

64-040 Modul IP7: Rechnerstrukturen

[http://tams.informatik.uni-hamburg.de/
lectures/2012ws/vorlesung/rs](http://tams.informatik.uni-hamburg.de/lectures/2012ws/vorlesung/rs)

– Kapitel 8 –

Andreas Mäder



Universität Hamburg
Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Wintersemester 2012/2013

Kapitel 8

Logische Operationen

Boole'sche Algebra

Boole'sche Operationen

Bitweise logische Operationen

Schiebeoperationen

Anwendungsbeispiele

Speicher-Organisation

Literatur



Nutzen einer (abstrakten) Algebra?!

Analyse und Beschreibung von

- ▶ gemeinsamen, wichtigen Eigenschaften
- ▶ mathematischer Operationen
- ▶ mit vielfältigen Anwendungen

Spezifiziert durch

- ▶ die Art der Elemente (z.B. ganze Zahlen, Aussagen, usw.)
- ▶ die Verknüpfungen (z.B. Addition, Multiplikation)
- ▶ zentrale Elemente (z.B. Null-, Eins-, inverse Elemente)

Anwendungen: z.B. fehlerkorrigierende Codes auf CD/DVD

Boole'sche Algebra

- ▶ George Boole, 1850: Untersuchung von logischen Aussagen mit den Werten *true* (wahr) und *false* (falsch)
- ▶ Definition einer Algebra mit diesen Werten
- ▶ Vier grundlegende Funktionen:
 - ▶ NEGATION (NOT) Schreibweisen: $\neg a$, \bar{a} , $\sim a$
 - ▶ UND $-"-$ $a \wedge b$, $a \& b$
 - ▶ ODER $-"-$ $a \vee b$, $a \mid b$
 - ▶ XOR $-"-$ $a \oplus b$, $a \wedge b$
- ▶ Claude Shannon, 1937: Realisierung der Boole'schen Algebra mit Schaltfunktionen (binäre digitale Logik)

Grundverknüpfungen

- ▶ zwei Werte: *wahr* (*true*, 1) und *falsch* (*false*, 0)
- ▶ vier grundlegende Verknüpfungen:

NOT(x)

x	
0	1
1	0

AND(x, y)

y	x	0	1
0	0	0	0
1	0	0	1

OR(x, y)

y	x	0	1
0	0	0	1
1	1	1	1

XOR(x,y)

y	x	0	1
0	0	0	1
1	1	1	0

- ▶ alle logischen Operationen lassen sich mit diesen Funktionen darstellen

⇒ *vollständige Basismenge*

Anzahl der binären Funktionen

- ▶ insgesamt 4 Funktionen mit einer Variable
 $f_0(x) = 0, f_1(x) = 1, f_2(x) = x, f_3(x) = \neg x$

- ▶ insgesamt 16 Funktionen zweier Variablen (s. Beispiel)

- ▶ allgemein 2^{2^n} Funktionen von n Variablen

- ▶ später noch viele Beispiele

Anzahl der binären Funktionen (cont.)

$x =$ 0 1 0 1 $y =$ 0 0 1 1	Bezeichnung	Notation	Alternativnotation	Java/C-Notation
0 0 0 0	Nullfunktion	0		0
0 0 0 1	AND	$x \cap y$		$x \& y$
0 0 1 0	Inhibition	$y > x$		$y > x$
0 0 1 1	Identität y	y		y
0 1 0 0	Inhibition	$x > y$		$x > y$
0 1 0 1	Identität x	x		x
0 1 1 0	XOR	$x \oplus y$	$x \neq y$	$x != y$
0 1 1 1	OR	$x \cup y$		$x y$
1 0 0 0	NOR	$\neg(x \cup y)$		$!(x y)$
1 0 0 1	Äquivalenz	$\neg(x \oplus y)$	$x = y$	$x == y$
1 0 1 0	NICHT x	$\neg x$	x'	$!x$
1 0 1 1	Implikation	$x \leq y$	$x \rightarrow y$	$y \geq x$
1 1 0 0	NICHT y	$\neg y$	y'	$!y$
1 1 0 1	Implikation	$x \geq y$	$x \leftarrow y$	$x \geq y$
1 1 1 0	NAND	$\neg(x \cap y)$		$!(x \& y)$
1 1 1 1	Einsfunktion	1		1

Boole'sche Algebra

- ▶ 6-Tupel $\langle \{0, 1\}, \vee, \wedge, \neg, 0, 1 \rangle$ bildet eine Algebra
- ▶ $\{0, 1\}$ Menge mit zwei Elementen
- ▶ \vee ist die „Addition“
- ▶ \wedge ist die „Multiplikation“
- ▶ \neg ist das „Komplement“ (nicht das Inverse!)
- ▶ 0 (false) ist das Nullelement der Addition
- ▶ 1 (true) ist das Einselement der Multiplikation

Rechenregeln: Ring / Algebra

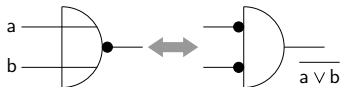
Eigenschaft	Ring der ganzen Zahlen	Boole'sche Algebra
Kommutativgesetz	$a + b = b + a$ $a \times b = b \times a$	$a \vee b = b \vee a$ $a \wedge b = b \wedge a$
Assoziativgesetz	$(a + b) + c = a + (b + c)$ $(a \times b) \times c = a \times (b \times c)$	$(a \vee b) \vee c = a \vee (b \vee c)$ $(a \wedge b) \wedge c = a \wedge (b \wedge c)$
Distributivgesetz	$a \times (b + c) = (a \times b) + (a \times c)$	$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
Identitäten	$a + 0 = a$ $a \times 1 = a$	$a \vee 0 = a$ $a \wedge 1 = a$
Vernichtung	$a \times 0 = 0$	$a \wedge 0 = 0$
Auslöschung	$-(-a) = a$	$\neg(\neg a) = a$
Inverses	$a + (-a) = 0$	—

Rechenregeln: Ring / Algebra (cont.)

Eigenschaft	Ring der ganzen Zahlen	Boole'sche Algebra
Distributivgesetz	—	$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$
Komplement	— —	$a \vee \neg a = 1$ $a \wedge \neg a = 0$
Idempotenz	— —	$a \vee a = a$ $a \wedge a = a$
Absorption	— —	$a \vee (a \wedge b) = a$ $a \wedge (a \vee b) = a$
De-Morgan Regeln	— —	$\neg(a \vee b) = \neg a \wedge \neg b$ $\neg(a \wedge b) = \neg a \vee \neg b$

De-Morgan Regeln

$$\neg(a \vee b) = \neg a \wedge \neg b$$



$$\neg(a \wedge b) = \neg a \vee \neg b$$



1. Ersetzen von *UND* durch *ODER* und umgekehrt
⇒ Austausch der Funktion
2. Invertieren aller Ein- und Ausgänge

Verwendung

- ▶ bei der Minimierung logischer Ausdrücke
- ▶ beim Entwurf von Schaltungen
- ▶ siehe Abschnitte: „Schaltfunktionen“ und „Schaltnetze“

XOR: Exklusiv-Oder / Antivalenz

⇒ entweder a oder b (ausschließlich)
 a ungleich b

(⇒ Antivalenz)

▶ $a \oplus b = (\neg a \wedge b) \vee (a \wedge \neg b)$
genau einer von den Termen a und b ist wahr

▶ $a \oplus b = (a \vee b) \wedge \neg(a \wedge b)$
entweder a ist wahr, oder b ist wahr, aber nicht beide gleichzeitig

▶ $a \oplus a = 0$

Logische Operationen in Java und C

- ▶ Datentyp für Boole'sche Logik
 - ▶ Java: Datentyp boolean
 - ▶ C: implizit für alle Integertypen
- ▶ Vergleichsoperationen
- ▶ Logische Grundoperationen
- ▶ Bitweise logische Operationen
 - = parallele Berechnung auf Integer-Datentypen
- ▶ Auswertungsreihenfolge
 - ▶ Operatorprioritäten
 - ▶ Auswertung von links nach rechts
 - ▶ (optionale) Klammerung

Vergleichsoperationen

- ▶ $a == b$ wahr, wenn a gleich b
- $a != b$ wahr, wenn a ungleich b
- $a >= b$ wahr, wenn a größer oder gleich b
- $a > b$ wahr, wenn a größer b
- $a < b$ wahr, wenn a kleiner b
- $a <= b$ wahr, wenn a kleiner oder gleich b

- ▶ Vergleich zweier Zahlen, Ergebnis ist logischer Wert
- ▶ Java: Integerwerte alle im Zweierkomplement
- C: Auswertung berücksichtigt signed/unsigned-Typen

Logische Operationen in C

► zusätzlich zu den Vergleichsoperatoren $<$, $<=$, $==$, $!=$, $>$, $>=$

► drei **logische** Operatoren:

! logische Negation

&& logisches UND

|| logisches ODER

► Interpretation der Integerwerte:

der Zahlenwert $0 \Leftrightarrow$ logische 0 (false)

alle anderen Werte \Leftrightarrow logische 1 (true)

\Rightarrow völlig andere Semantik als in der Mathematik

\Rightarrow völlig andere Funktion als die bitweisen Operationen

Achtung!

Logische Operationen in C (cont.)

- ▶ verkürzte Auswertung von links nach rechts (*shortcut*)
 - ▶ Abbruch, wenn Ergebnis feststeht
 - + kann zum Schutz von Ausdrücken benutzt werden
 - kann aber auch Seiteneffekte haben, z.B. Funktionsaufrufe

▶ Beispiele

- ▶ `(a > b) || ((b != c) && (b <= d))`

Ausdruck	Wert
<code>!0x41</code>	<code>0x00</code>
<code>!0x00</code>	<code>0x01</code>
<code>!!0x00</code>	<code>0x00</code>
<code>0x69 && 0x55</code>	<code>0x01</code>
<code>0x69 0x55</code>	<code>0x01</code>

Logische Operationen in C: Logisch vs. Bitweise

- der Zahlenwert 0 \Leftrightarrow logische 0 (false)
alle anderen Werte \Leftrightarrow logische 1 (true)
- Beispiel: $x = 0x66$ und $y = 0x93$

bitweise Operation		logische Operation	
Ausdruck	Wert	Ausdruck	Wert
x	0110 0110	x	0000 0001
y	1001 0011	y	0000 0001
x & y	0000 0010	x && y	0000 0001
x y	1111 0111	x y	0000 0001
~x ~y	1111 1101	!x !y	0000 0000
x & ~y	0110 0100	x && !y	0000 0000

Logische Operationen in C: verkürzte Auswertung

- ▶ logische Ausdrücke werden von links nach rechts ausgewertet
- ▶ Klammern werden natürlich berücksichtigt
- ▶ Abbruch, sobald der Wert eindeutig feststeht (*shortcut*)
- ▶ Vor- oder Nachteile möglich (codeabhängig)
 - + `(a && 5/a)` niemals Division durch Null. Der Quotient wird nur berechnet, wenn der linke Term ungleich Null ist.
 - + `(p && *p++)` niemals Nullpointer-Zugriff. Der Pointer wird nur verwendet, wenn `p` nicht Null ist.

Ternärer Operator

- ▶ `<condition> ? <true-expression> : <false-expression>`
- ▶ Beispiel: `(x < 0) ? -x : x` Absolutwert von `x`

Logische Operationen in Java

- ▶ Java definiert eigenen Datentyp `boolean`
- ▶ elementare Werte `false` und `true`
- ▶ alternativ `Boolean.FALSE` und `Boolean.TRUE`
- ▶ **keine** Mischung mit Integer-Werten wie in C

- ▶ Vergleichsoperatoren `<`, `<=`, `==`, `!=`, `>`, `>=`
- ▶ verkürzte Auswertung von links nach rechts (*shortcut*)

- ▶ Ternärer Operator
 $\langle \text{condition} \rangle ? \langle \text{true-expression} \rangle : \langle \text{false-expression} \rangle$
- ▶ Beispiel: $(x < 0) ? -x : x$ Absolutwert von x

Bitweise logische Operationen

Integer-Datentypen doppelt genutzt:

1. Zahlenwerte (Ganzzahl, Zweierkomplement, Gleitkomma)
arithmetische Operationen: Addition, Subtraktion, usw.

2. Binärwerte mit w einzelnen Bits (Wortbreite w)
Boole'sche Verknüpfungen, bitweise auf allen w Bits
 - ▶ Grundoperationen: Negation, UND, ODER, XOR
 - ▶ Schiebe-Operationen: shift-left, rotate-right, usw.

Bitweise logische Operationen (cont.)

- ▶ Integer-Datentypen interpretiert als Menge von Bits
- ⇒ bitweise logische Operationen möglich

- ▶ in Java und C sind vier Operationen definiert:

Negation	$\sim x$	Invertieren aller einzelnen Bits			
UND	$x \& y$	Logisches UND aller einzelnen Bits			
OR	$x y$	—"–	ODER	—"–	
XOR	$x \wedge y$	—"–	XOR	—"–	

- ▶ alle anderen Funktionen können damit dargestellt werden
es gibt insgesamt 2^{2^n} Operationen mit n Operanden

Bitweise logische Operationen: Beispiel

$x = 0010\ 1110$

$y = 1011\ 0011$

$\sim x = 1101\ 0001$ alle Bits invertiert

$\sim y = 0100\ 1100$ alle Bits invertiert

$x \ \& \ y = 0010\ 0010$ bitweises UND

$x \ | \ y = 1011\ 1111$ bitweises ODER

$x \ ^ \ y = 1001\ 1101$ bitweises XOR

Schiebeoperationen

- ▶ Ergänzung der bitweisen logischen Operationen
- ▶ für alle Integer-Datentypen verfügbar

- ▶ fünf Varianten

Shift-Left `shl`

Logical Shift-Right `srl`

Arithmetic Shift-Right `sra`

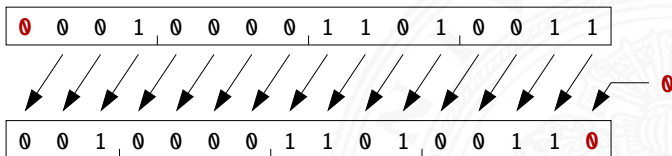
Rotate-Left `rol`

Rotate-Right `ror`

- ▶ Schiebeoperationen in Hardware leicht zu realisieren
- ▶ auf fast allen Prozessoren im Befehlssatz

Shift-Left (shl)

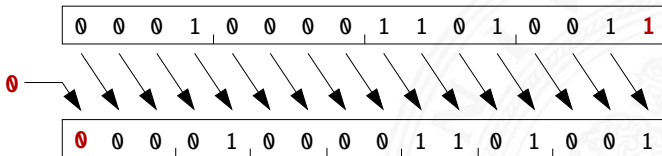
- ▶ Verschieben der Binärdarstellung von x um n bits nach links
- ▶ links herausgeschobene n bits gehen verloren
- ▶ von rechts werden n Nullen eingefügt



- ▶ in Java und C direkt als Operator verfügbar: $x \ll n$
- ▶ shl um n bits entspricht der Multiplikation mit 2^n

Logical Shift-Right (sr1)

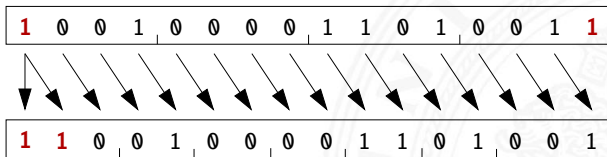
- ▶ Verschieben der Binärdarstellung von x um n bits nach rechts
- ▶ rechts herausgeschobene n bits gehen verloren
- ▶ von links werden n Nullen eingefügt



- ▶ in Java direkt als Operator verfügbar: $x \ggg n$
in C nur für unsigned-Typen definiert: $x \gg n$
für signed-Typen nicht vorhanden

Arithmetic Shift-Right (sra)

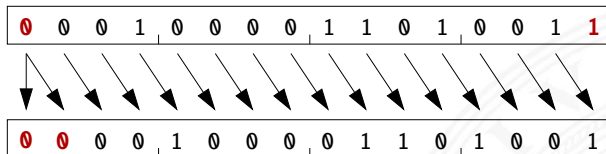
- ▶ Verschieben der Binärdarstellung von x um n bits nach rechts
- ▶ rechts herausgeschobene n bits gehen verloren
- ▶ von links wird n -mal das MSB (Vorzeichenbit) eingefügt
- ▶ Vorzeichen bleibt dabei erhalten (gemäß Zweierkomplement)



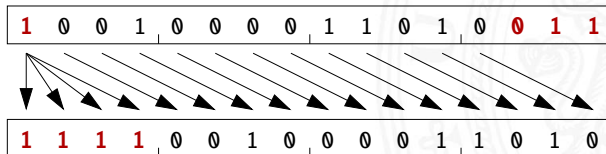
- ▶ in Java direkt als Operator verfügbar: $x \gg n$
in C nur für signed-Typen definiert: $x \gg n$
- ▶ sra um n bits ist ähnlich der Division durch 2^n

Arithmetic Shift-Right: Beispiel

- $x \gg 1$ aus $0x10D3$ (4307) wird $0x0869$ (2153)



- $x \gg 3$ aus $0x90D3$ (-28460) wird $0xF21A$ (-3558)



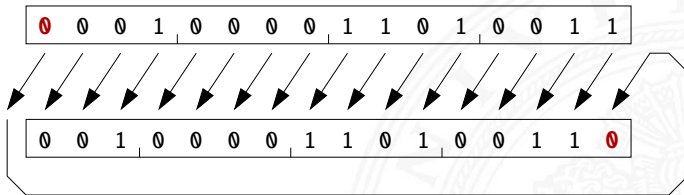
Arithmetic Shift-Right: Division durch Zweierpotenzen?

- ▶ positive Werte: $x \gg n$ entspricht Division durch 2^n
- ▶ negative Werte: $x \gg n$ Ergebnis ist zu klein (!)
- ▶ gerundet in Richtung negativer Werte statt in Richtung Null:

1111 1011	(-5)
1111 1101	(-3)
1111 1110	(-2)
1111 1111	(-1)
- ▶ in C: Kompensation durch Berechnung von $(x + (1 \ll k) - 1) \gg k$
Details: Bryant, O'Hallaron [BO11]

Rotate-Left (rol)

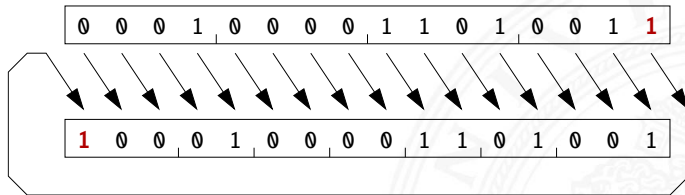
- ▶ Rotation der Binärdarstellung von x um n bits nach links
- ▶ herausgeschobene Bits werden von rechts wieder eingefügt



- ▶ in Java und C nicht als Operator verfügbar
- ▶ Java: `Integer.rotateLeft(int x, int distance)`

Rotate Right (ror)

- ▶ Rotation der Binärdarstellung von x um n bits nach rechts
- ▶ herausgeschobene Bits werden von links wieder eingefügt



- ▶ in Java und C nicht als Operator verfügbar
- ▶ Java: `Integer.rotateRight(int x, int distance)`

Shifts statt Integer-Multiplikation

- ▶ Integer-Multiplikation ist auf vielen Prozessoren langsam oder evtl. gar nicht als Befehl verfügbar
- ▶ Add./Subtraktion und logische Operationen: typisch 1 Takt
Shift-Operationen: meistens 1 Takt
- ⇒ eventuell günstig, Multiplikation mit Konstanten durch entsprechende Kombination aus shifts+add zu ersetzen
 - ▶ Beispiel: $9 \cdot x = (8 + 1) \cdot x$ ersetzt durch $(x \ll 3) + x$
 - ▶ viele Compiler erkennen solche Situationen

Beispiel: bit-set, bit-clear

Bits an Position p in einem Integer setzen oder löschen?

- ▶ Maske erstellen, die genau eine 1 gesetzt hat
- ▶ dies leistet $(1 \ll p)$, mit $0 \leq p \leq w$ bei Wortbreite w

```
public int bit_set( int x, int pos ) {
    return x | (1 << pos);          // mask = 0...010...0
}

public int bit_clear( int x, int pos ) {
    return x & ~(1 << pos);         // mask = 1...101...1
}
```


Beispiel: Byte-Swapping *network to/from host*

Linux: /usr/include/bits/byteswap.h

```
...
/* Swap bytes in 32 bit value.  */
#define __bswap_32(x) \
    (((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >> 8) |\
    (((x) & 0x0000ff00) << 8) | (((x) & 0x000000ff) << 24))
...
```

Linux: /usr/include/netinet/in.h

```
...
# if __BYTE_ORDER == __LITTLE_ENDIAN
#   define ntohl(x) __bswap_32 (x)
#   define ntohs(x) __bswap_16 (x)
#   define htonl(x) __bswap_32 (x)
#   define htons(x) __bswap_16 (x)
# endif
...
```

Beispiel: RGB-Format für Farbbilder

Farbdarstellung am Monitor / Bildverarbeitung?

- ▶ Matrix aus $w \times h$ Bildpunkten
- ▶ additive Farbmischung aus Rot, Grün, Blau
- ▶ pro Farbkanal typischerweise 8-bit, Wertebereich 0..255
- ▶ Abstufungen ausreichend für (untrainiertes) Auge
- ▶ je ein 32-bit Integer pro Bildpunkt
- ▶ typisch: `0x00RRGGBB` oder `0xAARRGGBB`
- ▶ je 8-bit für Alpha/Transparenz, rot, grün, blau
- ▶ `java.awt.image.BufferedImage(TYPE_INT_ARGB)`

Beispiel: RGB-Rotfilter

```
public BufferedImage redFilter( BufferedImage src ) {
    int    w = src.getWidth();
    int    h = src.getHeight();
    int type = BufferedImage.TYPE_INT_ARGB;
    BufferedImage dest = new BufferedImage( w, h, type );

    for( int y=0; y < h; y++ ) {           // alle Zeilen
        for( int x=0; x < w; x++ ) {        // von links nach rechts
            int  rgb = src.getRGB( x, y ); // Pixelwert bei (x,y)
                                           // rgb = 0xAARRGGBB
            int  red = (rgb & 0x00FF0000); // Rotanteil maskiert
            dest.setRGB( x, y, red );
        }
    }
    return dest;
}
```

Beispiel: RGB-Graufilter

```
public BufferedImage grayFilter( BufferedImage src ) {
    ...
    for( int y=0; y < h; y++ ) {    // alle Zeilen
        for( int x=0; x < w; x++ ) { // von links nach rechts
            int    rgb = src.getRGB( x, y );           // Pixelwert
            int    red = (rgb & 0x00FF0000) >>>16;    // Rotanteil
            int    green = (rgb & 0x0000FF00) >>> 8;   // Grünanteil
            int    blue = (rgb & 0x000000FF);          // Blauanteil

            int    gray = (red + green + blue) / 3;    // Mittelung

            dest.setRGB( x, y, (gray<<16)|(gray<<8)|gray );
        }
    }
    ...
}
```

Beispiel: Bitcount (mit while-Schleife)

Anzahl der gesetzten Bits in einem Wort?

- ▶ Anwendung z.B. für Kryptalgorithmen (Hamming-Distanz)
- ▶ Anwendung für Medienverarbeitung

```
public static int bitcount( int x ) {
    int count = 0;

    while( x != 0 ) {
        count += (x & 0x00000001); // unterstes bit addieren
        x = x >>> 1;             // 1-bit rechts-schieben
    }

    return count;
}
```

Beispiel: Bitcount (parallel, tree)

- ▶ Algorithmus mit Schleife ist einfach aber langsam
- ▶ schnelle parallele Berechnung ist möglich

```
int BitCount(unsigned int u)
{ unsigned int uCount;
  uCount = u - ((u >> 1) & 033333333333)
           - ((u >> 2) & 011111111111);
  return ((uCount + (uCount >> 3)) & 030707070707) % 63;
}
```

- ▶ viele Algorithmen: bit-Maskierung und Schieben
 - ▶ <http://gurmeet.net/puzzles/fast-bit-counting-routines>
 - ▶ <http://graphics.stanford.edu/~seander/bithacks.html>
 - ▶ D.E. Knuth: *The Art of Computer Programming*: Volume 4A, Combinational Algorithms: Part1, Abschnitt 7.1.3 [Knu09]
 - ▶ `java.lang.Integer.bitCount()`
- ▶ viele neuere Prozessoren/DSPs: eigener bitcount-Befehl

Tipps & Tricks: Rightmost bits

D. E. Knuth: *The Art of Computer Programming*, Vol 4.1 [Knu09]

Grundidee: am weitesten rechts stehenden 1-Bits / 1-Bit Folgen erzeugen Überträge in arithmetischen Operationen

► Integer x , mit $x = (\alpha 0 [1]^a 1 [0]^b)_2$

beliebiger Bitstring α , eine Null, dann $a + 1$ Einsen und b Nullen, mit $a \geq 0$ und $b \geq 0$.

► Ausnahmen: $x = -2^b$ und $x = 0$

$$\Rightarrow x = (\alpha 0 [1]^a 1 [0]^b)_2$$

$$\bar{x} = (\bar{\alpha} 1 [0]^a 0 [1]^b)_2$$

$$x - 1 = (\alpha 0 [1]^a 0 [1]^b)_2$$

$$-x = (\bar{\alpha} 1 [0]^a 1 [0]^b)_2$$

$$\Rightarrow \bar{x} + 1 = -x = \overline{x - 1}$$

Tipps & Tricks: Rightmost bits (cont.)

D. E. Knuth: *The Art of Computer Programming*, Vol 4.1 [Knu09]

$$\begin{aligned} x &= (\alpha 0 [1]^a 1 [0]^b)_2 & \bar{x} &= (\bar{\alpha} 1 [0]^a 0 [1]^b)_2 \\ x - 1 &= (\alpha 0 [1]^a 0 [1]^b)_2 & -x &= (\bar{\alpha} 1 [0]^a 1 [0]^b)_2 \end{aligned}$$

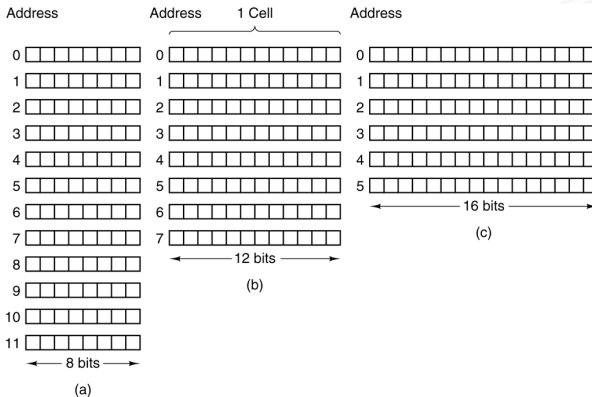
$x \& (x - 1) = (\alpha 0 [1]^a 0 [0]^b)_2$	letzte 1 entfernt
$x \& -x = (0^\infty 0 [0]^a 1 [0]^b)_2$	letzte 1 extrahiert
$x \mid -x = (1^\infty 1 [1]^a 1 [0]^b)_2$	letzte 1 nach links verschmiert
$x \oplus -x = (1^\infty 1 [1]^a 0 [0]^b)_2$	letzte 1 entfernt und verschmiert
$x \mid (x - 1) = (\alpha 0 [1]^a 1 [1]^b)_2$	letzte 1 nach rechts verschmiert
$\bar{x} \& (x - 1) = (0^\infty 0 [0]^a 0 [1]^b)_2$	letzte 1 nach rechts verschmiert
$((x \mid (x - 1)) + 1) \& x = (\alpha 0 [0]^a 0 [0]^b)_2$	letzte 1-Bit Folge entfernt

Aufbau und Adressierung des Speichers

- ▶ Abspeichern von Zahlen, Zeichen, Strings?
 - ▶ kleinster Datentyp üblicherweise ein Byte (8-bit)
 - ▶ andere Daten als Vielfache: 16-bit, 32-bit, 64-bit, ...
- ▶ Organisation und Adressierung des Speichers?
 - ▶ Adressen typisch in Bytes angegeben
 - ▶ erlaubt Adressierung einzelner ASCII-Zeichen, usw.
- ▶ aber Maschine/Prozessor arbeitet wortweise
- ▶ Speicher daher ebenfalls wortweise aufgebaut
- ▶ typischerweise 32-bit oder 64-bit

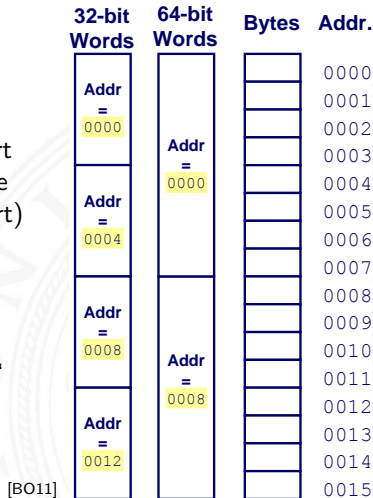
Speicher-Organisation

- Speicherkapazität: Anzahl der Worte \cdot Bits/Wort
- Beispiele: $12 \cdot 8$ $8 \cdot 12$ $6 \cdot 16$ Bits



Wort-basierte Organisation des Speichers

- ▶ Speicher Wort-orientiert
- ▶ Adressierung Byte-orientiert
 - ▶ die Adresse des ersten Bytes im Wort
 - ▶ Adressen aufeinanderfolgender Worte unterscheiden sich um 4 (32-bit Wort) oder 8 (64-bit)
 - ▶ Adressen normalerweise Vielfache der Wortlänge
 - ▶ verschobene Adressen „in der Mitte“ eines Worts oft unzulässig



Datentypen auf Maschinenebene

- ▶ gängige Prozessoren unterstützen mehrere Datentypen
- ▶ entsprechend der elementaren Datentypen in C, Java, ...
- ▶ `void*` ist ein **Pointer** (Referenz, Speicheradresse)
- ▶ Beispiel für die Anzahl der Bytes:

C Datentyp	DEC Alpha	typ. 32-bit	Intel IA-32 (x86)
int	4	4	4
long int	8	4	4
char	1	1	1
short	2	2	2
float	4	4	4
double	8	8	8
long double	8	8	10/12
void *	8	4	4

Datentypen auf Maschinenebene (cont.)

Abhängigkeiten (!)

- ▶ Prozessor
- ▶ Betriebssystem
- ▶ Compiler

segment word size	16 bit			32 bit						64 bit			
compiler	Microsoft	Borland	Watcom	Microsoft	Intel Windows	Borland	Watcom	Gnu v.3.x	Intel Linux	Microsoft	Intel Windows	Gnu	Intel Linux
bool	2	1	1	1	1	1	1	1	1	1	1	1	1
char	1	1	1	1	1	1	1	1	1	1	1	1	1
wchar_t		2		2	2	2	2	2	2	2	2	4	4
short int	2	2	2	2	2	2	2	2	2	2	2	2	2
int	2	2	2	4	4	4	4	4	4	4	4	4	4
long int	4	4	4	4	4	4	4	4	4	4	4	8	8
_int64				8	8			8	8	8	8	8	8
enum	2	2	1	4	4	4	4	4	4	4	4	4	4
float	4	4	4	4	4	4	4	4	4	4	4	4	4
double	8	8	8	8	8	8	8	8	8	8	8	8	8
long double	10	10	8	8	16	10	8	12	12	8	16	16	16
_m64				8	8				8		8	8	8
_m128				16	16				16	16	16	16	16
_m256					32				32		32		32
pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
far pointer	4	4	4										
function pointer	2	2	2	4	4	4	4	4	4	8	8	8	8
data member pointer (min)	2	4	6	4	4	8	4	4	4	4	4	8	8
data member pointer (max)		4	6	12	12	8	12	4	4	12	12	8	8
member function pointer (min)	2	12	6	4	4	12	4	8	8	8	8	16	16
member function pointer (max)		12	6	16	16	12	16	8	8	24	24	16	16

Table 1 shows how many bytes of storage various objects use for different compilers.

www.agner.org/optimize/calling_conventions.pdf

Byte-Order

- ▶ *Wie sollen die Bytes innerhalb eines Wortes angeordnet werden?*
- ▶ Speicher wort-basiert \Leftrightarrow Adressierung byte-basiert

Zwei Möglichkeiten / Konventionen:

- ▶ **Big Endian:** Sun, Mac, usw.
das MSB (*most significant byte*) hat die kleinste Adresse
das LSB (*least significant byte*) hat die höchste —"
- ▶ **Little Endian:** Alpha, x86
das MSB hat die höchste, das LSB die kleinste Adresse

satirische Referenz auf Gulliver's Reisen (Jonathan Swift)

Byte-Order: Beispiel

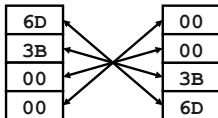
```
int A = 15213;
int B = -15213;
long int C = 15213;
```

Dezimal: 15213

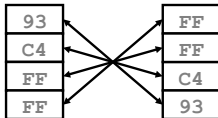
Binär: 0011 1011 0110 1101

Hex: 3 B 6 D

Linux/Alpha A Sun A



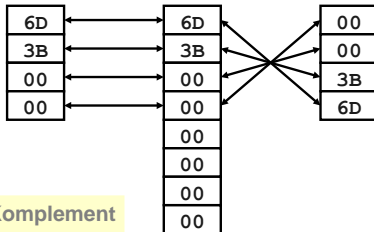
Linux/Alpha B Sun B



Linux C

Alpha C

Sun C



2-Komplement

Big Endian

Little Endian

Byte-Order: Beispiel Datenstruktur

```
/* JimSmith.c - example record for byte-order demo */

typedef struct employee {
    int    age;
    int    salary;
    char    name[12];
} employee_t;

static employee_t jimmy = {
    23,                // 0x0017
    50000,             // 0xc350
    "Jim Smith",       // J=0x4a i=0x69 usw.
};
```


Byte-Order: Beispiel x86 und SPARC

```
tams12> objdump -s JimSmith.x86.o
JimSmith.x86.o:      file format elf32-i386

Contents of section .data:
 0000 17000000 50c30000 4a696d20 536d6974  ....P...Jim Smit
 0010 68000000                               h...

tams12> objdump -s JimSmith.sparc.o
JimSmith.sparc.o:    file format elf32-sparc

Contents of section .data:
 0000 00000017 0000c350 4a696d20 536d6974  ....PJim Smit
 0010 68000000                               h...
```

Netzwerk Byte-Order

- ▶ Byte-Order muss bei Datenübertragung zwischen Rechnern berücksichtigt und eingehalten werden
- ▶ Internet-Protokoll (IP) nutzt ein Big Endian Format
- ⇒ auf x86-Rechnern müssen alle ausgehenden und ankommenden Datenpakete umgewandelt werden
- ▶ zugehörige Hilfsfunktionen / Makros in `netinet/in.h`
 - ▶ inaktiv auf Big Endian, **byte-swapping** auf Little Endian
 - ▶ `ntohl(x)`: network-to-host-long
 - ▶ `htons(x)`: host-to-network-short
 - ▶ ...

Beispiel: Byte-Swapping *network to/from host*

Linux: /usr/include/bits/byteswap.h

```
...
/* Swap bytes in 32 bit value.  */
#define __bswap_32(x) \
    (((x) & 0xff000000) >> 24) | (((x) & 0x00ff0000) >> 8) |\
    (((x) & 0x0000ff00) << 8) | (((x) & 0x000000ff) << 24))
...
```

Linux: /usr/include/netinet/in.h

```
...
# if __BYTE_ORDER == __LITTLE_ENDIAN
#   define ntohl(x) __bswap_32 (x)
#   define ntohs(x) __bswap_16 (x)
#   define htonl(x) __bswap_32 (x)
#   define htons(x) __bswap_16 (x)
# endif
...
```

Programm zum Erkennen der Byte-Order

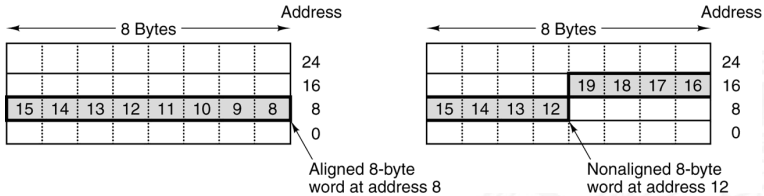
- ▶ Programm gibt Daten byteweise aus
- ▶ C-spezifische Typ- (Pointer-) Konvertierung
- ▶ Details: Bryant, O'Hallaron: 2.1.4 (Abb. 2.3, 2.4) [BO11]

```
void show_bytes( byte_pointer start, int len ) {
    int i;
    for( i=0; i < len; i++ ) {
        printf( " %.2x", start[i] );
    }
    printf( "\n" );
}

void show_double( double x ) {
    show_bytes( (byte_pointer) &x, sizeof( double ));
}

...
```

Misaligned Memory Access



[Tan09]

- Speicher Byte-weise adressiert
- aber Zugriffe lesen/schreiben jeweils ein ganzes Wort

Was passiert bei „krummen“ (*misaligned*) Adressen?

- automatische Umsetzung auf mehrere Zugriffe
- Programmabbruch

(x86)
(SPARC)

Literatur

[BO11] R.E. Bryant, D.R. O'Hallaron:

Computer systems – A programmers perspective.

2nd edition, Pearson, 2011. ISBN 0-13-713336-7

[Knu09] D.E. Knuth: *The Art of Computer Programming, Volume 4, Fascicle 1, Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 2009. ISBN 978-0-321-58050-4

[Tan06] A.S. Tanenbaum: *Computerarchitektur: Strukturen, Konzepte, Grundlagen.* 5. Auflage, Pearson Studium, 2006. ISBN 3-8273-7151-1

Literatur (cont.)

[Tan09] A.S. Tanenbaum: *Structured Computer Organization*.
5th rev. edition, Pearson International, 2009.
ISBN 0-13-509405-4

[Hei05] K. von der Heide: *Vorlesung: Technische Informatik 1 —
interaktives Skript*. Universität Hamburg, FB Informatik, 2005.
tams.informatik.uni-hamburg.de/lectures/2004ws/vorlesung/t1