & \overline{0100} & \overline{1110} & & \overline{1010} & & \overline{0011}
 \end{array}$$

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.

	NOT	AND	OR																																				
Funktionsgleichung	$y = \overline{x1}$	$y = x1 \wedge x2$	$y = x1 \vee x2$																																				
C logical (0=false; 1=true)	y= !x1;	y= x1 && x2;	y= x1 x2;																																				
Wahrheitstabelle	<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>	x_1	y	0	1	1	0	<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>	x_2	x_1	y	0	0	0	0	1	0	1	0	0	1	1	1	<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>	x_2	x_1	y	0	0	0	0	1	1	1	0	1	1	1	1
x_1	y																																						
0	1																																						
1	0																																						
x_2	x_1	y																																					
0	0	0																																					
0	1	0																																					
1	0	0																																					
1	1	1																																					
x_2	x_1	y																																					
0	0	0																																					
0	1	1																																					
1	0	1																																					
1	1	1																																					

Data rep - Bit manipulations - logical operations 2/2

▪ Beispiel:

Expression	Result
<code>!0x41</code>	<code>0x00</code>
<code>!0x00</code>	<code>0x01</code>
<code>!!0x41</code>	<code>0x01</code>
<code>0x69 && 0x55</code>	<code>0x01</code>
<code>0x69 0x55</code>	<code>0x01</code>

Data rep - Bit manipulations - shift operations 1/2

- Left shift: $x \ll k$; $//x$ um k stellen nach links schieben (und mit Nullen nachfüllen)
 - Right shift: $x \gg k$; $//x$ 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 <code>x</code>	[01100011]	[10010101]
<code>x << 4</code>	[00110000]	[01010000]
<code>x >> 4 (logical)</code>	[00000110]	[00001001]
<code>x >> 4 (arithmetic)</code>	[00000110]	[11111001]

Int arithmetic - unsigned - addition

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

- Beispiel:

$$\begin{array}{rcccccccc} & 0001 & 0010 & 0011 & 0100 & & (& 4660) \\ + & 1000 & 0111 & 0110 & 0101 & & (34661) \\ \hline & & 11 & 11 & 1 & & \\ = & 1001 & 1001 & 1001 & 1001 & & (39321) \end{array}$$

Int arithmetic - unsigned - addition - Übung

▪ Binäre Addition

$$\begin{array}{r} 0100\ 0100\ 0111\ 0010 \\ +\ 1000\ 0110\ 0011\ 0101 \\ \hline \end{array}$$

=

?

Int arithmetic - unsigned - multiplication

- Multiplikation = $w - 1$ Additionen des verschobenen Multiplikators
- **Achtung!** Zwei w breite Zahlen können ein $2w$ 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 - unsigned - multiplication - Übung

- 5dez * 26dez in Binärsystem berechnen:

*

=

Int arithmetic - signed (twc) - addition

- Die Regeln für die Binäre Addition für unsigned int kann genauso auf die Binäre Addition für signed int (twc) verwendet werden
- -> **Großer Vorteil -> weite Verbreitung des two's complement**
- Beispiel:

$$\begin{array}{rcl}
 0001\ 0010\ 0011\ 0100 & & (4660) \\
 +\ 1000\ 0111\ 0110\ 0101 & & (-30875) \\
 \hline
 & \textcolor{red}{11} & \textcolor{red}{11} & \textcolor{red}{1} \\
 =\ 1001\ 1001\ 1001\ 1001 & & (-26215)
 \end{array}$$

- Anm: Achtung overflow mit Vorzeichenwechsel!
 - $0x7F + 0x01 \rightarrow 0x80$
 - $(127 + 1 \rightarrow -128)$

Int arithmetic - signed (twc) - addition - Übung

- Berechnen Sie im Binärsystem durch Addition des TWCs
33dec - 7dec mit 16 Bit:

+

=

?

Int arithmetic - signed (twc) - multiplication

- Multiplikand und Multiplikator werden links mit Vorzeichenbit auf $2w$ Stellen aufgefüllt
- vom Ergebnis werden die $2w$ niederwertigsten Stellen verwendet
- Beispiel ($3 \cdot (-5) = -15$): $// -5: 5 = 0101 \rightarrow 1010 + 1 \rightarrow 1011 \rightarrow 11111011$

00000011 * 11111011

00000011

00000011

00000011

00000011

00000011

00000000

00000011

00000011

11111111

000001011110001

$//T2U(11110001) = -128 + 64 + 32 + 16 + 1 = -15$

oder

$T2U(11110001) = 0b00001110 + 1 = 0b0001111 = 15$

Rückblick 1/2

- Warum ist nun

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

- Rein mathematisch: $x = 12'000'000'000$

- Binärdarstellung:

$x = 10'11001011'01000001'01111000'00000000$

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

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

$$= -884\,901\,888$$

Rückblick 2/2

- Lösung/ Vermeidung:
 - Verwendung ausreichend großer Variablen UND AUCH richtige Suffix Angabe der Literale
 - `unsigned long x = 2001u * 3001u * 4001u * 5001u; //64bit`
 - `unsigned long long x = 2001lu * 3001lu * 4001lu * 5001lu; //32bit`